FrontPage

Advanced Weather Interactive Processing System II (AWIPS II) - Software System Design Description

Document No. AWPR.DSN.A2.SSDD-08.00
1 March 2023

Prepared Under
Contract EA133W-17-CQ-0082
Advanced Weather Interactive Processing System (AWIPS)
Operations and Maintenance

Submitted to:
Mr. Timothy Wampler
Contracting Officer
U.S. Department of Commerce
National Oceanic and Atmospheric Administration
Eastern Region Acquisition Division
Room 815
200 Granby Street
Norfolk, Virginia 23510

Prepared by:


Raytheon Technologies
8401 Colesville Road, Suite 800
Silver Spring, MD 20910

Revision History

Document No.	Publication Date	Section(s) Affected	Description of Change(s)
AWPR.DSN.A2.SSDD-01.00	1 March 2018	N/A	Bi-Annual Review
AWPR.DSN.A2.SSDD-03.00	9 September 2020	N/A	Bi-Annual Review
AWPR.DSN.A2.SSDD-04.00	1 March 2021	N/A	Bi-Annual Review
AWPR.DSN.A2.SSDD-05.00	1 September 2021	N/A	Bi-Annual Review
AWPR.DSN.A2.SSDD-06.00	1 March 2022	N/A	Bi-Annual Review
AWPR.DSN.A2.SSDD-07.00	12 September 2022	N/A	Bi-Annual Review
AWPR.DSN.A2.SSDD-08.00	1 March 2023	N/A	Bi-Annual Review

List of Tables

List of Figures

Preface

This Software System Design Description (SSDD) provides AWIPS II software developers with a reference of key information when developing code in the AWIPS II environment. The document is topically organized, simple, and straightforward; using the Table of Contents is an easy way to find a topic of interest.

This Preface provides a bit of background on the purpose and scope of the SSDD, assumptions made about its users and uses, and the AWIPS II operational and development environments. [**Note:** For a list of acronyms and abbreviations used in the SSDD, see Appendix A.]

SSDD Motivation, Assumptions, Contents
AWIPS II Environment, Development Approach, Driving Requirements

---

Overview

AWIPS II Architecture
Plugins
Use of ADE

RPM Overview

Common

EDEX, Common, and Viz (Visualization) Plugins
UFStatus
Localization
Dynamic Serialization
JAXB Serialization
TopoAccess
JMS and QPID
Creating a New PluginDataObject Derived Class
Point Data
GeoTools and JTS Use - Best Practices
Python
The Python Virtual Environment
IDataStore
Python Job Coordinator
Data Access Framework
Adding and Upgrading Java FOSS

EDEX

EDEX Camel Spring
Camel EDEX Adapters
Thread Pools - Usage of Generic Decoder
EDEX Data Routing
Persistence, Hibernate, Postgres, and CoreDao
EDEX Decoder Plugins
PluginRegistry
AWIPS II Data Purging
Request JVM
Clustering
AWIPS II deploy-install.xml
Logging Configuration
Uframe feature.xml
Ignite

CAVE

RCP Framework
SWT
How to Write Dialogs for CAVE Classes

How to add performance logging in CAVE Classes
Menu Customization
CAVE Resources
CAVE Alert Observer
CAVE Features
CAVE Maps
CAVE - Right-Clicking In Editor
CAVE - Right-Clicking on the Legends

Derived Parameters
CAVE Graphics Tips
CAVE Performance Logging

Data Flow

Standard AWIPS Data and Notification Flow
How Does Ingested Data Get Into CAVE?
Special Case Ingest Using Manual Dropped-in Files

Appendix A. Acronyms and Abbreviations

ADE	Eclipse/AWIPS Development Environment
AMQP	Advanced Messaging Queuing Protocol
API	Application Program Interface
ARSR	Air Route Surveillance Radar
ASR	Aggregation Service Routers
AWIPS	Advanced Weather Interactive Processing System
CAVE	Common AWIPS Visualization Environment
CONUS	Conterminous/Contiguous/Commercial United States
COTS	Commercial off the shelf
D2D	Display Two Dimensional
DAO	Data Access Object
DVB	Digital Video Broadcast
EDEX	Enterprise Data Exchange
ESB	Enterprise Service Bus
FOSS	Free and Open Source Software
FQDN	Fully Qualified Domain Name
GFE	Graphical Forecast Editor
GFS	Global Forecast System
GNU	GNUs Not Unix
GRIB	Gridded Binary
GUI	Graphical User Interface
HDF5	Hierarchical Data Format 5-multi-object file format for the transfer of graphical and numerical data between computers
HQL	Hibernate Query Language
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
JAXB	Java Architecture for XML Binding
JEP	Java Embedded Python
JMS	Java Messaging Service
JVM	Java Virtual Machine
JTS	Java Transaction Service
LDM	Local Data Manager
LSB	Linux Standards Base
MOS	Model Output Statistics
MPE	Multi-Programming Executive
NWS	National Weather Service
OCONUS	Outside Conterminous/Contiguous/Commercial United States

OSGi	Open Services Gateway initiative
PDO	Plugin Data Object
PYPIES	Python Processing Isolated Enhanced Storage
QPID	Queue Processor Interface Daemon
RCP	Rich Client Platform
RGB	Red, Green, Blue
RPM	Redhat Package Manager
SOA	Service Oriented Architecture
SBN	Satellite Broadcast Network
SEDA	Staged Event Driven Architecture
SHEF	Standard Hydrometeorology Exchange
SQL	Structured Query Language
SSDD	Software System Design Description
SWT	Standard Widget Toolkit
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
VIZ	Visualization
XML	eXtensible Markup Language
YUM	Yellowdog Updater Modified

CAVE Resources

Every item drawn on a display in CAVE is being drawn by a resource. A resource is divided into two classes, the resource itself and the resource data. The resource data is responsible for constructing the resource; it is also the part of the resource that is serialized when displays are saved, The resource is responsible for actually drawing on the display.

Every resource must extend AbstractVizResource. Every resourceData object must extend the AbstractResourceData. For data that is being displayed from PluginDataObjects it is often better to extend AbstractRequestableResourceData because this class provides some additional help in requesting and managing the data in D2D. The rest of this document will explain what needs to be done to extend these Abstract classes in order to get a functional display in CAVE.

AbstractVizResource

All resources must extend AbstractVizResource. This is the class that is responsible for drawing data on the display and any other user interaction. It should use information from the resource data to determine what needs to be drawn.

There are three methods that must be implemented (paintInternal, initInternal, and disposeInternal). There are other methods that can be overridden to provide more functionality. Descriptions of five methods follow.

- **paintInternal.** This is the reason you have a resource, to draw something on the screen. Use the methods of the graphics target to draw whatever needs to be drawn. You should try to avoid doing anything other than painting in this method; for example, requesting data over the network or reading a file should not be handled here, because doing so can lead to the whole application hanging if it takes longer than expected. Often a Job is scheduled within paint to handle these things, or they can be handled in initInternal.
- **initInternal.** This method is called before the first time a resource is painted, and it is called in a background thread. initInternal allows you to perform any tasks to prepare to paint. You may need to request data or load files in this method, or you may want to prepare some of the graphics resources you will be using in paint. For simple or new resources, this method may be very small or empty.
- **disposeInternal.** This is the opposite of initInternal. It is the method called on a resource; the resource will not paint after this. The most important thing here is to call dispose on any graphics resources you may be using.
- **getName.** This method allows you to set a name that is displayed in the legend; in D2D the time will be appended to this name automatically by the legend resource.
- **Inspect.** This is the method used to display information when sampling is enabled. If you implement this method you should use the coordinate provided to find what you are displaying under or close to that point and return a String containing any additional information the user might be interested in.

AbstractResourceData vs. AbstractRequestableResourceData

The two classes you might be extending for your resource data are AbstractResourceData and AbstractRequestableResourceData.

If you are writing a data plugin to display data from PluginDataObjects, it is better to extend AbstractRequestableResourceData. This is used for almost every datatype in D2D, including satellite, grib, and radar.

For anything else you want to be displayed, you should extend AbstractResourceData. The best example of this is maps, although some system resources also extend this, e.g., the colorbar and legend resources.

AbstractResourceData

This is the base class for implementing a resource data; even AbstractRequestableResourceData extends this class. The two tasks the resource data must perform are constructing the resource and serializing any data needed to reload the resource from a bundle.

Constructing the resource can be as simple as calling returning a new resource object; it is usually not much more complex than this.

To be serialized a resource data class will need xml annotations on any part of the resource data that needs to be persisted. Also your resourceData class should be added to the **com.raytheon.uf.common.serialization.ISerializableObject** file.

Here is a brief description of the functions you will want to override if you extend this class.

- **construct.** This is the method that generally does the most work; it just needs to return a resource to draw on the display.
- **update.** This function was meant to provide updates to your resource, but it is often unused, except by AbstractRequestableResourceData (see the following subsection AbstractRequestableResourceData).
- **equals.** By implementing equals, a resource data can ensure that the same resource is not loaded twice on one display. When two resource datas are equal, the descriptor will only include one in the display.

AbstractRequestableResourceData

This is the class to extend if you are writing a data plugin for D2D. This class handles many of the details needed to correctly time match and display in D2D.

AbstractRequestableResourceData works by providing a metadata map. This map limits what data can be requested. When you create a resource data, either through a bundle or through the product browser you will fill in this map to limit what can be loaded for a resource. The metadata map is used to populate menu times automatically for bundle menu items. The metadatamap is used to retrieve available times for time matching. The metadatamap is used to request PluginDataObjects for your resource.

When you extend this class you will not need to implement construct or update from AbstractResourceData, instead you only need to implement constructResource. constructResource serves the same basic purpose as construct in AbstractResourceData but it provides you with data object that have already been retrieved and time matched for your resource. Most of the time this class will construct a resource and add in the provided data objects before returning it.

When implementing a resource for AbstractRequestableResourceData, you will still extend AbstractVizResource, but there are some extra things you can do to work smoothly in D2D:

- **Add an IResourceDataChanged listener to the resource data.** Many resources simply extend this interface themselves and add themselves as a listener. You should listen specifically for DATA_UPDATE changes since this will contain any new data objects you need to display. Updates will be sent automatically when new data arrives or if the user changes frames or other time matching options
- **Manage the dataTimes list.** Every time a new record is added, add the time to the list. The default remove method will automatically remove old times so if you override remove be sure to call super. The dataTimes list will be used by the time matcher along with updates and remove to manage your data for you.
- **Get the current data time.** In paint you will need to get the current data time from the paintProps and display any matching records.

Resource Interfaces

There are various interfaces that resources may implement to add additional functionality, depending on the needs of the resource:

- IExtraTextGeneratingResource

- IGraphableResource
- IInsetMapResource
- IHodographResource
- IRangeableResource
- IMiddleClickCapableResource

Use of ADE

The ADE is the Eclipse/AWIPS Development Environment. It is Eclipse RCP packaged with the AWIPS II baseline plugins. It is where plugin development occurs. Documentation on how to use the Eclipse in general can be found on the Eclipse documentation websites. How to install and set up the ADE is provided in the **AWIPS Flow Tag Instructions: ADE Setup** guide (links to this and other documents can be found in the AWIPS Release Documents wiki (/group/awips-community/library#awips-release-docs)). This section covers certain aspects of the ADE that will help with development.

Plugin Creation

The following subsections provide the steps required to create each type of plugin. There are different steps for creating Common/EDEX, CAVE, and COTS/FOSS plugins.

Common/EDEX

1. File->New->Project...
2. Select **Plug-in Development/Plug-in Project** and click "Next" as shown in **Figure 1-2.**
3. Provide project name based on **Plug-in Naming** section as shown in **Figure 1-3.**
4. If default location of the plugin to be created is incorrect, replace with correct path. THIS PATH MUST INCLUDE THE PLUGIN NAME.
5. If remaining default settings are sufficient, select "Next."
6. Replace "Name:" section with a more descriptive name as shown in **Figure 1-4.**
7. **UN-check** "Generate an activator" **and** "This plug-in will make contributions to the UI."
8. Select "Finish."

CAVE

1. File->New->Project...
2. Select **Plug-in Development/Plug-in Project** and click "Next," as shown in **Figure 1-2.**
3. Provide project name based on **Plug-in Naming** section as shown in **Figure 1-3.** If default location of plugin to be created is incorrect, replace with correct path. THIS PATH MUST INCLUDE THE PLUGIN NAME.
4. If remaining default settings are sufficient, select "Next."
5. Replace "Name:" section with a more descriptive name as shown in **Figure 1-4.**
6. **CHECK** "Generate an activator" **and** "This plug-in will make contributions to the UI."
7. Select "Finish."

COTS/FOSS

1. File->New->Project...
2. Select **Plug-in Development/Plug-in from Existing JAR Archives** and click "Next" as shown in **Figure 1-2.**
3. Click "Add External..." and browse to the folder the COTS JARs are in, select them, and click "Open" as shown in **Figure 1-5.**
4. Once all JARs are added, click "Next."
5. Provide project name based on **Plug-in Naming** section as shown in **Figure 1-6.**
6. If default location of plugin to be created is incorrect, replace with correct path. THIS PATH MUST INCLUDE THE PROJECT NAME FROM STEP 5.
7. Replace "Plug-in Name:" section with more descriptive name.
8. **UN-check** "Unzip the JAR archives into the project."
9. Select "Finish."

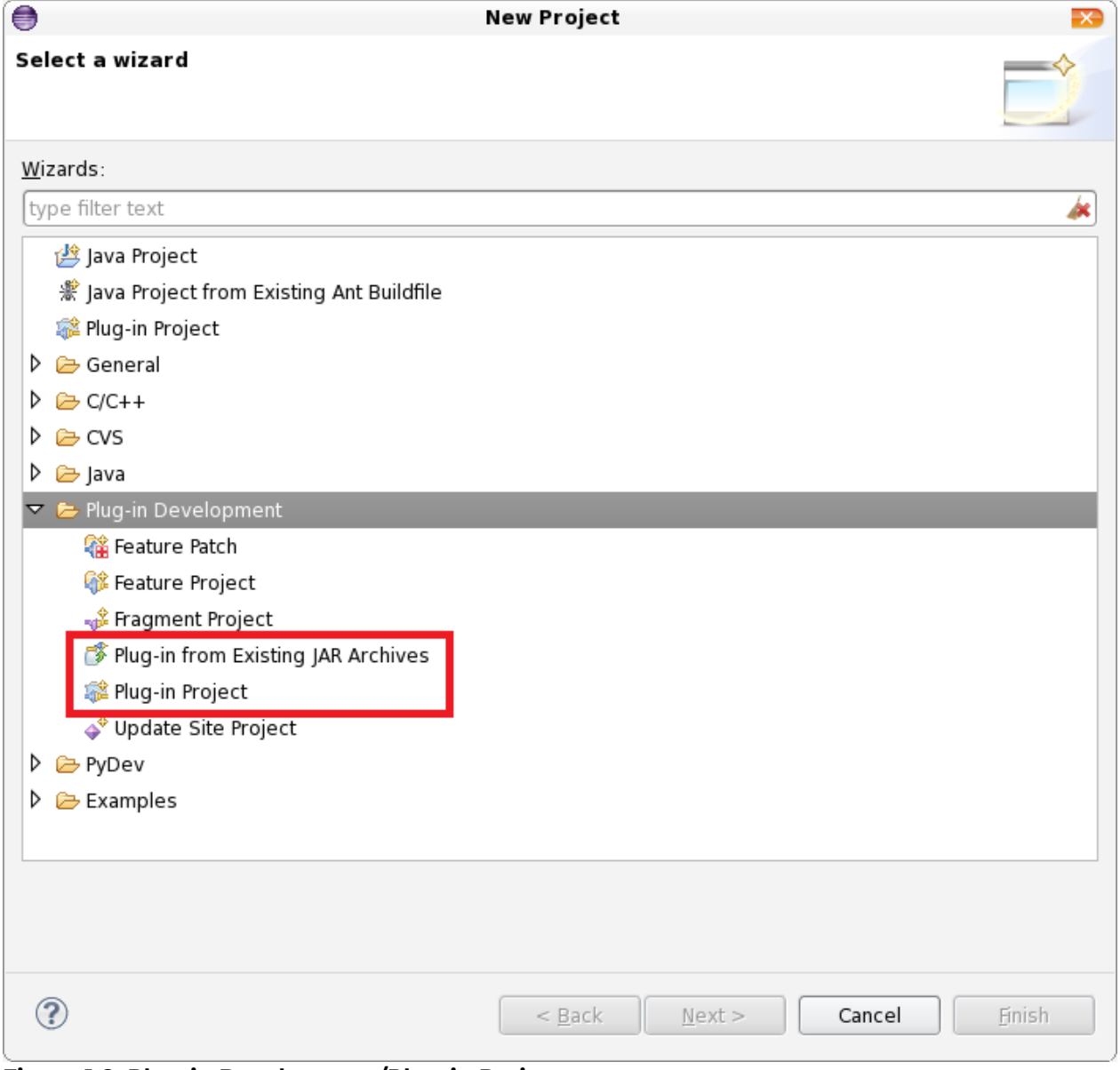

Figure 1-2. Plug-in Development/Plug-in Project

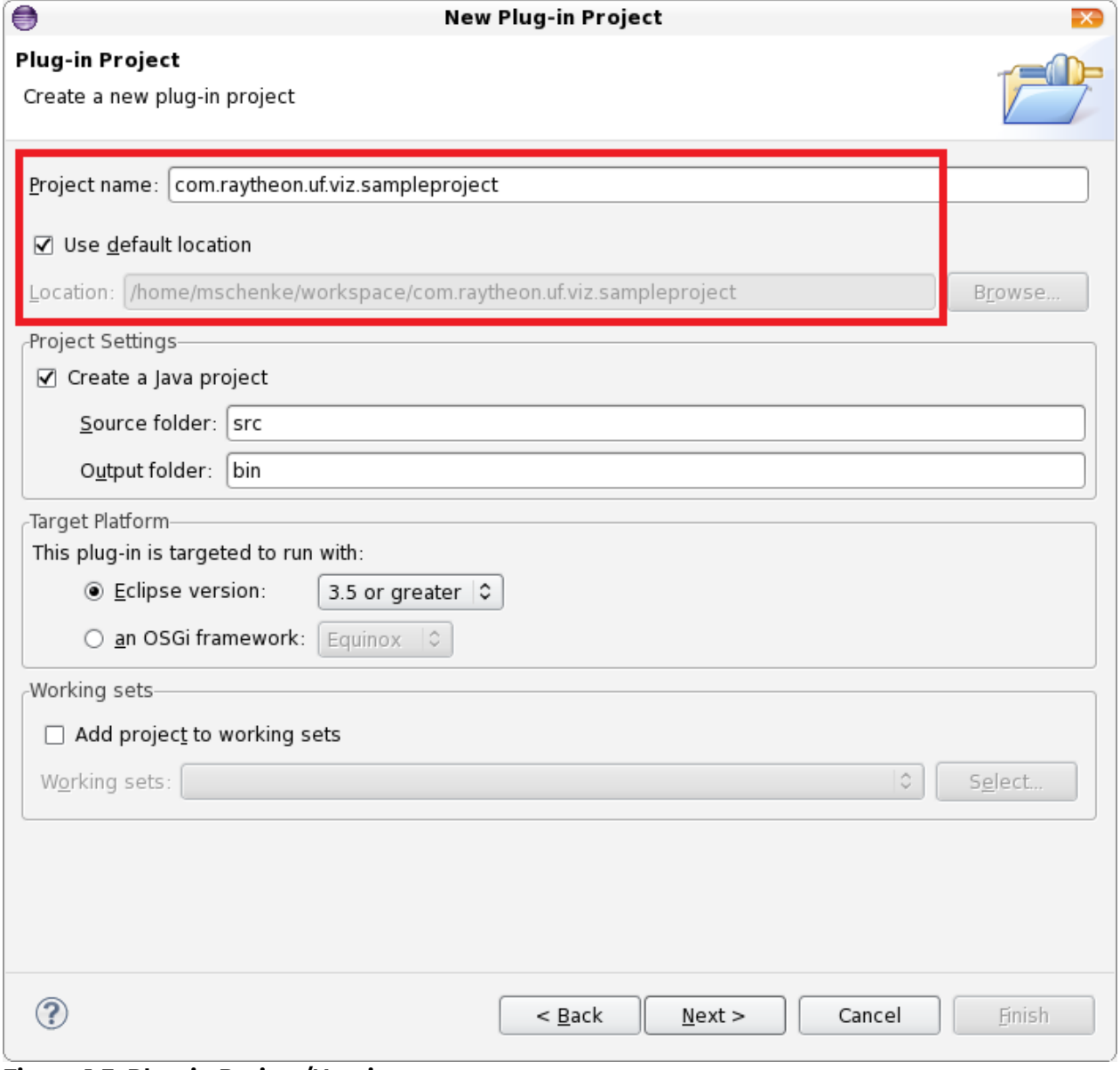

Figure 1-3. Plug-in Project/Naming

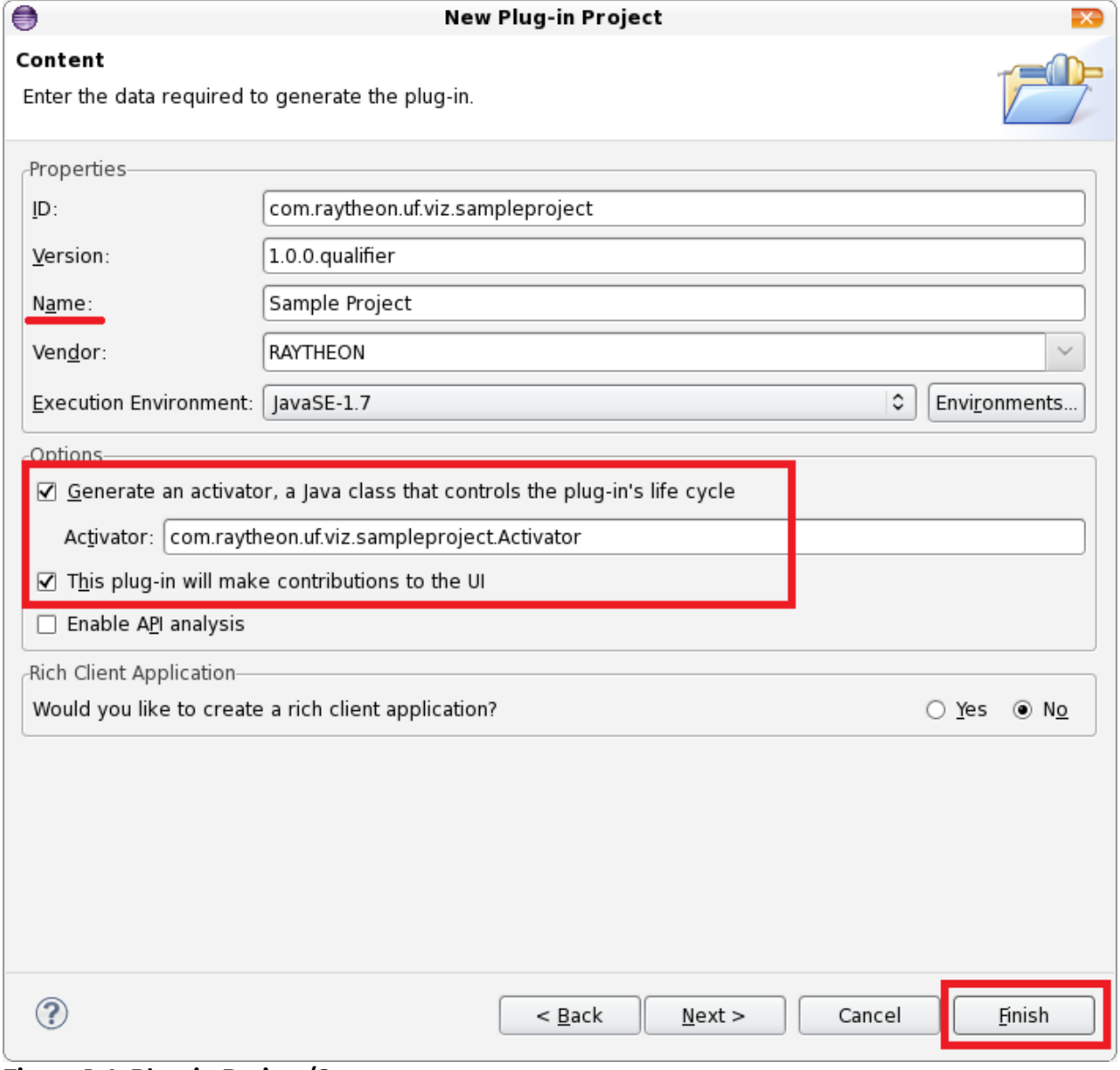

Figure 1-4. Plug-in Project/Content

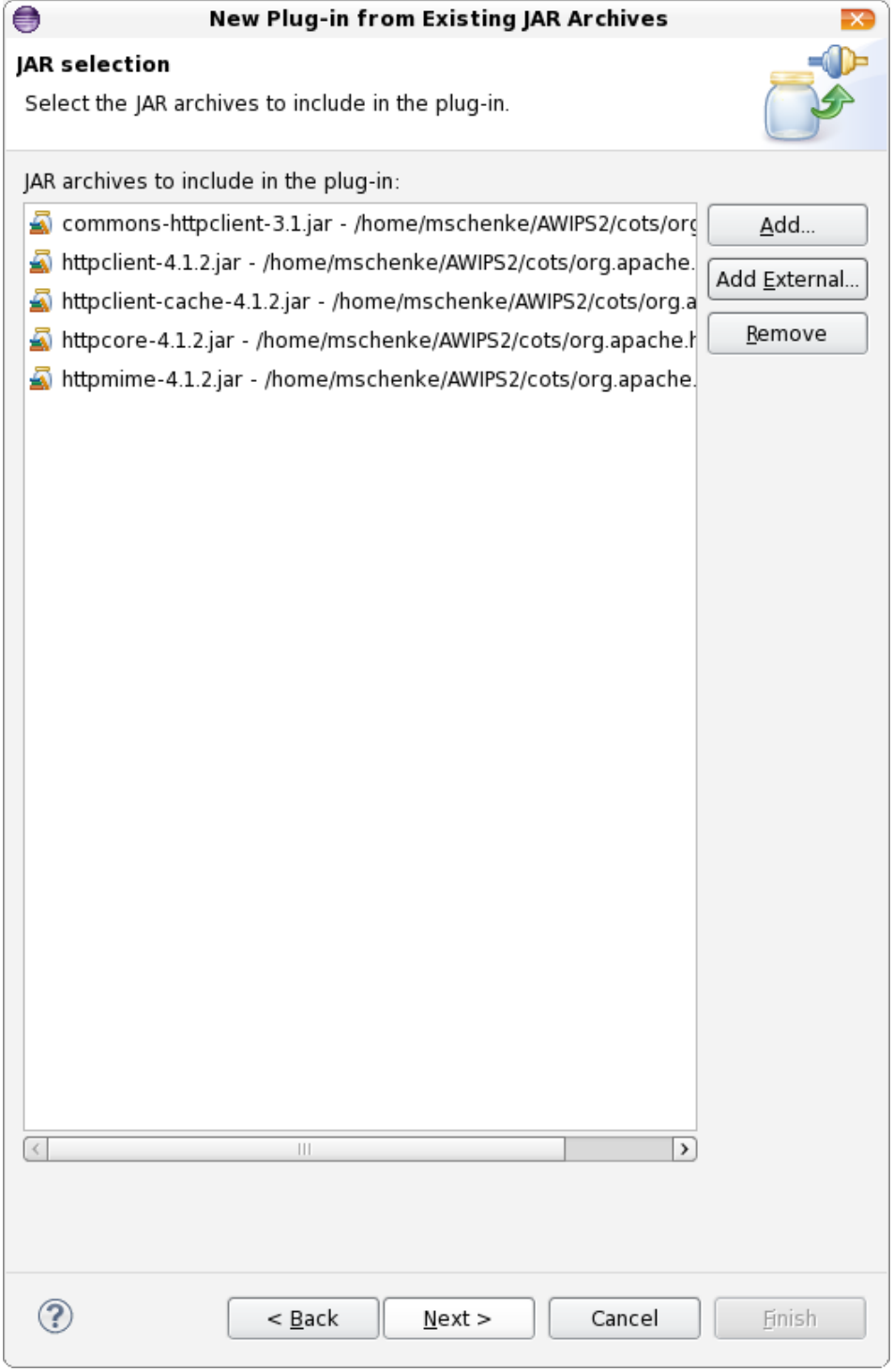

Figure 1-5. JAR Selection

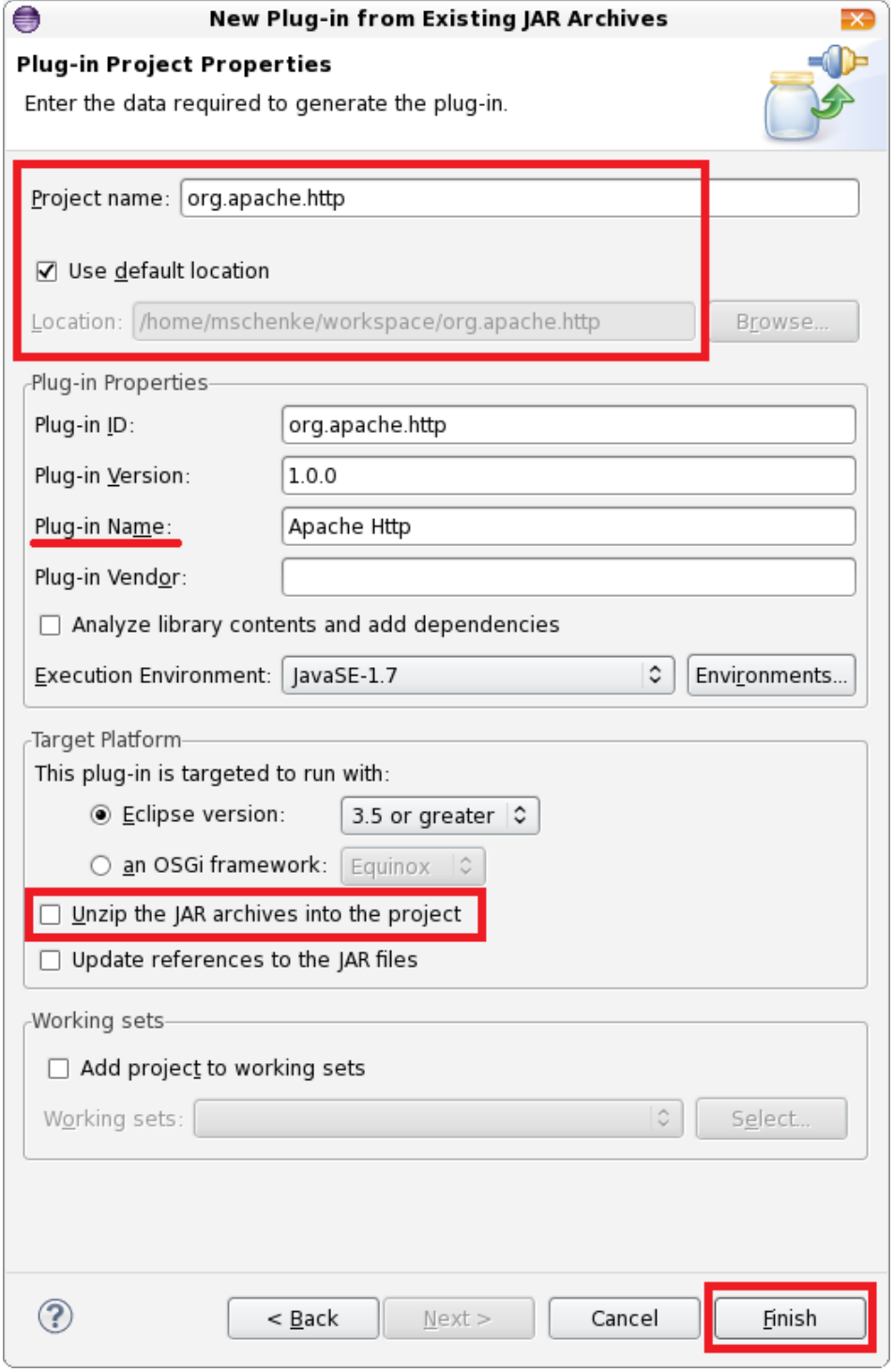

Figure 1-6. Plug-in Project Properties

Plugin Dependency Management

Plugin dependencies are managed through the Eclipse MANIFEST Editor. It can be accessed by opening the project file: META-INF/MANIFEST.MF. This editor controls many plugin configuration settings, one being dependency management. To modify plugin dependencies, the MANIFEST file should be opened and the "Dependencies" tab should be selected as shown in **Figure 1-7.**

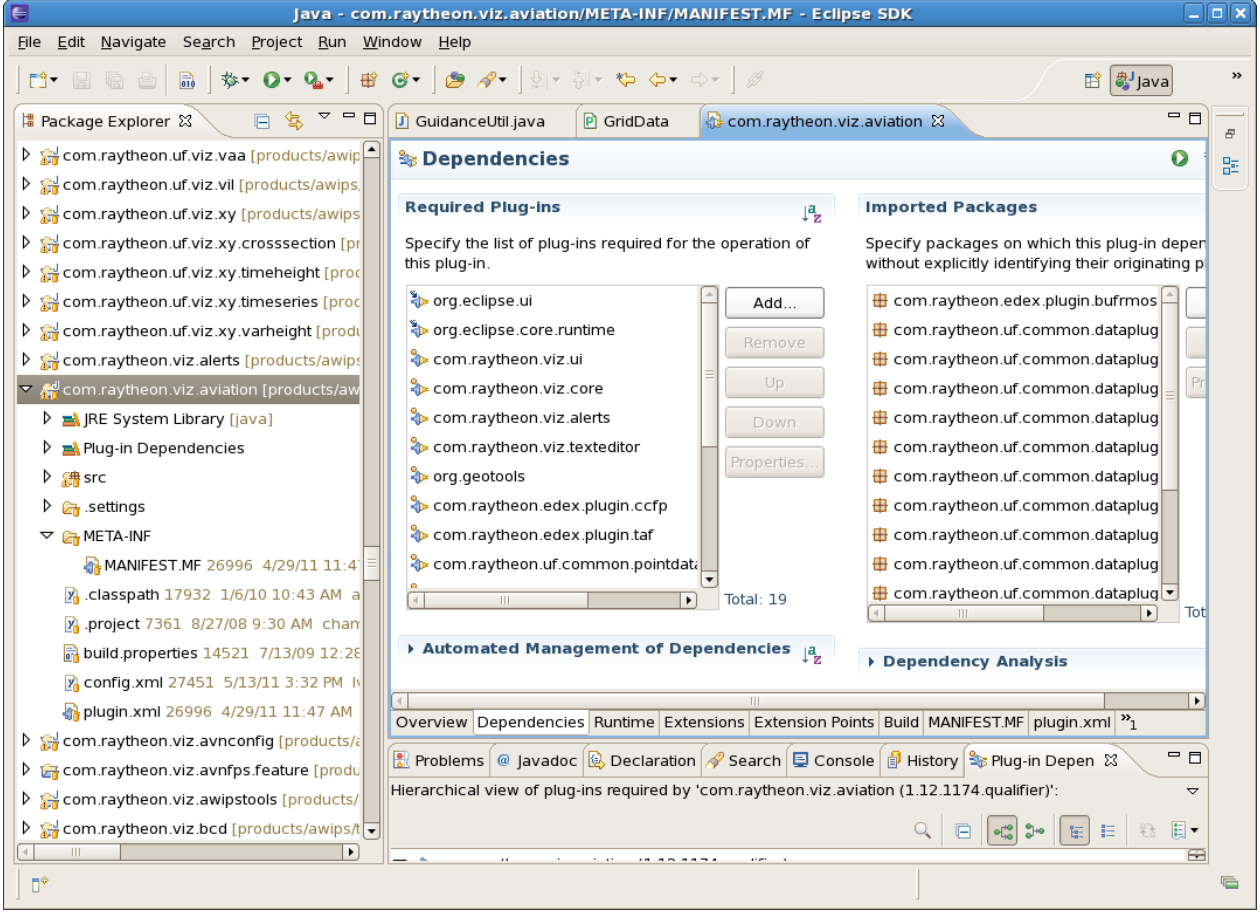

Figure 1-7. Manifest Dependencies Tab

The "Required Plug-ins" column on the left is the manifest's Required-Bundle: statement, and the "Imported Packages" column on the right is the Import-Package: statement, when viewing manifest, the source in the MANIFEST.MF editor tab. With an Imported-Package: there is no control over what plugin the package comes from. This can cause problems when plugin dependencies are automatically determined for builds/installation. **For this reason, it is recommended that only "Required Plug-ins" be used and "Import Packages" be ignored in most cases. Exceptions include: javax.servlet, org.apache.commons.logging, org.apache.log4j where there are multiple possible COTS/FOSS providing implementations of them.** When developing a plugin that will be used by other plugins, it is important to ensure the proper packages are visible to those plugins. This is controlled through the "Runtime" tab in the manifest editor as shown in **Figure 1-8.**

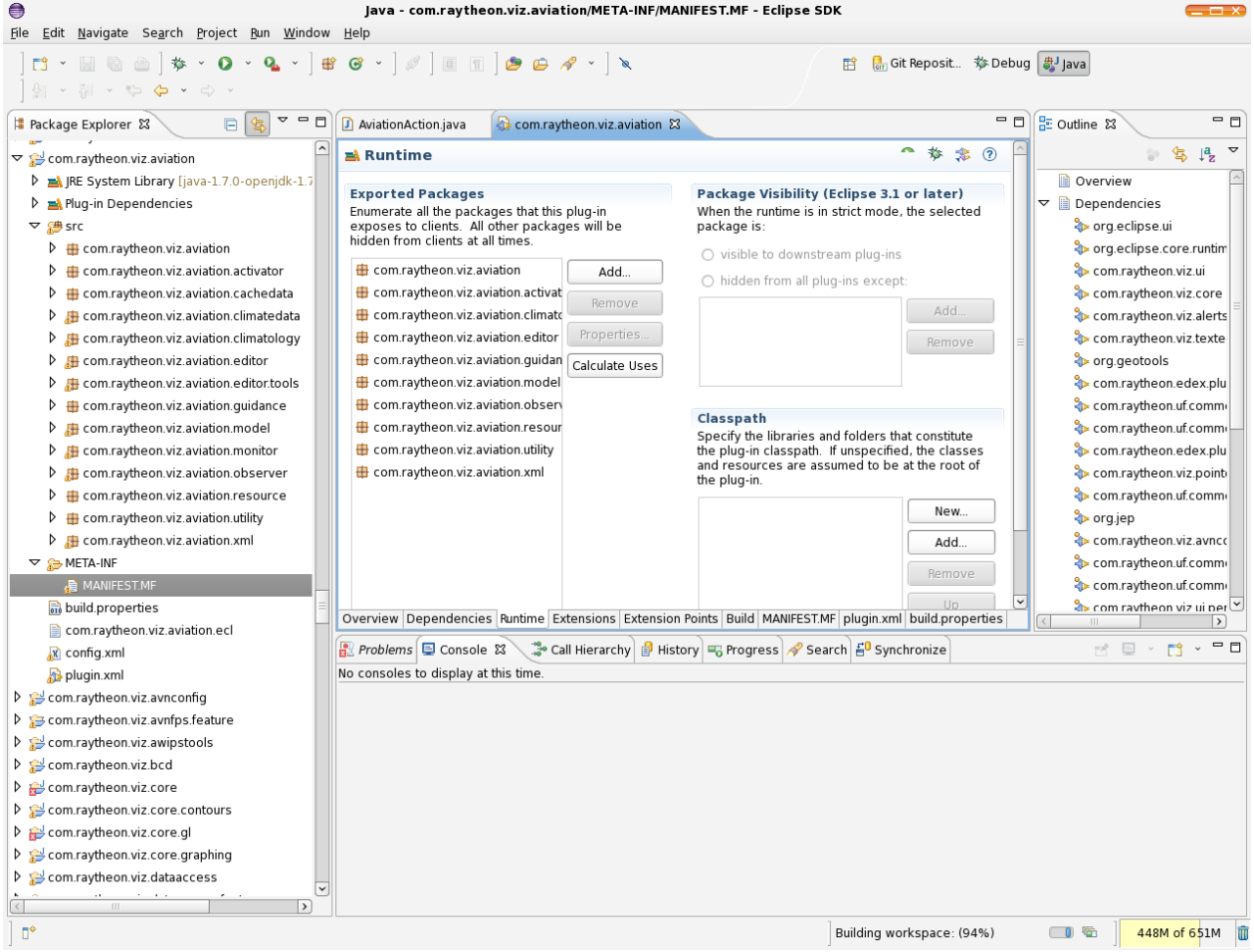

Figure 1-8. Runtime Tab

The "Exported Packages" column on the left is the manifest's Exported-Packages: statement when viewing manifest the source in the MANIFEST.MF editor tab. The packages listed in this section are those that can be imported into another plugin's code when depending on the developed plugin. Not all packages need to be imported; there are certain circumstances where packages may be deliberately hidden from other plugins and only used internally. The "Package Visibility" and "Classpath" sections should not be modified in any way. **Note:** When a COTS/FOSS plugin is created, the "Classpath" section will be prepopulated with the JARs selected during creation.

Helpful Eclipse Shortcuts

The following is a list of useful Eclipse shortcuts. The list is in no particular order, and it is not exhaustive. The list assumes the key bindings have not been changed from the default.

- **Ctrl+L.** Opens a dialog to enter a line number to jump to in the current editor.
- **Ctrl+G.** Places in the Search tab all references in the workspace to the selected element.
- **Ctrl+T.** Displays the type hierarchy of the selected element. If a class is selected, it shows the full type hierarchy; if a class method is selected, it shows the hierarchy of classes that extend/implement that method from the class type hierarchy as shown in **Figure 1-9.**

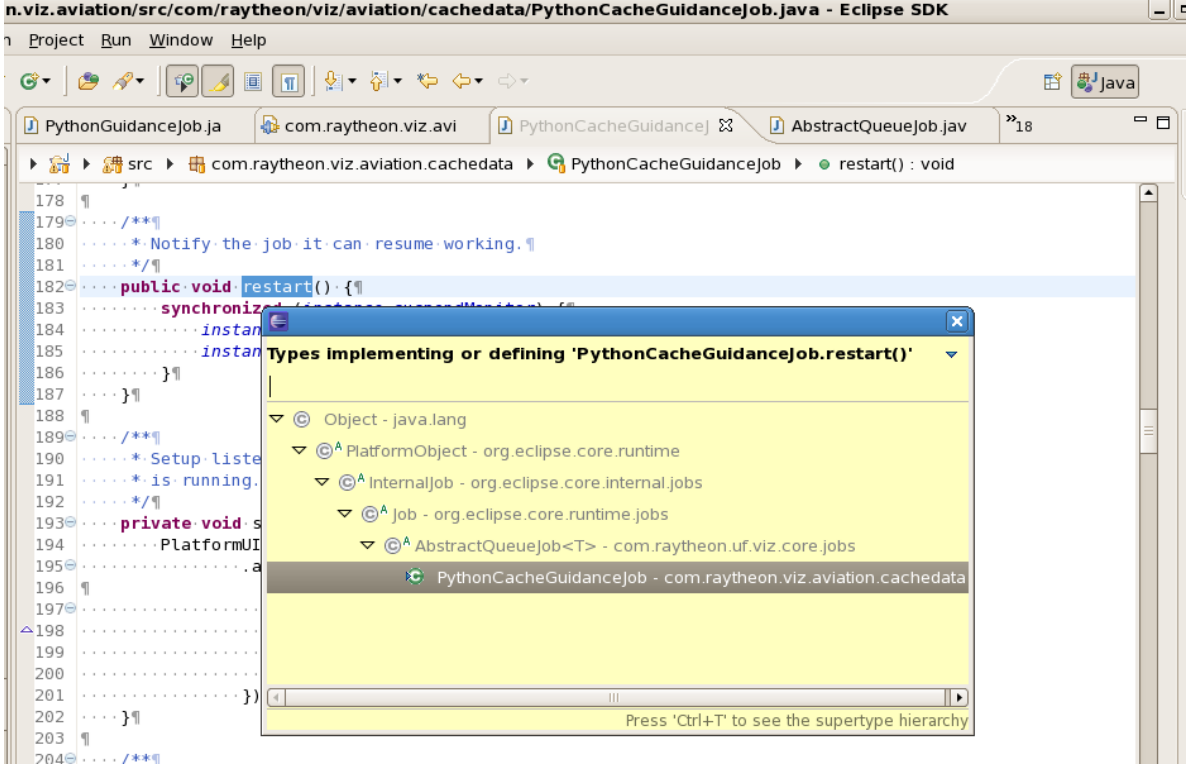

Figure 1-9. Type Hierarchy

- **F3.** In the editor: Opens an edit window displaying the definition of the selected element. In Package Explorer: Opens the selected file in the editor.
- **Shift+Ctrl+L.** Opens a list of the commands and shortcuts for quick execution. Repeating the key command while this list is open brings up a preference window where all commands and key bindings can be edited. **Figure 1-10** displays the Shortcut List.

Activate Editor	F12
Add Artifact to Target Platform	Shift+Ctrl+Alt+A
Add Block Comment	Shift+Ctrl+/
Add Import	Shift+Ctrl+M
Add Javadoc Comment	Shift+Alt+J
All Instances	Shift+Ctrl+N
Backward History	Alt+Left
Build All	Ctrl+B
Change Method Signature	Shift+Alt+C
Close	Ctrl+W
Close All	Shift+Ctrl+W
Collapse	Ctrl+Numpad_Subtract
Collapse All	Shift+Ctrl+Numpad_Divide
Content Assist	Ctrl+Space
Context Information	Shift+Ctrl+Space
Copy	Ctrl+C
Copy Lines	Ctrl+Alt+Down
Correct Indentation	Ctrl+I
Cut	Ctrl+X

Press "Shift+Ctrl+L" to open the preference page.

Figure 1-10. Shortcut List

- **Shift+Ctrl+T.** When editing a Java file, this shortcut pops up a dialog with "Open Type" selected. Pressing the Enter key brings up the "Open Type" dialog. From there, a case-insensitive search using wild cards for classes/interfaces can be performed. If a python editor is active when performing the key binding, the "Pydev: Globals Browser" will be opened; it

performs a similar function on python files. **Figure 1-11** provides an example of finding classes with avn in the name.

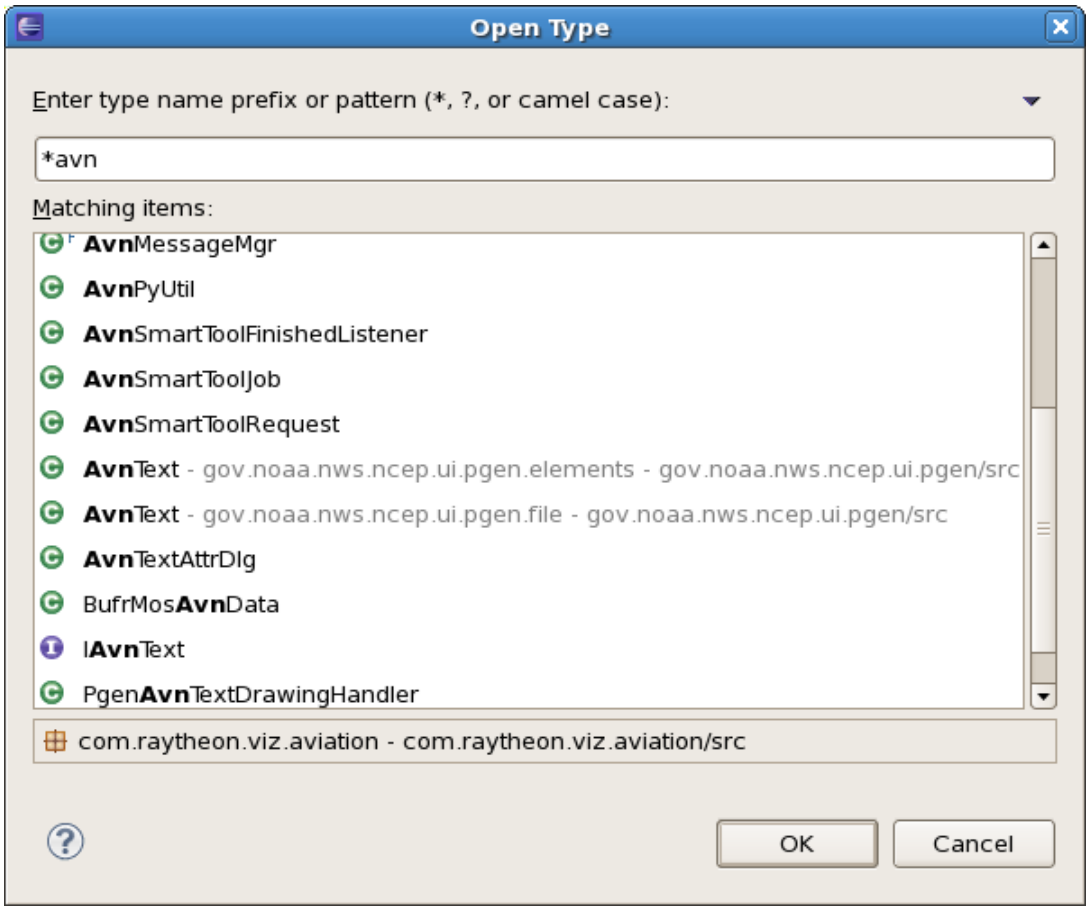

Figure 1-11. Example Open Type Dialog

- **Shift+Ctrl+R.** This shortcut displays the "Open Resource" dialog, which allows for case-insensitive wildcard searches of all files in the workspace. The example in **Figure 1-12** shows files with config in the name that end with xml.

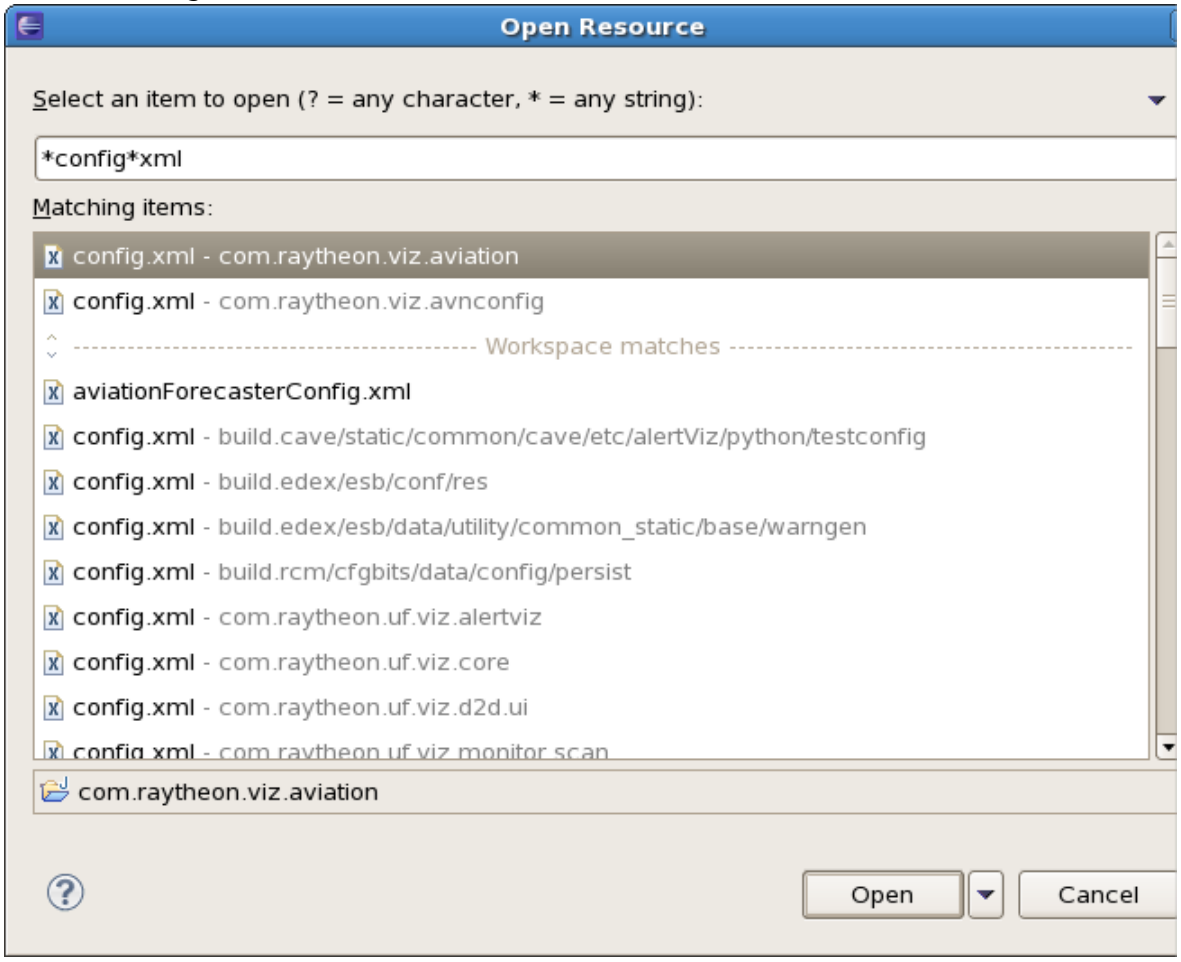

Figure 1-12. Example of Open Resource

- **Ctrl+H.** Pops up the Search dialog with selected text populating the search text fields for the various tabs.
- **Shift+Ctrl+F.** Formats the selected text. (Save Action will perform this for the whole file when code formatting is enabled in the section "Setup of Code Formatters and Save Actions.")
- **Shift+Ctrl+O.** Organizes Imports. (Save Action will perform this if enabled.)
- **Ctrl+S.** Saves the changes made in the file in the active edit. When it contains a Java file, the save action is performed. (See the next section, **Setup of Code Formatters and Save Action**, which follows.)

Setup of Code Formatters and Save Actions

It is important to maintain consistent code formatting/styling for the entire AWIPS II baseline. It aids in keeping the code in compliance with Raytheon's AWIPS II coding standards, and it allows developers across organizations to compare changes made to files easily. For this reason, an eclipse template and code formatting file are provided in the baseline and should be imported for use.

Follow these steps to import the AWIPS II Code Template and Formatter (see **Figures 1-13, 1-14, and 1-15**):

1. In the ADE, select the menu item Window --> Preferences.
2. Select Java/Code Style/Code Templates on the left and select "Import..." on the right.
3. In the import browser, browse to the AWIPS II EDEX plugin directory path (e.g., /home/user/AWIPSII/edexOsgi/). From there, browse to build.edex/opt/eclipse/ and select the codeTemplate.xml file and select "Apply."
4. Select the "Formatter" section on the left and import the formatter.xml file from the same location as codeTemplate.xml and select "Apply."
5. On the left, select Java/Editor/Save Actions.
6. To enable formatting on Save, make sure the following are selected/checked:
    - Perform the selected actions on save
    - Format source code
    - Format all lines
    - Organize Imports
7. Select "Apply" and then "OK."

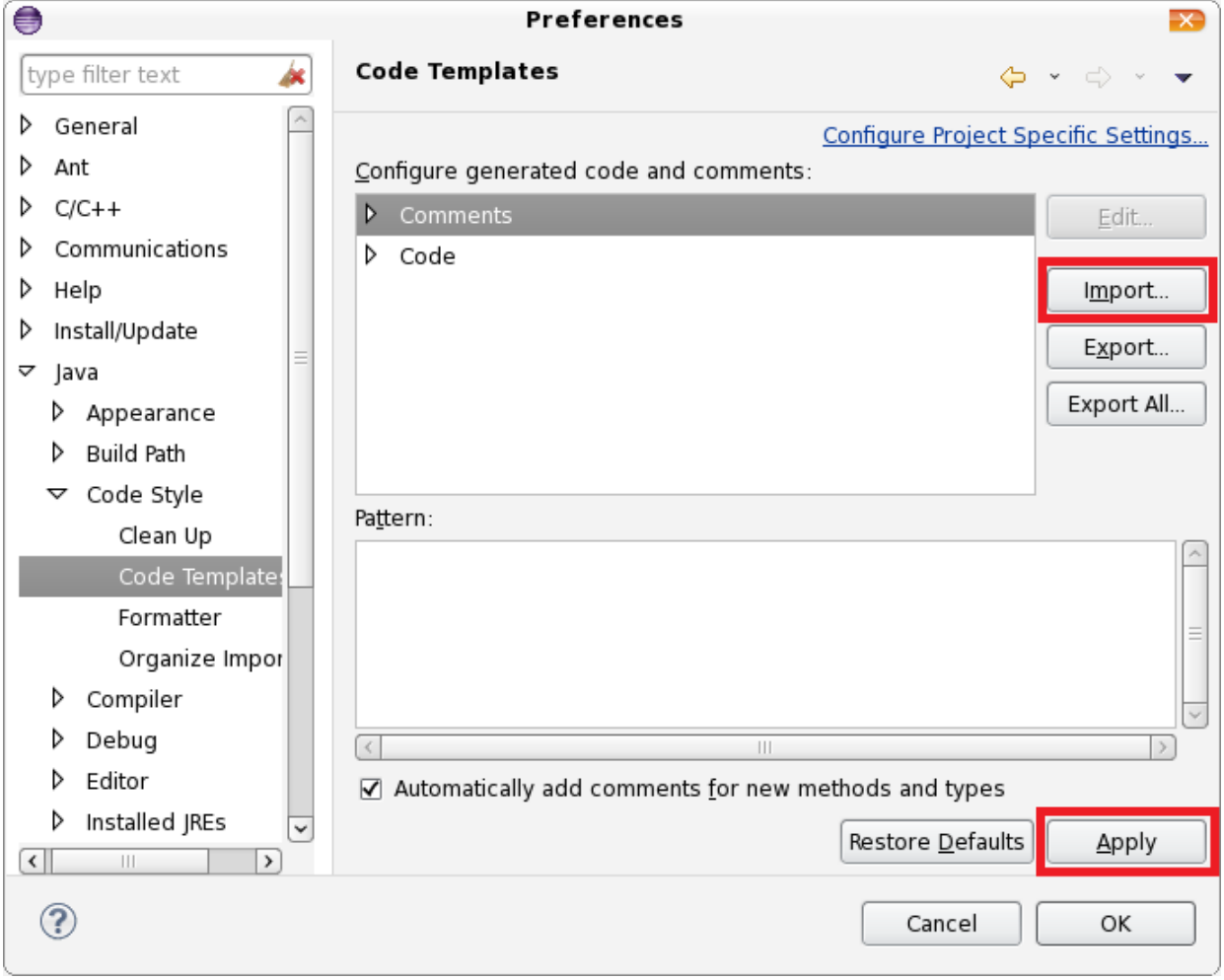

Figure 1-13. Code Templates

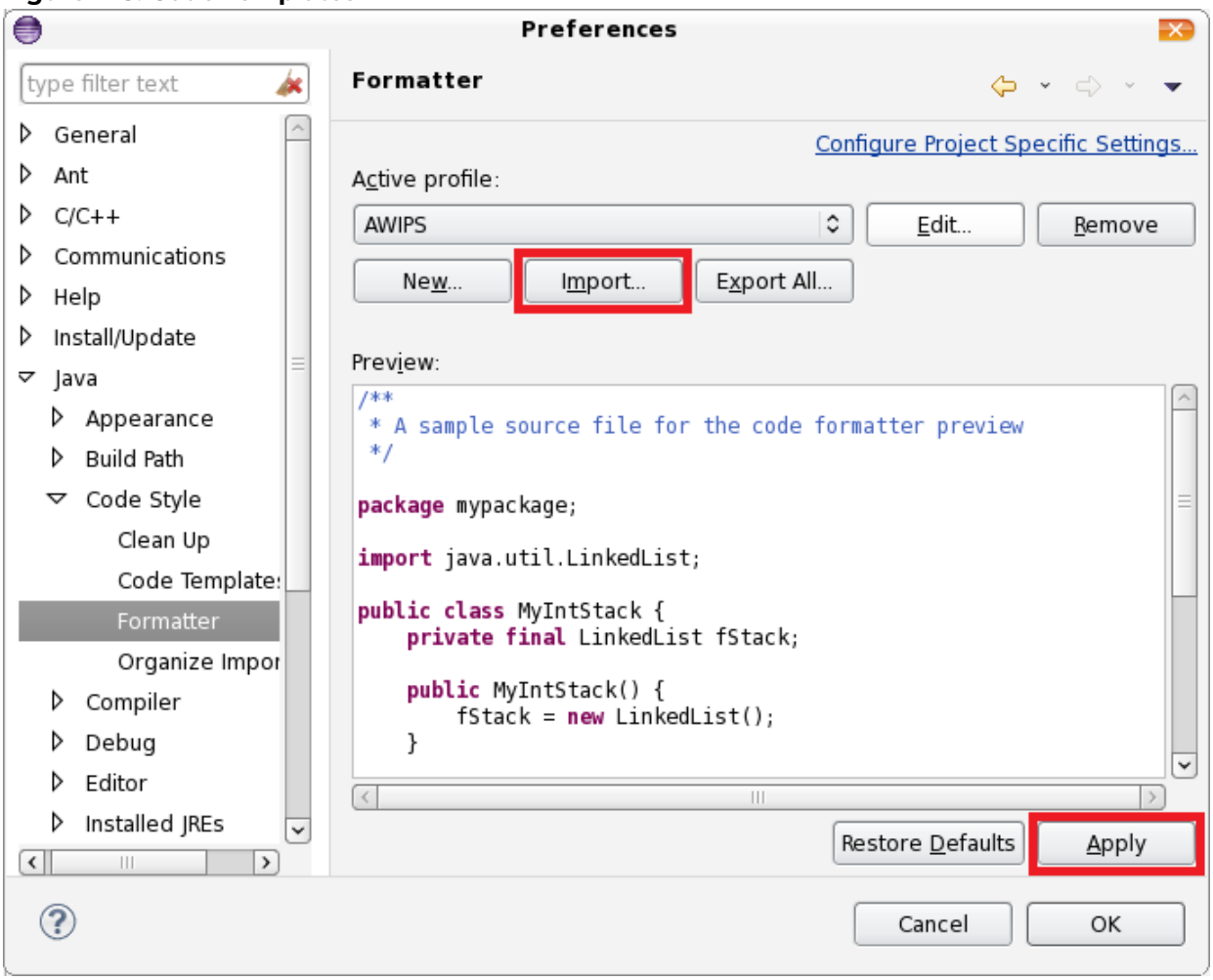

Figure 1-14. Formatter

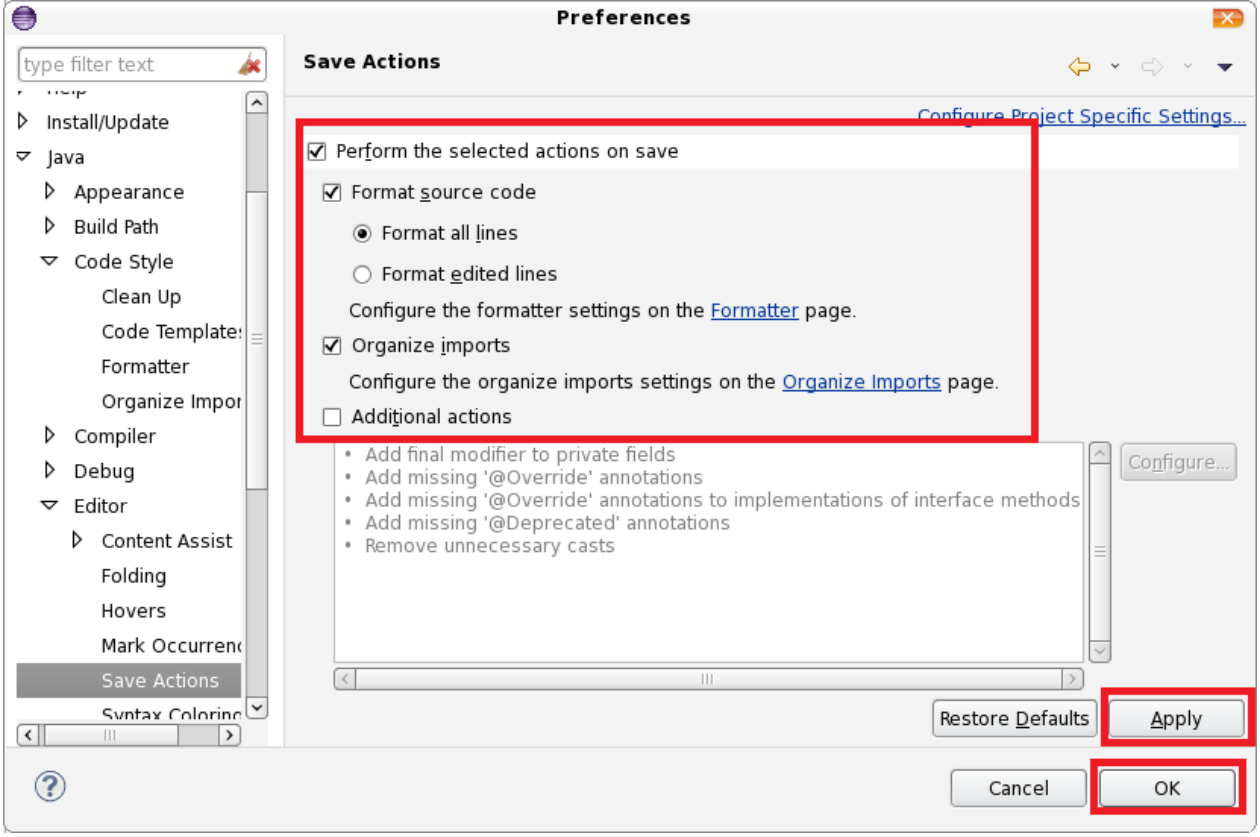

Figure 1-15. Save Actions

RPM Overview

Redhat Package Manager (RPM) is the package manager. It is a program designed to build and manage packages of software including the source and binaries. It is portable and can be run on different platforms.

RPMs (*.rpm) typically include the compiled programs and/or libraries needed for the package, documentation, install, verify, and uninstall scripts, and cryptographic signatures for each file in the package. This makes it easy to verify the integrity of the package. It also includes a list of packages that it depends on, and a list of services that are provided by the package.

RPM maintains a database of all installed packages in /var/lib/rpm/*. Included in the database is a list of files installed by the RPM and which package they belong to. This makes it a very powerful tool for finding out more about each package.

Sources are often provided in source RPMs (*.src.rpm or *.spm). These sources include the pristine developer source code, any patches applied by the package builder, and a SPEC file that is used to tell RPM how to compile the package.

Note: Root privileges are required to install, upgrade, or remove RPM packages. RPM queries can be run as any user.

Most major Linux distributions utilize RPM Package Manager format, including Red Hat (which is the primary AWIPS II Linux distribution), SuSE, and Caldera. Any Linux distribution considered Linux Standards Base (LSB) compliant must supply applications either packaged in the RPM packaging format as defined in the LSB specification, or supply an installer which is LSB conforming (for example, calls LSB commands and utilities).

Common RPM Commands

To interact with RPMs, use the rpm executable (/bin/rpm). The rpm command is standard on most Linux distributions. The rpm executable is used to install, update, and remove packages as well as to execute queries for information about packages.

- To install an RPM, use: `rpm -ivh ${RPM}`
- To update / upgrade an RPM, use: `rpm -Uvh ${RPM}`
- To remove an RPM, use: `rpm -e ${RPM}`
- To execute an RPM query, use: `rpm -q …` (there are multiple query types that can be executed utilizing the `-q` argument).

Building RPMs

A specs file is used to create an RPM. A specs file essentially consists of a header with basic information about the RPM, one or multiple scriptlets that are run during various phases of the build / install, and a list of files included in the RPM.

The commonly used tags and descriptions of each tag are provided in **Table 1-1**. Note that some tags are required. These are designated in the table.

Table 1-1. Tags Commonly Used in Building RPMs

Tag	Required Tag	Description
Name:	✓	The name of the package.
Summary:	✓	A basic summary of what the package is and its purpose.
Version:	✓	The package version. The version is in the format X.Y.Z where it is the major release, Y is the minor release, and Z is the revision.
Release:	✓	The package release. The package release is generally an integer.
Group:	✓	Group is used to specify the package type.

BuildRoot:	✓	A temporary directory that will be used to assemble the package. The buildroot makes it possible to assemble the package without compromising / altering your root file system.
URL:	✓	A link to a website about the package or etc.
License:	✓	The license associated with the RPM package.
Distribution:	✓	
Vendor:	✓	The company and/or group that created the package.
Packager:	✓	The package author.
provides:		The packages and/or services the RPM provides. Every individual package or service requires a separate "provides" tag.
requires:		A list of the packages and/or services that are required by the RPM. Every individual package or service requires a separate "requires" tag.
%description	✓	A description of the RPM package.
%prep		This scriptlet is executed during the RPM build. This scriptlet contains instructions for the first phase of the build. The first phase is generally used to gather and unpack source and other dependencies that are required to build the RPM.
%build		This scriptlet is executed during the RPM build. This scriptlet contains instructions for the second phase of the build. The second phase is generally used to build the source code (running configure and/or make, etc.).
%install		This scriptlet is also executed during the RPM build. This scriptlet contains instructions for the third and final phase of the build.
%pre		This scriptlet is executed during installation. This scriptlet is run before the package files are placed on the filesystem.
%post		This scriptlet is also executed during installation. This scriptlet is run after the package files are placed on the filesystem.
%preun		This scriptlet is executed during uninstallation. This scriptlet is run before the package files are removed from the filesystem.
%postun		This scriptlet is also executed during uninstallation. This scriptlet is run after the package files are removed from the filesystem.
%files		A list of the files that are included in the package. File attributes including owner, group, and file permissions can also be specified as part of this tag.

Once a specs file has been created, the rpmbuild application (/usr/bin/rpmbuild) can be used to actually build an RPM. The rpmbuild application is not installed by default on every Linux distribution and must be installed before it can be used.

To build an RPM using rpmbuild: `rpmbuild -ba ${SPECS}`

If the rpmbuild is successful, the RPM that was built can be found within one of the architecture-specific directories {generally one of: [i386, noarch, x86_64]} in: /usr/src/redhat/RPMS.

Using YUM

When the RPM executable is used to install one or multiple rpms, it is the responsibility of the user to ensure that the RPMs are installed in the correct order when installing multiple RPMs, as well as accounting for all dependencies. This is not a difficult task when there are just a few RPMs; however, if there are dozens of RPMs (and there are close to 100 AWIPS II RPMs) installing all of the RPMs can become a time-consuming task that requires multiple commands. One solution to managing multiple package installations is Yellowdog Updater Modified (YUM).

YUM is an open-source command-line package-management utility for RPM-compatible Linux operating systems and has been release under the GNUs Not Unix (GNU) General Public License.

YUM is capable of installing one or multiple RPMs from a YUM repository or directly from the filesystem. Unlike RPM, YUM is capable of determining dependencies between RPMs and will install the RPMs in the correct order based on the dependencies. So, instead of using multiple rpm commands to install two or more RPMs, a single YUM command can be used.

The AWIPS II RPMs

There are more than 100 AWIPS II RPMs including 32-bit (i386), any architecture (noarch), and 64-bit (x86_64) RPMs. The AWIPS II RPMs have been divided into four "classes": core RPMs, EDEX RPMs, CAVE RPMs, and python extension (site-package) RPMs.

The core RPMs are RPMs that every other type of RPM is dependent on in some way. The following AWIPS II RPMs are included in the set of core RPMs: awips2-java, awips2-python, awips2-postgresql, awips2-database, and several others.

The EDEX RPMs include the edex-base RPM (consists of the EDEX directory structure as well as the edex configuration and scripts) and the edex component RPMs. The EDEX component RPMs divide the EDEX plugins into functional subsets: there is an EDEX component RPM that contains core plugins and another edex component RPM that contains radar plugins. The component RPMs make it possible to apply a patch or an enhancement to a single portion of edex to avoid the need for a complete reinstall.

The CAVE RPMs include the CAVE RPM (consisting of the CAVE RCP executable, cave scripts, and the cave directory structure) and CAVE p2 repository RPMs. The CAVE p2 repository RPMs extend the cave rcp executable and contribute functionality. Similar to the EDEX component RPMs, the CAVE p2 repository RPMs make it possible to apply a patch or an enhancement to a single functional portion of cave to avoid the need for a complete reinstall.

The python extension RPMs extend the functionality and capability of python when installed. Examples of the python site-package RPMs include awips2-python-numpy, awips2-python-ufpy, and awips2-python-nose.

Persistence, Hibernate, Postgres, and CoreDao

Postgres Database (v11.14)

EDEX uses Postgres as the repository for storing metadata extracted during the data ingest process. The database also contains the text database, the maps database, and the hydro database.

In a typical AWIPS II installation, the Postgres database, by default, is installed on the dv1 server under the **/awips2/postgresql** directory. For typical development purposes, the contents of this directory are not significant as most development can be carried out without concern for the underlying database configuration. Nevertheless, a few items in this directory may be of some importance. The following is an overview of some important directories and files in the postgres installation.

- **/awips2/postgresql/bin**
  - **start_developer_postgres.sh** and **start_postgres.sh.** Either of these scripts can be used to start the postgres server on a development workstation.
  - **Psql.** This is the command line interface for interacting with the postgres database. A typical usage of this command to connect to the metadata database is as follows: psql -d metadata -U awips -h dv1. You would then enter the password for user awips which is awips. Once connected, you are now able to interact with the database using SQL or the set of meta-commands provided by psql. Detailed documentation about using psql and psql's meta-commands can be found here: https://www.postgresql.org/docs/11/app-psql.html (https://www.postgresql.org/docs/11/app-psql.html)
  - Further documentation about the client applications in this directory can be found here: https://www.postgresql.org/docs/11/reference-client.html (https://www.postgresql.org/docs/11/reference-client.html).
- **/awips2/postgresql/doc.** This directory contains a complete set of html documents detailing the usage of postgres.
- **/awips2/postgresql/include.** This directory contains code used internally by postgres and should not be manually modified
- **/awips2/postgresql/lib.** This directory contains libraries used by postgres and should not be manually modified
- **/awips2/postgresql/man.** Unix man pages for client applications included in the postgres installation.

The /awips2/data directory is used by postgres to store table information (schemas, tablespaces, etc.) and user configurable files. Some important directories and files contained in this directory are:

- **/awips2/data/postgresql.conf.** This file controls a number of items defining how Postgres behaves behind the scenes including memory usage, logging, and querying. Modifying items in this file can have significant performance implications. Therefore, modifications should be carefully considered. A series of documentation explaining the various configuration items contained in this file is here: http://www.postgresql.org/docs/8.3/static/runtime-config-file-locations.html (http://www.postgresql.org/docs/8.3/static/runtime-config-file-locations.html)
- **/awips2/data/pg_hba.conf.** This file controls client connection permissions and authentication. Detailed documentation about this file and other client authentication concerns can be found here: http://www.postgresql.org/docs/8.3/static/client-authentication.html (http://www.postgresql.org/docs/8.3/static/client-authentication.html)
- **/awips2/data/pg_log.** This directory contains the postgres logs. Logging behavior can be enabled/disabled and modified in the aforementioned postgresql.conf file.
- The complete set of documentation for Postgres 11 is located here: https://www.postgresql.org/docs/11/index.html (https://www.postgresql.org/docs/11/index.html).

PGAdmin4

PGAdmin4 is a graphical interface to view postgres databases. Refer to the PGAdmin4 documentation for usage details: http://www.pgadmin.org/docs/ (http://www.pgadmin.org/docs/)

Hibernate

At a very high level, Hibernate is a COTS product that can be used to map Java classes to a relational database. This effectively removes the user/developer from being concerned with constructing complex and sometimes confusing SQL commands to interact with persistent data and focus more on the behavior and interactions of Java objects. AWIPS II currently uses Hibernate v5.4. Detailed documentation about Hibernate can be found here: http://docs.jboss.org/hibernate/core/3.5/reference/en-US/html/ (http://docs.jboss.org/hibernate/core/3.5/reference/en-US/html/)

Configuring Hibernate

Hibernate is injected into EDEX via the /awips2/edex/conf/spring/edex.xml file. In this file, SessionFactory objects are defined for each database currently in use. The Hibernate SessionFactory object is the critical link between a persistent Java class and the database. The SessionFactory gets database connections, controls transactions, and generates the SQL statements from the provided Java objects.

The SessionFactory uses a configuration file to configure how it is going to connect and interact with the database. These configuration files are located at /awips2/edex/conf/db/hibernateConfig. Each database uses its own SessionFactory and therefore each database has its own hibernation configuration file. For example, the configuration file for the metadata database is located here: **/awips2/edex/conf/db/hibernateConfig/metadata/hibernate.cfg.xml**. Detailed information about the options available for use in this document can be found here: http://docs.jboss.org/hibernate/core/3.5/reference/en/html/session-configuration.html (http://docs.jboss.org/hibernate/core/3.5/reference/en/html/session-configuration.html)

Database connection pooling is also used. A package called c3p0 is used for this purpose. The c3p0 connection pooling parameters are defined in the hibernate.cfg.xml files mentioned above. The parameters available are described here: http://www.mchange.com/projects/c3p0/index.html (http://www.mchange.com/projects/c3p0/index.html). See Appendix C in that document for information about using c3p0 with Hibernate.

The SessionFactory must be made aware of which classes are to be mapped to the database. These classes may be specified in the SessionFactory Spring bean definition. Since EDEX was designed to be extensible via data type plugins, this approach was insufficient. As a result, EDEX dynamically determines the set of mapped classes at startup. The **com.raytheon.uf.common.serialization.SerializableManager** class scans the class path for classes with the javax.persistence.Entity (@Entity) annotation. Any classes found with that annotation are included in the SessionFactory.

Adding Hibernate Annotations

Hibernate provides two methods for mapping classes, configuration files and annotations. AWIPS II uses annotations for mapping. For informational purposes, information about using configuration files for mapping classes can be found here: http://docs.jboss.org/hibernate/core/3.5/reference/en/html/mapping.html (http://docs.jboss.org/hibernate/core/3.5/reference/en/html/mapping.html)

Detailed information about mapping classes using annotations is located here: http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/ (http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/).

A few highlights of the most important annotations follow.

- **@Entity.** The @Entity annotation must be attached at the class level. The @Entity annotation is used to make the class Hibernate aware.

- **@Table.** The @Table annotation must be attached at the class level. The @Table annotation is used to specify the table name, catalog, schema, and any unique constraints. All attributes on the @Table annotation are optional.
- **@Id.** The @Id annotation attached to the property in the Java class that is to be used as the primary key for the associated database table. All classes mapped with the @Entity annotation must have an @Id parameter specified (except in the case of a complex key or an embedded object)
- **@Embeddable.** The @Embeddable annotation is placed at the class level. The @Embeddable annotation informs Hibernate that this class is not mapped to its own table. Instead, the properties contained in this class will be included in the containing class' database table
- **@Embedded.** The @Embedded annotation is placed at the property level. The @Embedded annotation means that the columns specified by an @Embeddable class are to be included in this class's database table
- **@Column.** The @Column annotation is placed at the property level (or on the getter for that property). The @Column annotation is used to give Hibernate hints about how to create the associated column in the database. Hibernate is able to dynamically figure out the associated column type based on the Java type in most cases (i.e., String maps to varchar, int maps to integer). *The @Column annotation is implied (using Hibernate derived defaults) for all properties in an @Entity annotated class.* In other words, if you do not explicitly annotate a property with the @Column annotation, Hibernate will map the property for you anyway.
- **@Transient.** The @Transient annotation is placed at the property level. The @Transient annotation tells Hibernate not to include this property in the database mapping.
- **@Index.** The @Index annotation is placed at the property level. The @Index annotation tells Hibernate to create a database index based on the this property.
- **@Type.** The @Type annotation is placed at the property level. This annotation is used in the case that you have specified a property type that cannot be automatically determined by Hibernate. The @Type annotation expects you to give it the FQN of an implementation of org.hibernate.usertype.UserType. A UserType implementation details how Hibernate should transform the class into a form that can be inserted into and retrieved from the database.
    - An example of this is the utilityFlags property in **com.raytheon.uf.common.time.DataTime.** The type is defined as **com.raytheon.edex.db.mapping.DataTimeFlagType** which is an implementation of **org.hibernate.usertype.UserType**.

Data Access Objects

Data type plugins may specify their own data access objects for data access. The **com.raytheon.uf.edex.database.dao.CoreDao** is used as the base class from which all other data access objects inherit. The CoreDao constructor takes a **com.raytheon.uf.edex.database.dao.DaoConfig** object, which specifies which session factory to use (essentially tells the dao which database to look at). Once a CoreDao object has been instantiated, it may be used to insert, delete, update, and query data from the database.

There are several methods available for inserting data into the database. These are create, persist, persistAll, saveOrUpdate, and mergeAll. The create method saves an object that has not been previously inserted into the database. The persist, persistAll, and saveOrUpdate methods save (or update if previously saved) objects into the database.

Users have the option of using several different methods for querying data:

- **executeSQLQuery and executeNativeSQL.** Users may submit SQL strings to directly query the underlying database structure
- **executeHQLQuery and executeHQLStatement.** Hibernate provides its own query language for querying objects. The Hibernate Query Language (HQL) language is similar to Structured Query Language (SQL). Documentation on HQL can be found here: http://docs.jboss.org/hibernate/core/3.5/reference/en/html/queryhql.html (http://docs.jboss.org/hibernate/core/3.5/reference/en/html/queryhql.html). The executeHQLQuery is used for querying the database and the executeHQLStatement method is used for all other non-query (i.e. insert, update, etc.) statements.

- **queryByCriteria.** There are several queryByCriteria methods available on CoreDao. These methods expect a **com.raytheon.uf.edex.database.query.DatabaseQuery** object to be submitted.
    - The DatabaseQuery object allows users to easily specify which parameters to query for. The constructor of the DatabaseQuery expects the developer to specify which class they are querying for. Developers may use the addQueryParam methods to specify the name, value and operator used to query on. The addReturned field specifies which fields are to be returned from the query. *Note that when adding query parameters and returned fields, the names used are those contained in the Java class and not the database column names.* It is worth noting that attempting to do queries using table joins using the addJoinField on the DatabaseQuery class will not yield correct results.
    - An example use of queryByCriteria:

```
  LambertConformalGridCoverage coverage = (LambertConformalGridCoverage) grid;

DatabaseQuery query = new DatabaseQuery(this.daoClass);

query.addQueryParam("dx", coverage.getDx());

List<LambertConformalGridCoverage> result = (List<LambertConformalGridCoverage>) qu
eryByCriteria(query);
```

If the basic methods in CoreDao are insufficient for the needs of a data type plugin, a developer may extend CoreDao and implement additional methods. Otherwise, CoreDao may be instantiated and used out of the box.

Ignite

What is ignite

Ignite is an in memory cache. It is used in AWIPS to store large data objects quickly and reliably. The previous solution was PyPies which had disk and network bottlenecks. Ignite solves these problems by storing data in memory to avoid disk and it is clustered which avoids the network bottleneck. PyPies is still used for persistent storage with ignite acting as a cache between PyPies and the other services.

High level Architecture

Ignite will be inserted as a caching layer between PyPies and other processes that access PyPies (primarily CAVE and EDEX). When new data comes into the system it will be stored into Ignite and then be immediately available to be read out of Ignite. In the background Ignite will store the data back into PyPies. Ignite is holding all the data in memory, so when memory is needed then data that is already stored in PyPies will be dropped from the Ignite cache. If this data is requested later then it will be read from PyPies and stored back in the cache. When the system is restarted Ignite will start out empty and accumulate data from new store operations and reading data back from PyPies as needed.

Ignite will be operating as a cluster on the new hardware. Each physical server will be running a VM dedicated to running an ignite instance. Each piece of data will be stored on two different ignite instances which ensures that the overall system can continue operation even in the event of a single hardware failure. WFOs run with 3 ignite server nodes, and National Centers run with 6 ignite server nodes. Ignite consistently failed with obscure error messages at National Centers with a 6-node cluster, so only 3-node clusters are used in both cases. Thus, WFOs have a single ignite cluster consisting of the cache1-3 VMs, whereas National Centers have 2 clusters, one consisting of cache1-3, and another consisting of cache4-6.

To maximize compatibility there is a servlet within EDEX that is able to emulate a PyPies server and handle any requests in the same way that PyPies currently does. This is necessary to support things like the CAVE Thin Client which uses an HTTP proxy to communicate between CAVE and PyPies making it impossible to communicate using the native ignite protocols. This compatibility layer will be used for all communications with any application that currently accesses PyPies except EDEX. To maximize performance, EDEX will load the ignite library directly and operate as an ignite node running in client mode. This will allow EDEX to connect to the ignite cluster using ignite internal protocols. CAVE will be able to load data from ignite by hitting any of the EDEX servlets with PyPies style requests.

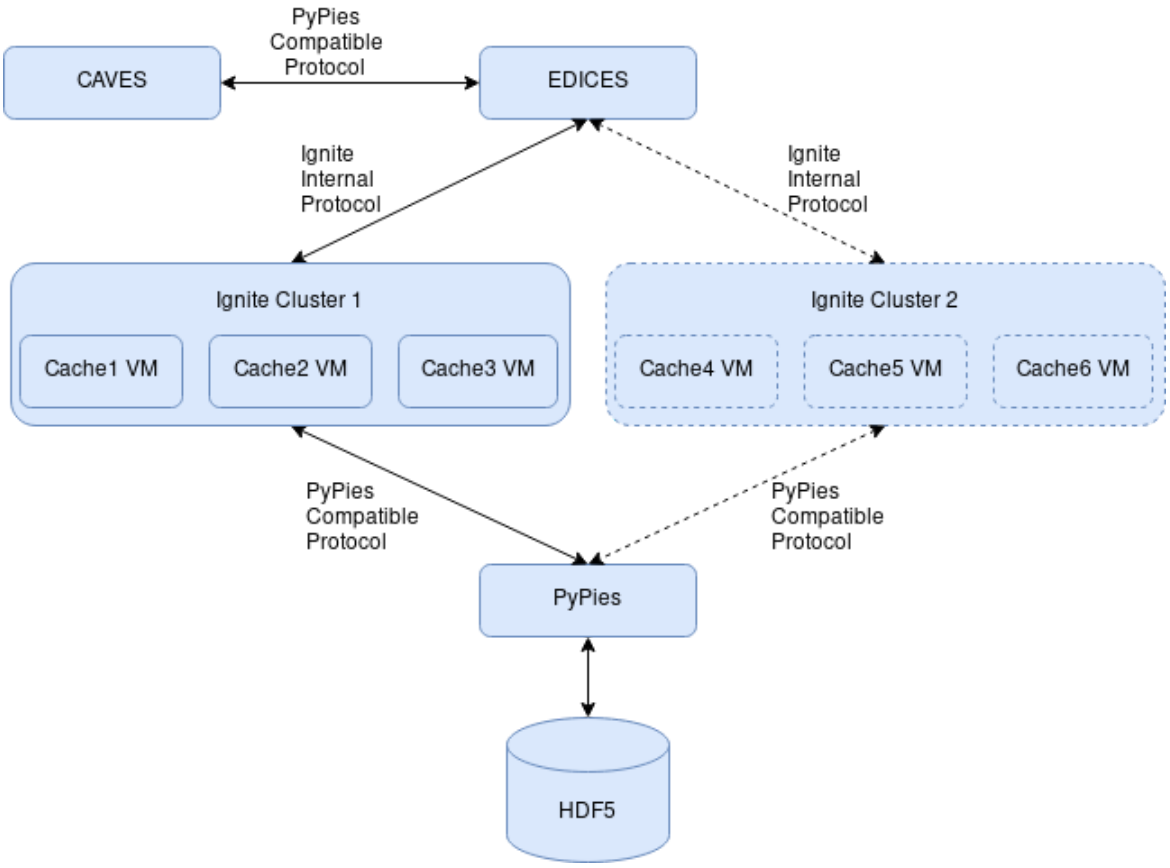

Figure 3.6: Ignite High-Level Architecture

Ignite cluster, node, and cache structure

Each instance of ignite that's running is called a node. There are server nodes and client nodes. In our case, the server nodes are the standalone ignite instances that are running on the cache VMs. These are the nodes where data is actually cached, and that access PyPies behind the scenes to interact with the persisted data. The client nodes are instances of ignite that are ran within EDEX, which essentially provide ways to interact with the server nodes to retrieve or modify data.

A group of ignite nodes that know about each other is called a cluster. In our case, we have either 1 or 2 clusters, depending on if the system is a WFO or a National Center. Each EDEX instance that interacts with ignite will then have 1 or 2 ignite clients, 1 for each cluster.

Within a single cluster, there can be multiple caches for different types of data, not to be confused with the cacheN VMs. Currently we have separate caches for grid, satellite, radar, and point data, along with a default cache for everything else. Each cache is spread across all server nodes. Ignite lets you specify an affinity function which is used to determine which server node a particular piece of data goes to. In our case, the affinity function is based on the hdf5 file path that the data goes in, so all data for a single file goes to the same node.

In a standard single cluster system, all of these caches are naturally in the one cluster. In a dual cluster configuration, the grid cache is mapped to the second cluster, while all others go in the first cluster.

The below diagrams outlines this structure for a single EDEX JVM with a 2 cluster system.

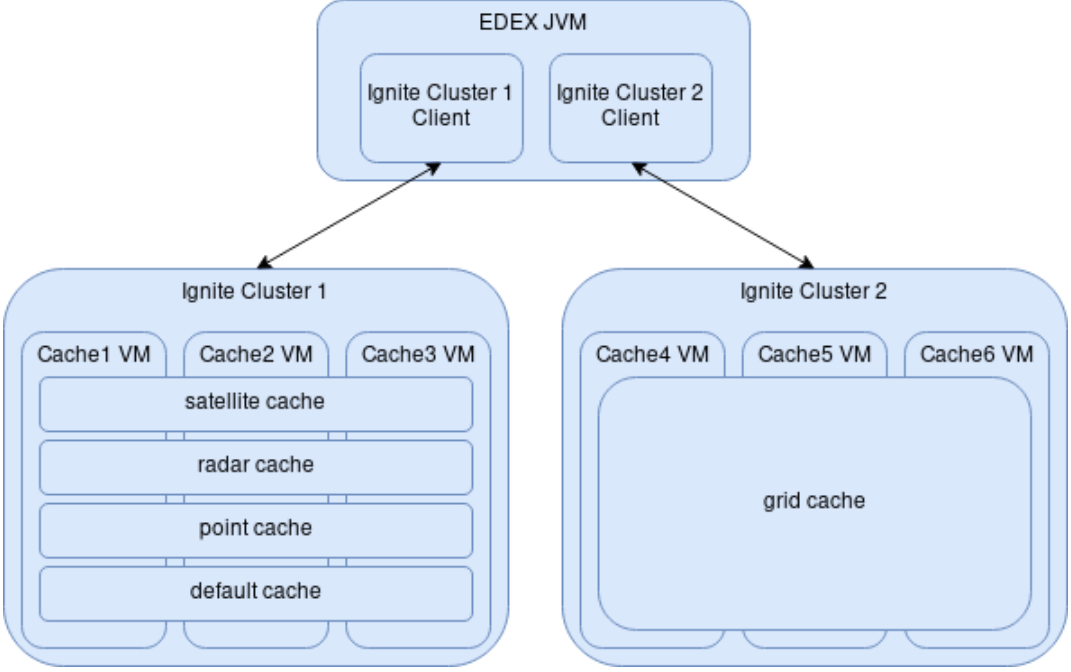

Figure 3.7: Ignite Cluster and Cache Structure

Running ignite service

Ignite is installed as a systemd service. The service calls into the a2_ignite.sh script which contains a variety of configurable parameters.

The service and the script take a single argument that specifies a group of configuration options. The two base modes are:

- production - Run using more memory and with a cache backup
- developer - Run using less memory and without a cache backup

There is one additional option that can be provided:

- debug - Allow socket connections from a java debugger, this will allow a debugger to connect on port 5102

For example to start/stop ignite in production mode would look like this:

```
  systemctl start ignite@production
systemctl stop ignite@production
```

For a developer who wants to debug the ignite service, the start/stop commands would instead look like this:

```
  systemctl start ignite@developer-debug
systemctl stop ignite@developer-debug
```

Configuring ignite

Generally for configuration changes to take effect, all ignite nodes (clients and servers) must all be shutdown together, and then all started back up. If each node is restarted individually, then the nodes that stay up while one node is restarted will remember the old configuration and tell the restarted node to continue using it.

Note that some of the setup.env variables mentioned below are auto-configured in component.spec files (awips2-ignite component.spec for servers, and awips2-edex-configuration for clients).

Server nodes

If you need to change some of the settings in ignite for a particular instance you can override the systemd unit configuration. For example to change the settings to increase the max data region size on a developer ignite, you could run this command:

```
sudo systemctl edit ignite@developer
```

Set the contents to something like this:

```
[Service]
Environment=IGNITE_DATA_REGION_MAX_SIZE_GB=4
```

And then run the service normally.

To see what settings can be changed from the environment look at the start script in /awips2/ignite/bin/a2_ignite.sh.

Additional environment settings are also configurable in /awips2/ignite/bin/setup.env. There are variables for configuring the cluster that this particular server is a part of. There is also a LOCAL_ADDRESS variable to tell other ignite nodes how to communicate with this ignite server. This is necessary to prevent an ignite node from identifying itself as a value that doesn't uniquely identify it, such as ev-cache. Finally, there are values for setting up qpid and jms, which should match the configuration values in EDEX's setup.env.

Most environmental settings are used in spring xml located at /awips2/ignite/config/awips2-config.xml. This spring also has additional settings that have been chosen for the awips use case. The ignite documentation (https://ignite.apache.org/docs/latest/) has extensive examples and descriptions of the various options for configuring ignite in spring.

Client nodes

Since EDEX is running as an ignite client node, there are various ignite specific settings within EDEX. First in /awips2/edex/bin/setup.env there is the DATASTORE_PROVIDER which can be set to ignite or pypies. setup.env also contains variables for configuring the ignite clusters to run with. There are also variables for configuring the PyPies compatibility server location. Finally, there is a LOCAL_ADDRESS variable like in ignite's setup.env, to ensure that each ignite node tells the other nodes to communicate with it via a valid address.

Next there is /awips2/edex/conf/spring/edex-datastore.xml which contains the spring configuration for ignite. Since edex is in client mode it does not need as much configurations for the ignite process itself but the many of the configuration options are the same as those used in awips2-config.xml. The configuration of each data-specific cache is in this file as well, along with registering each cache with cluster 1 or 2. Also note that there are 2 ignite configuration beans in this file, but an actual ignite instance is only created for the second cluster configuration if IGNITE_CLUSTER_2_SERVERS is non-empty.

Additionally within some plugin spring files there is an optional element to configure a specific cache for that plugin. If this is not present a plugin will use the default cache. If too many caches are used then starting and stopping ignite and edex will be slower and there will be more internode communication required to manage the caches so we try to keep the number of dedicated caches small. However for the plugins with the most data there are some performance problems if they share the same cache and compete for resources so it is necessary to allow some plugins to store to their own cache. An example fo the plugin specific cache configuration is below:

```
  <bean factory-bean="ignitePluginRegistry" factory-method="registerPluginCacheName">
    <constructor-arg value="grid" />
    <constructor-arg value="gridDataStore" />
</bean>
```

Finally, there is some configuration within EDEX modes files. For EDEX modes that use ignite, they specify communication and discovery ports for each cluster, such as in request.sh. For EDEX modes that don't use ignite, they disable it by setting DATASTORE_PROVIDER to pypies, such as in centralRegistry.sh.

Low level Design

Overview

Most code interacts with ignite through the IDataStoreFactory interface. This allows code to be written that can work with either PyPies or ignite or another future IDataStoreFactory. This also allows shared code to access ignite directly on EDEX while using PyPies in CAVE.

For ignite, the IgniteDataStoreFactory essentially maps an HDF5 file name to a cache and cluster combination, and then creates an appropriate IgniteDataStore instance for that cache and cluster. For example, an obs data file is mapped to the point cache and cluster 1. The IgniteDataStore instance itself is essentially just a wrapper around an IgniteCache that translates the IDataStore calls into cache operations. The key used in the cache is a DataStoreKey, which is a combination of the file path and the group name. The value stored in cache is a DataStoreValue, which just wraps an array of IDataRecord.

Most code does not access ignite APIs directly, as its access is instead centralized in helper classes. These helper classes wrap the ignite APIs to provide consistent timeout, exception, and retry handling for all ignite operations. The main helper classes are AbstractIgniteManager and its client/server implementations for top-level ignite access, along with IgniteCacheAccessor for cache-level access. Each of these contain do*Op methods that provide convenient access to Ignite/IgniteCache instances while wrapping the operation in exception handling.

PyPies compatibility servlet

The PyPies compatibility is implemented as a servlet that is independent of Ignite and can serve PyPies from any IDataStore implementation. The PyPiesCompatibiltyService runs within EDEX and instantiates a PyPiesServlet with the same IgniteDataStoreFactory used elsewhere in EDEX, which interacts with the actual ignite clusters in the same way. When CAVEs or other services that don't know about ignite ask EDEX for the PyPies server, EDEX lies to them in GetServersHandler so that the requests are rerouted to this servlet, which then reroutes them to ignite.

Ways to interact with servers from clients

There are three main ways to interact with servers/caches from ignite clients: key/value operations, entry processors, and computes.

Key/value operations let you interact with a cache as you would with any map implementation. These are done via IgniteCache methods such as put, get, and getAndPutIfAbsent. Cache entry locking is automatically handled.

EntryProcessors are used via the various IgniteCache.invoke methods. These allow the cache operations to be encapsulated in a single object that is sent to the ignite server nodes to perform an operation which is often faster than trying to perform cache operations locally. When doing this, the processor is then executed on the server node that corresponds to the given key. The processor automatically loads the data value for that key, either from the cache or by reading it through from PyPies, and allows you to then work with the cache value or modify it. It also automatically locks the cache entry for the duration of the processor. However, the fact that it automatically loads the data value may not be preferable performance-wise for operations that don't always need it.

The IgniteCompute interface is accessed via Ignite.compute. This allows more free-form access to the server nodes, as you can provide any Runnable/Callable to execute, and no automatic data loading or locking is done.

Examples of each of these options are available in AWIPS' usage of ignite.

Handling point data

Point data is particularly problematic for ignite because it continually appends to the same data record, usually starting over every hour. Because of this ignite must constantly be loading the entire data record, appending, and then storing back. Throughout this repeated operation will slow down. This is a case where it is critical to use an EntryProcessor to avoid sending all the data across the network for all operations. Also all pointdata plugins are automatically placed in a dedicated cache to avoid impacting other plugins when operations slow down late in the hour.

Determining where data goes in ignite

In short, data plugins are mapped to a cache for a general data type, which is mapped to a cluster. For example, the obs and pirep plugins are mapped to the point data cache, which is mapped to the first cluster. There is also a default data cache that any unregistered plugins use. Note that the cache-to-cluster mapping is only relevant at National Centers where 2 clusters are used. Currently the grid cache is mapped to the second cluster and all other caches use the first cluster.

The plugin-to-cache mapping is handled by CachePluginRegistry, while the cache-to-cluster mapping is handled by IgniteClusterManager.

Each cache within a cluster is spread across all server nodes in that cluster. An affinity key is used to determine which node a particular data piece goes to. In our case, the affinity key is the hdf5 file path that the data will be stored to, so all data for a single hdf5 file will go to the same server node. The affinity key is determined by the affinity annotation on DataStoreKey.path.

Exception handling

There are two main ways that exceptions are handled/prevented in our usage of ignite. The first is via the centralized ignite API access mentioned earlier, which provides consistent timeout, exception, and retry handling for all ignite operations.

Exceptions are also handled by configuring all ignite nodes to restart on critical failures. On the server side, this is handled at the end of a2_ignite.sh by code copied from the ignite-provided ignite.sh, along with setting the failure handler to ignite's built-in RestartProcessFailureHandler in awips2-config.xml. On the client side, ignite's built-in RestartProcessFailureHandler doesn't work since the client node isn't started up from the command-line like the server nodes. Instead, a custom IgniteClientFailureHandler is configured to restart the ignite client when necessary.

Write behind and data batching

Ignite allows for each cache to use either write through or write behind behavior. Write through mean that each data storage operation updates the cache and writes the data through to PyPies as a single atomic operation. Write behind means that each data storage operation simply updates the cache and returns, and ignite then asynchronously writes the data through to PyPies.

Each cache can be configured differently for this, starting with whether to use write through or write behind. When using write behind, there are extra options such as how many write behind threads to use, how often write behind should occur, and the maximum size of each write behind batch. Generally write behind is preferred for performance, although it introduces some issues discussed in the following section. The configuration of these values is in edex-datastore.xml.

Whether using write through or write behind, the DataStoreCacheStore class is used to tell ignite how to interact with the persistent data store (PyPies).

The internal ignite class that is primarily responsible for handling write behind is GridCacheWriteBehindStore. Within GridCacheWriteBehindStore, a configurable number of flusher threads wake up at the configured interval to check if there's data to flush. If so, those threads all race to grab what they can from the pending write cache, then apply the entries they grabbed by telling DataStoreCacheStore to write them through. If one thread gets behind, the other threads will still go ahead and start working on the next batch at the next interval. When a value is successfully written through, it is then removed from the pending write behind cache.

The write behind logic also has flush size and critical size threshold values. Each time an entry is added to the write behind cache, the cache size is checked. If it exceeds the flush size, then all the flusher threads are immediately woken up to write behind entries. If the critical size is exceeded, then a single entry is immediately written through on the current thread. Thus, it temporarily switches to write through to prevent the write behind cache from getting too large. This helps slow down ingest when necessary to prevent running out of memory.

Data storage auditing

Before ignite, the database and data store were automatically kept in sync by the order of operations. A data storage operation would attempt to store the raw data directly to PyPies first. If this succeeded, it would then proceed to store the metadata to the database. If the PyPies write failed, an exception would be thrown that would prevent the metadata from being written, which keeps things in sync. Now, when a data storage operation writes to ignite, it simply puts the raw data in its cache and returns that it succeeded, which allows the metadata to be written to the database. When ignite later attempts to write the raw data through to PyPies, this can fail and cause things to be out of sync.

To resolve this, a centralized data storage auditer runs in a single EDEX JVM. Each data storage operation has a unique trace ID, and the key parts of the storage operation are sent to the auditer, which compares the success or failure of the different pieces and deletes the metadata or raw data if necessary to keep things in sync.

Because point data has to track the index that it was stored at in a particular hdf5 file, this auditing is not sufficient to correct the index values on an hdf5 storage failure. As a result, point data uses write through instead of write behind.

The main relevant code areas for this are IDataStorageAuditer and its implementations, DefaultDataStorageAuditListener, DataStorageAuditEvent, and PersistableDataObject.traceId.

Ignite internal caches and locking

For a single ignite cache (e.g. the grid data cache), there are two main caches in the code, the main cache and the write behind cache. When a particular data value is looked up, the main cache is first checked. If it's not found there, then the pending write behind cache is checked. If it's not found there, then it is read through from PyPies.

The main cache and the write behind cache each have their own separate locks, although both locks are on a per-key basis. So two threads cannot work with the same key in the main cache at the same time, but one thread can be working with a key in the main cache while another thread works with the same key in the write behind cache. There are no shared references between the two caches, so this is safe. For write-through ignite caches, the main cache lock is held the whole time while the key is worked with in the main cache (e.g. via an entry processor) and while the data is written through to PyPies.

The main cache locking is done using the GridCacheMapEntry.(un)lockEntry methods, while the write behind cache works with StatefulValue objects, which extend a lock implementation themselves.

Overview of plugins

- com.raytheon.uf.common.datastore.ignite - Contains the main implementation of IDataStore that maps operations to ignite caches including all the entry processors and write behind logic.
- com.raytheon.uf.common.datastore.pypies.servlet - a standalone Servlet that can emulate pypies.
- com.raytheon.uf.common.datastore.ignite.pypies - ties the servlet into an ignite service.
- com.raytheon.uf.edex.pointdata.ignite - To automatically associate point data plugins with the point data cache.
- com.raytheon.uf.ignore.core - contains server-specific files for ignite (note that this is currently installed via an EDEX RPM though, as the ignite component.spec hasn't been setup to install

jars)

Ignite logs

You can view logs in /awips2/ignite/logs or use journalctl like this:

```
 journalctl -u ignite@developer
journalctl -f -u ignite@developer
```

One particularly useful log message that can be monitored is the metrics logging which is produced every minute and gives a good overall view of the state of ignite, here is an example:

```
INFO 2020-01-23 17:16:28,698 4675 [grid-timeout-worker-<span class="hljs-comment">#71] IgniteKer
 Metrics <span class="hljs-keyword">for</span> <span class="hljs-built_in">local</span> node (to
 ^-- Node [id=ec30a4c4, uptime=00:03:00.044]
 ^-- H/N/C [hosts=1, nodes=1, CPUs=32]
 ^-- CPU [cur=0.03%, avg=0.07%, GC=0%]
 ^-- PageMemory [pages=203]
 ^-- Heap [used=137MB, free=86.62%, comm=512MB]
 ^-- Off-heap [used=3MB, free=99.86%, comm=1104MB]
 ^-- sysMemPlc region [used=3MB, free=96.82%, comm=40MB]
 ^-- default region [used=0MB, free=100%, comm=1024MB]
 ^-- TxLog region [used=0MB, free=100%, comm=40MB]
 ^-- Outbound messages queue [size=0]
 ^-- Public thread pool [active=0, idle=0, qSize=0]
 ^-- System thread pool [active=0, idle=6, qSize=0]
```

Within EDEX all the ignite related logging is sent to a dedicated log file at /awips2/edex/logs/edex-<mode>-ignite-<date>.log.

The server-side logging is configured in ignite-logback.xml, while the EDEX/client-side logging is in the main logback-edex-*.xml files.

Disabling ignite

EDEX can be configured to ignore ignite and instead interact with PyPies directly. This is intended to save resources for developers who are running things locally and don't need ignite. This is done by setting the following variable in a local environment file, such as ~/.bashrc

```
 export DATASTORE_PROVIDER=pypies
```

/awips2/edex/bin/setup.env will then pick up that value and use it, instead of defaulting that variable to "ignite".

The Python Virtual Environment

AWIPS II historically has had its own Python installation completely separate from the system Python. As of AWIPS II release 21.4.1, it will use the Red Hat-provided system Python 3 installation along with a virtual environment.

Virtual environments (short: virtualenv or venv) are a standard feature provided by Python 3. A virtual environment is a directory whose structure is very similar to that of a standard Python installation, but instead of real binaries, it contains symbolic links to the appropriate Python binaries elsewhere in the system. It also makes use of the system's Python standard library, but maintains its own site-packages directory.

The virtual environment provides these benefits:

1. It is not necessary to build and install a complete Python installation anymore. We just symlink to the Red Hat-provided Python.
2. The separate site-packages directory allows us to maintain complete control over which Python packages are available to AWIPS (as was the case before the virtual environment).
3. The virtual environment is independent of Python patch-version upgrades. For example, the system Python can be upgraded from 3.6.8 to 3.6.9 without any changes whatsoever to the virtual environment and without a new release of the awips2-python RPM package.
4. A virtual environment can optionally make use of Python packages that are installed at the system level. The Python virtual environment used by AWIPS currently does not do this; it has no access to the system site-packages directory. But the availability of this feature allows for the future possibility of replacing Python FOSS packages that we package and distribute ourselves, with Red Hat-provided equivalents that would be installed to the system Python.

How AWIPS uses the Python virtual environment

The awips2-python RPM package contains an empty virtual environment, with no packages installed in it other than pip and setuptools (which, as essential packaging tools, are required by Red Hat's Python 3 and are included by default in all virtual environments). This virtual environment is installed to /awips2/python.

For invocations of the Python interpreter to make use of the virtual environment, the venv must first be activated. This is done by sourcing the file /awips2/python/bin/activate (or activate.csh if the csh shell is being used). This sets environment variables and updates the $PATH to make sure invocations of 'python3' (without a path) resolve to /awips2/python/bin/python3.

The activate script is sourced on login by all users except root, via the file /etc/profile.d/awips2Python.sh (or awips2Python.csh). Because of this, AWIPS software continues to function as it did when /awips2/python contained a full Python installation.

One can at any time confirm that the virtual environment is active by checking the VIRTUAL_ENV environment variable:

```
  $ echo $VIRTUAL_ENV
/awips2/python
```

Making your Python code work with the virtual environment

There are no special considerations required to make AWIPS Python code work inside the virtual environment. Just ensure that, as before, any Python scripts intended to be run directly have their first line as “#!/awips2/python/bin/python3”.

How Python FOSS packages are installed to the virtual environment

Currently, almost all Python FOSS packages provided by AWIPS are built from source, using the setup.py script included in the upstream source distribution of each package. These are the usual contents of the %install section of the RPM spec file (this example is taken from the awips2-python-cftime package):

```
  pushd . > /dev/null
cd %{_python_build_loc}/cftime-%{version}
/awips2/python/bin/python setup.py install \
   --root=%{_build_root} \
   --prefix=/awips2/python  | exit 1
popd > /dev/null

# Merge lib64 into lib to avoid problems with installing into the virtualenv
if [ -d "%{_build_root}/awips2/python/lib64/" ]; then
    rsync -a %{_build_root}/awips2/python/lib64/ %{_build_root}/awips2/python/lib || exit 1
    rm -rf %{_build_root}/awips2/python/lib64
fi
```

Note the bit of boilerplate code at the end which merges the lib64 directory into lib. This is necessary because some Python packages create a lib64 directory containing native binaries. In the virtual environment, lib64 is a symlink to ./lib, and we want lib to be the canonical location for everything that is installed. That said, **the last five lines in the above snippet are required at the end of the %install section in all AWIPS Python packages.**

Python wheels

Now that pip, the Python package manager, is available in the Red Hat Python distribution as well as in all virtual environments, **installation from a wheel (prebuilt Python package with .whl extension) is the preferred method for all AWIPS Python RPM builds going forward.**

Here is an example of the %install section in an AWIPS Python FOSS RPM spec file:

```
  v="%{_installed_python_short_no_dot}"
pkg_file="tables-%{version}-cp${v}-cp${v}m-manylinux1_%{_build_arch}.whl"
pkg_path="%{_baseline_workspace}/foss/tables-%{version}/packaged/${pkg_file}"

/awips2/python/bin/pip3 install \
   --ignore-installed \
   --no-deps \
   --no-index \
   --root=%{_build_root} \
   --prefix=/awips2/python \
   "${pkg_path}" \
   || exit 1

# Merge lib64 into lib to avoid problems with installing into the virtualenv
if [ -d "%{_build_root}/awips2/python/lib64/" ]; then
    rsync -a %{_build_root}/awips2/python/lib64/ %{_build_root}/awips2/python/lib || exit 1
    rm -rf %{_build_root}/awips2/python/lib64
fi
```

Again, the merge of lib64 into lib is included to ensure that lib, not lib64, is the location of files owned by AWIPS Python FOSS RPMs.

Adding and Upgrading Java FOSS

If you have to add a new Java FOSS package to CAVE or EDEX, or upgrade an existing package, the process is as follows:

1. Download the package(s) and all dependencies from Maven central
2. Sort downloaded JARs into: new / upgrade / no change
3. Install JARs into Eclipse project directories (create new projects if necessary)
4. Update MANIFEST.MF files and project classpaths
5. Create / update feature.xml files.
6. Fix any errors in Eclipse.
7. Test your changes.

A description of these steps follows.

Note that the steps generally apply regardless of what AWIPS components use the FOSS package(s), though there are some extra steps that apply only to FOSS used by EDEX; these extra steps are labeled as such.

These steps only apply specifically to COTS/FOSS packages. For general information on creating new plugins in AWIPS, see Use of ADE.

Download packages from Maven central

You will have to run Maven to download the full dependency tree for all packages that you know you need. A Bash script is provided that will do this for you. You must have Maven installed to run the script. If you don't have Maven installed you may install it either from your Linux distribution's package repository, or the AWIPS build of Maven, which is provided by the awips2-maven RPM package.

This is the Bash script that downloads JARs using Maven, copy and paste this into a file named "mvntool.sh" (or whatever you want).

```
 #!/usr/bin/env bash

# Script to download JARs using Maven. Downloads binary and source JARs and
# all transitive dependencies, and also provides a dependency tree in a text
# file, showing the dependency relationships between all JARs downloaded.
#
# No configuration is required, the script uses its own separate Maven
# configuration and ignores the one in $HOME/.m2. If you are behind a proxy
# you will have to add the proxy config to the "settings" variable in
# this script.
#
# The script takes its input from stdin. One JAR per line, the format is
# groupID artifactID version
#
# For example:
#
# $ ./mvntool.sh <<< "org.geotools gt-coverage 21.1"
#
# Will download gt-coverage-21.1.jar, gt-coverage-21.1-sources.jar and all
# dependencies.
#
# Another example:
# $ ./mvntool.sh << EOF
# org.geotools gt-coverage 21.1
# org.geotools gt-affine 21.1
# org.geotools gt-referencing 21.1
# EOF
#
# Downloads all three of the named JARs.
#
# The JARS are downloaded to a newly created directory in /tmp, the name
# of the directory is included at the end of the script output. In that
# directory you will also find dependency-tree.txt that includes the
# dependency tree, with the JARs specfied in the input at the top level.
#
# If you need additional repositories you should make a copy of this script
# and add them yourself to the "settings" variable.
#
# Author: tgurney

PATH=/awips2/maven/bin:$PATH

settings="<settings xmlns=\"http://maven.apache.org/SETTINGS/1.0.0\"
  xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
  xsi:schemaLocation=\"http://maven.apache.org/SETTINGS/1.0.0
  http://maven.apache.org/xsd/settings-1.0.0.xsd\">
  <activeProfiles>
    <activeProfile>securecentral</activeProfile>
  </activeProfiles>
  <profiles>
    <profile>
      <id>securecentral</id>
      <repositories>
        <repository>
          <id>central</id>
          <url>https://repo1.maven.org/maven2</url>
          <releases>
            <enabled>true</enabled>
          </releases>
        </repository>
        <!--
        <repository>
```

```
            <id>osgeo</id>
            <url>http://download.osgeo.org/webdav/geotools/</url>
        </repository>
        -->
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>central</id>
          <url>https://repo1.maven.org/maven2</url>
          <releases>
            <enabled>true</enabled>
          </releases>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  <proxies>
    <!-- Uncomment and update this if you are a behind a proxy -->
    <!--<proxy>
      <active>true</active>
      <protocol>http</protocol>
      <host>localhost</host>
      <port>80</port>
    </proxy>-->
  </proxies>
</settings>
"

settingsFile=$(mktemp)
echo "${settings}" > "${settingsFile}"

theXml="${theXml}
    <project xmlns=\"http://maven.apache.org/POM/4.0.0\"
    xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
    xsi:schemaLocation=\"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd\">

    <modelVersion>4.0.0</modelVersion>

    <groupId>test</groupId>
    <artifactId>test</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>

    <name>Test</name>
    <url>http://www.example.com</url>

    <dependencies>
"



while read line; do
    if [[ "${line}" == "" ]]; then
        continue
    fi
    group=$(echo $line | cut -d ' ' -f1)
    artifact=$(echo $line | cut -d ' ' -f2)
    version=$(echo $line | cut -d ' ' -f3)
    theXml="${theXml}
        <dependency>
            <groupId>${group}</groupId>
            <artifactId>${artifact}</artifactId>
            <version>${version}</version>
```

```
        </dependency>
    "
    if [[ -z ${group} || -z ${artifact} || -z ${version} ]]; then
        continue
    fi
done

theXml="${theXml}
    </dependencies>
</project>
"

pomdir=$(mktemp -d)
echo "${theXml}" > "${pomdir}"/pom.xml
pushd "${pomdir}"
mvnargs="-s \"${settingsFile}\""
mvn ${mvnargs} dependency:copy-dependencies
mvn ${mvnargs} dependency:sources
mvn ${mvnargs} dependency:tree > dependency-tree.txt
mv dependency-tree.txt ./target/dependency
pushd ./target/dependency
for item in $(ls -1 *.jar | sed 's/.jar/-sources.jar/g'); do
    find ~/.m2/repository -regex ".*\/${item}" -type f -exec cp '{}' . \;
done
popd # pomdir
mv target/dependency/* .
rm -rf target
popd
rm -rf "${settingsFile}"
echo JARs located at "${pomdir}"
```

Provide the script with a list of Maven artifacts, and it will download those artifacts and all dependencies. Read the comments at the top of the script for further details on how to use it.

The Bash script will also produce a graphical dependency tree in a text file. Keep this file. You will need to refer to this throughout the process.

You will also want to create a separate list of all JARs that were downloaded and treat it as a checklist to make sure you account for every single one.

If you run into issues with the script

The simplest thing to do is to run the steps manually. You will have to be set up to run Maven, that, is, you must have the file .m2/settings.xml in your home directory and it must be correct. If you need a settings.xml file, you can use the one in the "settings" variable in the above script as a basis.

Then create a new empty directory, and in that directory create a "pom.xml" file with these contents:

```
  <project xmlns=\"http://maven.apache.org/POM/4.0.0\"
    xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
    xsi:schemaLocation=\"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd\">

    <modelVersion>4.0.0</modelVersion>

    <groupId>test</groupId>
    <artifactId>test</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>

    <name>Test</name>
    <url>http://www.example.com</url>

    <dependencies>
        <!-- Add one <dependency> element for each JAR you need, and fill in the three fields.
-->
        <dependency>
            <groupId></groupId>
            <artifactId></artifactId>
            <version></version>
        </dependency>
    </dependencies>
</project>
```

In the "dependencies" list, add one "dependency" XML element for each JAR you need, and fill in the group ID (example: org.apache.camel), artifact ID (example: camel-core) and version number. Then in the directory containing pom.xml, run these commands to download all the JARs and create a dependency tree listing:

```
 #!/bin/bash
mvn dependency:copy-dependencies
mvn dependency:sources
mvn dependency:tree > dependency-tree.txt
```

The only issue with this is that Maven does not put the source code JARs in the current directory for you--only the binary JARs. It does download the source JARs, but leaves them in $HOME/.m2/repository, so you will have to hunt for them in that directory.

Important notes on downloading JARs

- If you don't know the group ID and artifact ID for each of the packages you want, for new packages you can use the search function on mvnrepository.com, and for packages already in AWIPS, you can infer these: the group ID is usually the name of the Eclipse project that contains the FOSS (example: "org.apache.camel") and the artifact ID is in the filename of the JAR file (example: "camel-core" from the JAR file "camel-core-2.25.2.jar".
- You must always download and look over the full dependency tree. Even when just upgrading a single existing FOSS package, it is not enough to only download the newest versions of JARs that are already present. New dependencies can be introduced with new releases of software, and the only practical way to make sure you get them all is to use Maven to download them.
- The website mvnrepository.com is useful for informational purposes: specifically for looking up names of JARs, group IDs, artifact IDs, and version numbers. But for downloading JARs, you should run Maven, and it will fetch them from the Maven central repository.

Sort downloaded JARs into "new" / "upgrade" / "no change"

The JARs you have downloaded should be treated in three separate categories:

1. No change: JARs that already exist in an AWIPS git repository and that meet the version requirements.
2. Upgrades: JARs that exist in AWIPS but do not meet the version requirements, so they need to be upgraded to the version you just downloaded via Maven.
3. New: JARs that do not exist in AWIPS at all.

You can start categorizing JARs by doing, for each JAR you have downloaded, a search of all AWIPS git repositories using a regex that matches any version of that JAR. (example: "camel-core.*\.jar"). You should check all repositories, though it is especially important to check AWIPS2_foss and ufcore-foss since these are where we keep the vast majority of FOSS JARs.

If you find a JAR that matches and whose version number is at least as great as the JAR you have downloaded, it falls into the "no change" category and you can simply delete the downloaded JAR and mark it as "no change" in your checklist.

You may find an older JAR that matches the regex, in which case you will have to note that JAR as one to be upgraded.

Finally, you might not find any JAR that matches the regex, in which case you'll note that as a new JAR.

Install JARs into Eclipse project directories

For JARs to be upgraded, you may delete the old JAR and put the new JAR in the same directory.

For new FOSS JARs, the standard is that they go into an Eclipse project that has the same name as the Maven group ID. For example, all JARs in the "org.apache.camel" group should be installed into the "org.apache.camel" Eclipse project. If there is no Eclipse project yet for a given group ID, then you will have to create one, and the best way to do this is to make a copy of an existing Eclipse project that contains FOSS JARs. Put the new project in the ufcore-foss git repo. (The split between ufcore-foss and AWIPS2_foss exists only for historical reasons; these two are likely to be merged into a single repo in the future.)

Make sure you install both the binary and source JARs.

Update MANIFEST.MF files and project properties

Each Eclipse project you have created or changed will have a MANIFEST.MF that you will have to update. You also have to update the project properties to include the JARs on the classpath and link all of the source JARs to their corresponding binary JARs. The steps are the same regardless of whether you are adding new JARs or upgrading existing ones. For each Eclipse project, do the following:

1. Refresh the Eclipse workspace to make sure the latest file list is visible.
2. Right-click on the project in the Package Explorer, and go to Properties->Java Build Path.
3. In the Libraries tab, remove all JARs that you have deleted. Then click "Add JARs" and select all of the non-sources JARs that you added to that project.
4. For each JAR in the list, click the triangle to the left of the JAR, double-click on Source Attachment, click Browse and then select the sources JAR that corresponds to that JAR.
5. Still in the Properties window, switch to the Order and Export tab and check the box next to each JAR to mark it as exported. Now click OK to close the Properties window.
6. In the Package Explorer, open META-INF/MANIFEST.MF.
7. If the project is a new project, fill in the ID field with the Maven group ID (example: org.apache.camel).
8. Fill in or update the version field to match the FOSS project version.
9. Update the Name field if necessary. Just put something descriptive in this field; it is for human use only.

10. In the Execution environments list, confirm that the listed JRE version matches the JRE version used by the AWIPS branch you are working in. If it doesn't, click Remove, then Add, and select the correct version. (It will most likely be either JavaSE-1.8 or JavaSE-11.)
11. Switch to the Dependencies tab in MANIFEST.MF. Look at the Required Plug-Ins list, and confirm that the list of required plug-ins matches the list of dependencies as produced by Maven. If it doesn't, use the Add and Remove buttons to add and remove plug-ins as needed so that all dependencies are satisfied. Refer to the dependency-tree.txt file to make sure that you get all the dependencies for every JAR in this Eclipse project. Also note that we are using the Required Plug-ins list rather than the Imported Packages list; this is the preferred way to do it in AWIPS.
12. Switch to the Runtime tab. Remove all items from the Exported Packages list. Then click add, and add all of the packages listed.
13. Still in the Runtime tab of MANIFEST.MF, remove all JARs from the Classpath list. Then click the Add button and select every non-sources JAR in this Eclipse project. Make sure the "Update the build path" box is checked before clicking OK.
14. Switch to the Build tab of MANIFEST.MF and review the Binary Build list at bottom-left. All non-sources JARs, plus the META-INF folder, should have a check mark. If not, you likely missed an earlier step.
15. Uncheck all boxes in the Source Build list.
16. Review the last two tabs (the MANIFEST.MF and build.properties tabs) and make sure the contents are correct.
17. Save the MANIFEST.MF file.

Create/update feature.xml files

If the FOSS package is already being used in AWIPS code, it will be listed in a feature.xml file somewhere. Do a textual search through all "feature.xml" files for the name of each Eclipse project that you modified. Usually if you find any such references, they will specify the version number "0.0.0"; leave these alone. If there is an actual version number there, though, update it to match the new version number of the FOSS package.

Special steps for new EDEX FOSS

FOSS packages used by EDEX are built and packaged individually. So for each new FOSS project that is or will be used by EDEX, you have to create a new Eclipse feature for the project. These features are kept in the ufcore repo, in the "features" directory. Follow these steps to create a new feature and integrate it into the EDEX build:

1. Create a new directory named "com.raytheon.uf.edex.foss." followed by the name of the Eclipse project, followed by ".feature". For example, for the "org.apache.camel" project, the name of the feature directory is "com.raytheon.uf.edex.foss.org.apache.camel.feature".
2. Copy the contents of the directory from one of the other EDEX FOSS feature directories. You need the following files: "build.properties", "feature.xml", ".project".
3. Open the ".project" file in the new directory, and change the project name to the name of the directory. Then save and close the file.
4. Open the "feature.xml" file, change the feature id attribute to the name of the directory, update the label attribute, update both version numbers to match the version number of the FOSS project, and update the plugin ID attribute to match the name of the Eclipse project. Then save and close the file.
5. Import the new feature into Eclipse via the Package Explorer: Right-click in the Package Explorer, click "Import", under General select "Existing Projects into Workspace" and click Next, then click Browse and select the ufcore/features directory. Then click Finish to import the project.
6. Add the feature to build.xml: Open AWIPS2_baseline/edexOsgi/build.edex/build.xml, you will find a list of "antcall" elements that refer to EDEX FOSS features, add your new feature to this list. The order is significant--a feature must not be built until all of its dependencies are built. So you have to put the feature in the list after the "com.raytheon.uf.edex.foss.feature" but before all FOSS packages that depend on that package, and after all packages that that package depends on.

7. Update comps.xml: Each EDEX FOSS feature results in an RPM package being built for that feature, and that RPM package has to be added to all of the EDEX package groups. Open AWIPS2_build/installers/Linux/comps.xml. In this file you will find lists of EDEX FOSS packages. Add the new package to every list that contains awips2-edex-foss.

Fix any errors in Eclipse

You may see some errors in your Eclipse workspace, whether from following the above steps incorrectly or because upgrading some packages resulted in code breakage. Fix all errors in the Eclipse workspace.

Test your changes

You should be able to build and deploy EDEX from within Eclipse, run EDEX on your workstation, and run CAVE and connect it to your local EDEX. Run through this process before committing any changes.

RHHI

With the virtualization of the AWIPS servers using Red Hat Hyperconverged Infrastructure (RHHI) came changes regarding the architecture of the server hardware and how sites setup. The figures below demonstrate these differences and list which services and components are expected to be run out of each virtualized server.

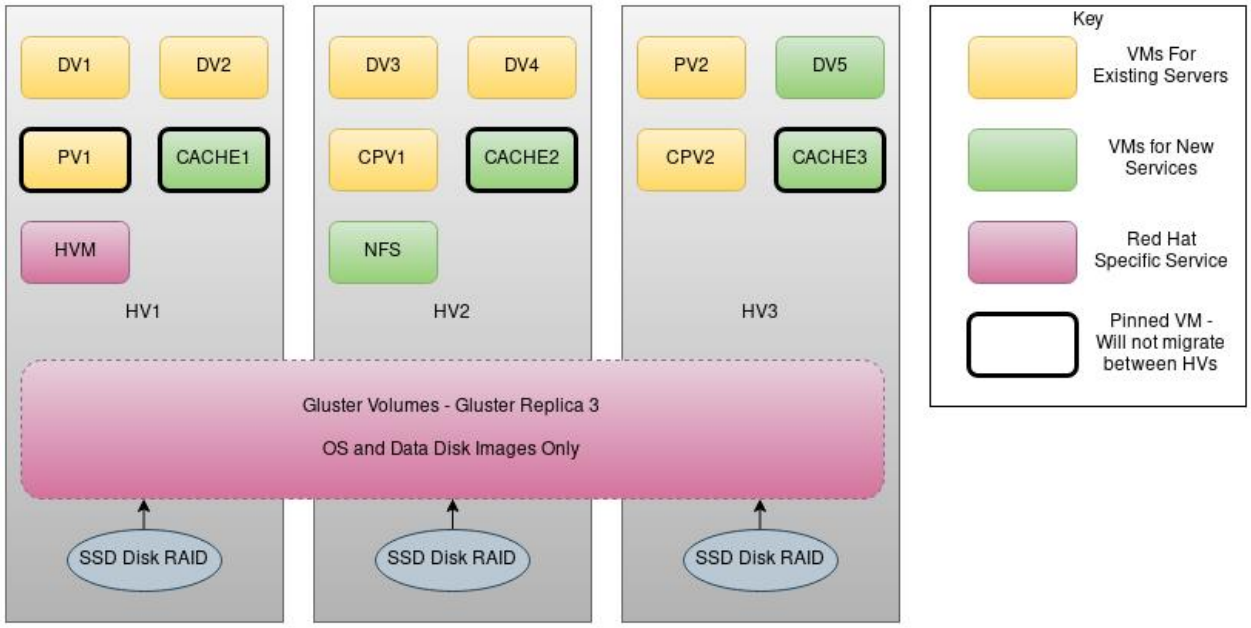

Figure 6.1 - WFO Server/VM Architecture

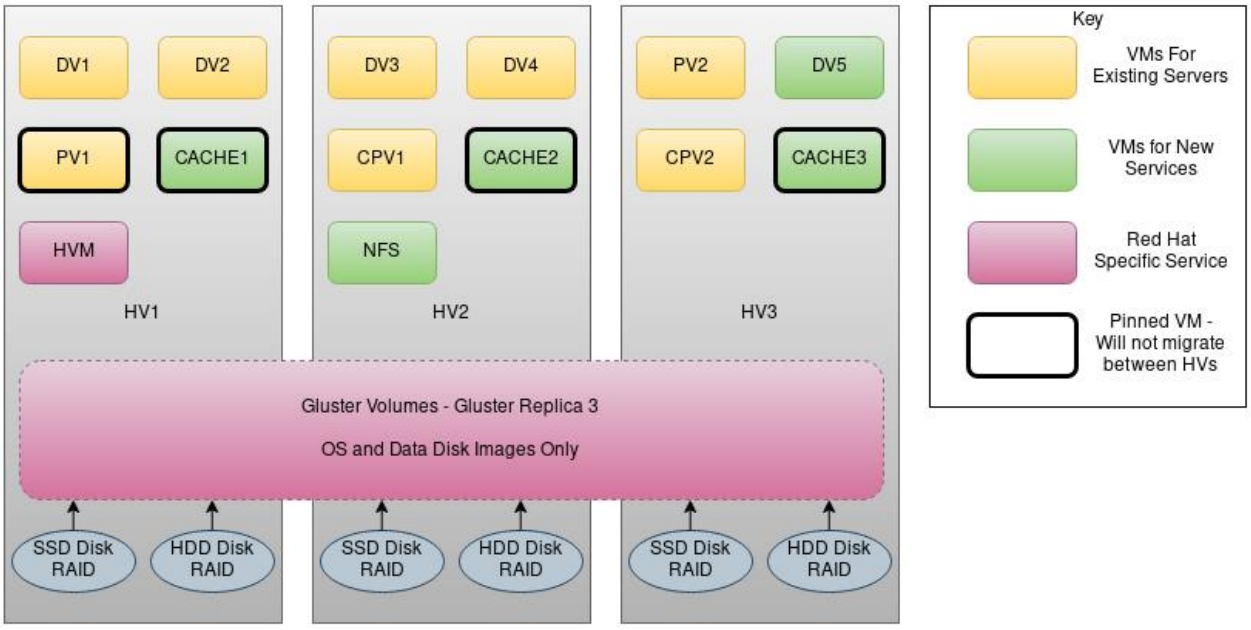

Figure 6.2 - RFC Server/VM Architecture

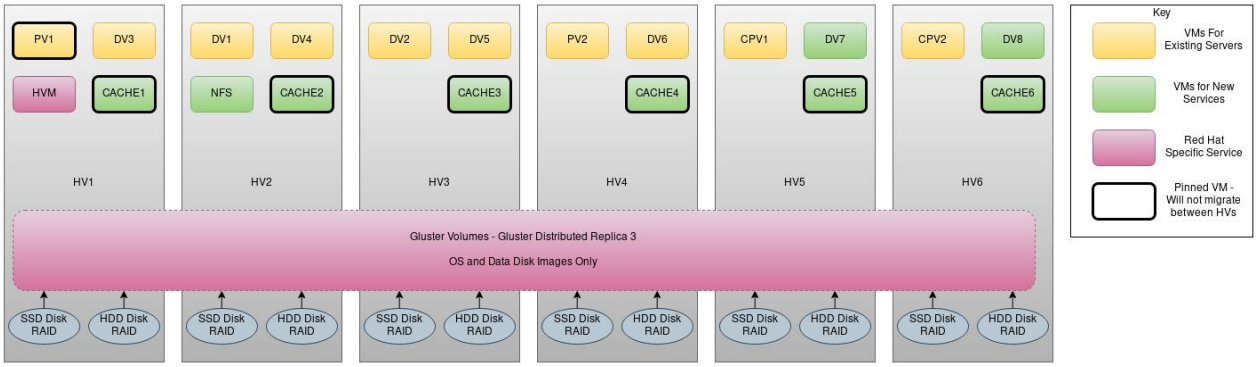

Figure 6.3 - RHQ & NC Server/VM Architecture

DV1	Postgres, RCM
DV2	PyPIES
DV3/4/5/n...	ingest, ingestDat, ingestGrib, request
PV1	BMH, Comms Manager
PV2	BMH, Comms Manager
CPV1	Qpid, LDM
CPV2	registry
CACHE1/2/3/n...	Ignite

Table 6.1 - Server and Services

CAVE Performance Logging

UI Thread Monitor

The UIThreadMonitor class monitors stalls on the CAVE UI Thread that make the application appear to "freeze" or "hang". When the UI Thread is stalled for longer than a specified threshold, a message is logged in the *cave_yyyymmdd_hhmmss_pid_nnnnn_logs.log* file indicating "UI Thread stalled for more than 500 ms:" followed by a stack trace of the UI thread to show where the thread may be stalled. If you see this message in your logs you should examine where the UI Thread is stalled and if it is in code you are developing, consider moving any long running processes onto a separate thread.

The default 500 ms threshold for declaring the UI thread stalled can be changed by setting the ui.thread.monitor.threshold.millis system property. This property is specified in the awips.product and developer.product files and can be overridden in the field by editing the value in the applicable *.ini file under /awips2/cave.

Request Logging

All requests leaving the CAVE application should be logged immediately prior to the request being sent and upon a response being returned. The response log should include the time in ms between the request being sent and the response being received. All responses should be logged whether the request was successfully handled, or an error condition or timeout was encountered. Each request/response log message should contain some kind of id which is unique to that request within the CAVE session to allow the request/response log messages to be properly paired.

Thrift request performance logging

CAVE Thrift requests and responses are logged to the *cave_yyyymmdd_hhmmss_pid_nnnnn_requests.log* file under ${user.home}/caveData/logs/consoleLogs/${HOSTNAME}/. These requests are automatically logged when ThriftClient.sendRequest() is called.

When creating a new request class that implements IServerRequest or modifying an existing one you should provide a toString() method that provides the info about the request that you want to appear in this log file.

The toString() method should include, at a minimum, the class name and any parameters that maybe useful in debugging performance issues. Large data arrays should not be logged in their entirety but should include their size. Parameters should be logged with their name followed by their contents enclosed in square brackets. Although they are not IServerRequests, the com.raytheon.uf.common.pypies.request.AbstractRequest class hierarchy contain good examples of toString() methods for requests.

Example Thrift request log messages:

```
INFO  2021-04-12 15:13:52,138 9210 [main] CaveRequestLogger: Sending request to URL http://ec-om
INFO  2021-04-12 15:13:52,214 9211 [main] CaveRequestLogger: Request id[825f8fe4-05bf-4b11-8e5b-
```

PyPies request performance logging

CAVE PyPies requests are logged to the *cave_yyyymmdd_hhmmss_pid_nnnnn_pypies.log* file under ${user.home}/caveData/logs/consoleLogs/${HOSTNAME}/. These requests are automatically logged whenever PyPiesDataStore functions are called. When creating a new request that extends com.raytheon.uf.common.pypies.request.AbstractRequest or a response that extends com.raytheon.uf.common.pypies.response.AbstractResponse you should provide a toString() method that provides the info about the request/response that you want to appear in this log file.

Example PyPies request log messages:

```
INFO 2021-04-12 15:14:26,274 9221 [Worker-4: Requesting Grid Data] PyPiesRequestLogger: Sending
INFO 2021-04-12 15:14:27,699 9338 [Worker-4: Requesting Grid Data] PyPiesRequestLogger: Request
```

General purpose performance logging

When creating or maintaining code that can consume significant processing time you should consider adding performance logging. Performance logging should be done using a PerformanceStatus handler vs using System.out.println() or the normal UFStatus or slf4j Logger used for error/debug logging . All PerformanceStatus log messages are logged to the cave_yyyymmdd_hhmmss_pid_nnnnn_perf.log

Example of general purpose performance logging using PerformanceStatus:

```
// get a performance status handler for your function
private static final IPerformanceStatusHandler perfLog = PerformanceStatus
    .getHandler("MapScalesManager:");

.
.
.

// get current time in millis immediately prior to the action
long t0 = System.currentTimeMillis();
try {
    getScaleBundle();
} finally {
    // following completion/failure of the action log the duration
    perfLog.logDuration("Loading scale " + this.displayName,
        (System.currentTimeMillis() - t0));
}
```

Example general purpose performance logging messages:

```
INFO 2021-04-12 15:14:01,041 9216 [main] PerformanceLogger: MapScalesManager: Loading scale N. H
INFO 2021-04-12 15:14:01,053 9217 [main] PerformanceLogger: MapScalesManager: Loading scale Nort
INFO 2021-04-12 15:14:01,066 9218 [main] PerformanceLogger: MapScalesManager: Loading scale CONU
INFO 2021-04-12 15:14:01,076 9219 [main] PerformanceLogger: MapScalesManager: Loading scale Regi
INFO 2021-04-12 15:14:01,085 9220 [main] PerformanceLogger: MapScalesManager: Loading scale Stat
INFO 2021-04-12 15:14:01,100 9221 [main] PerformanceLogger: MapScalesManager: Loading scale WFO
```

Special Case Ingest Using Manual Dropped-in Files

One of the most useful special cases is that of manual ingest. Manual ingest is useful for testing and small deployments that do not have an LDM. The nice thing about manual ingest is that the standard ingest pipelines are reused and the only change is how notifications originate, as shown in **Figure 5-3**.

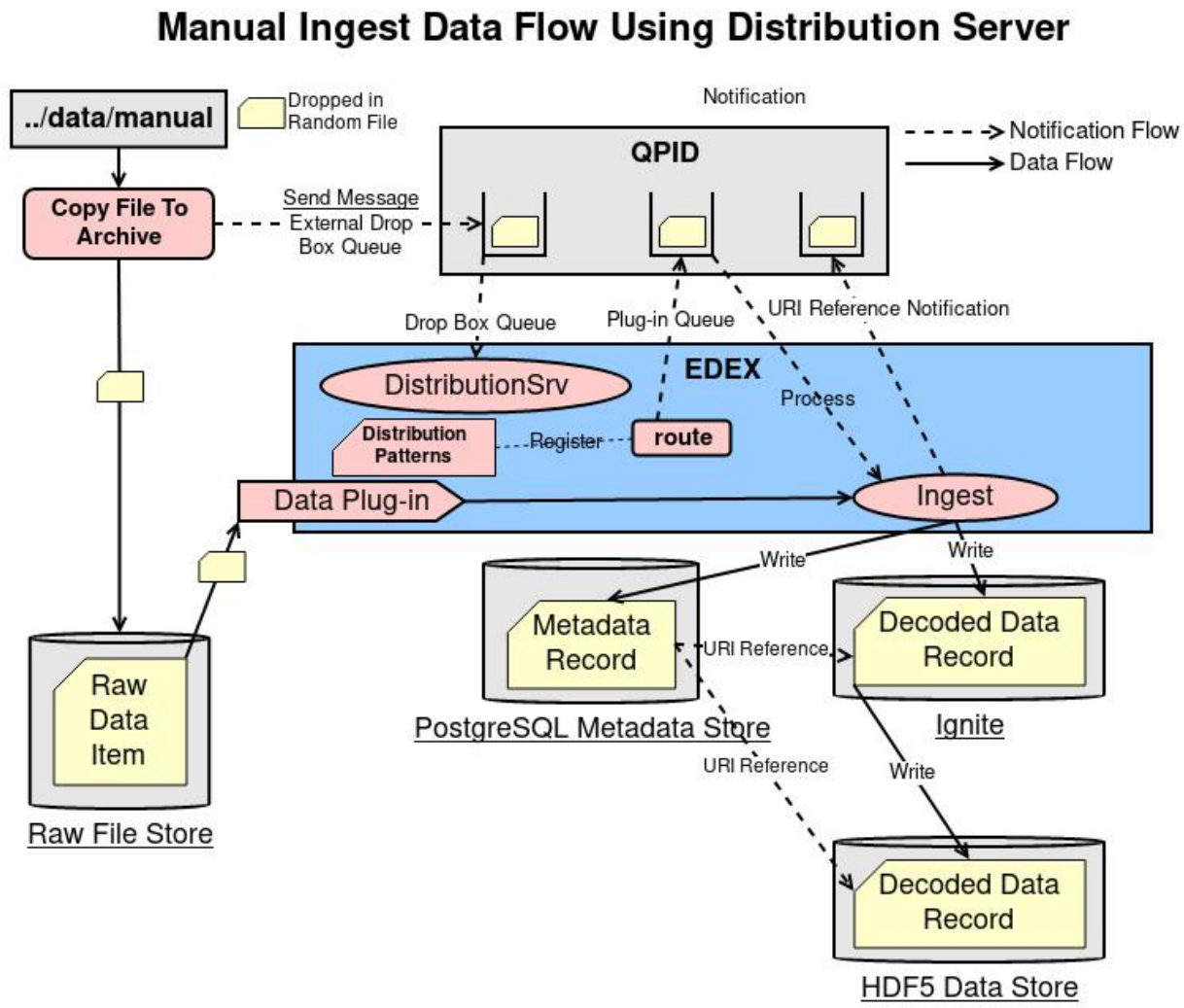

Figure 5-3. Manual Ingest Data Flow Using Distribution Server

- Data flow originates with files dropped into the "**{edex.home}/data/manual**" endpoint on an EDEX box. It does not matter how the files get there.
- A special EDEX plugin is listening to this file endpoint using a standard apache Camel file sniffer component.
- The "manualingest" plugin moves the dropped-in file to the raw archive. Files disappearing from the file endpoint indicate that EDEX is running and files are getting sniffed up.
- A notification to the external dropbox queue is sent to start the ingest pipeline from the "manualingest" plugin.
- From this point on ingest is identical to the standard ingest used by the LDM as described earlier.

Standard AWIPS Data and Notification Flow

AWIPS II establishes a standard pattern for ingesting raw data and making that data available to all components of the system. EDEX decodes incoming data and converts it on ingest into a set of metadata records for querying and data records for decoded data. **Figure 5-1** displays a top-level view of how the data flows from the Satellite Broadcast Network (SBN) system all the way to the CAVE display component. This data flow is generic and applies to all data types that come over the SBN. Local radar, Local Data Acquisition and Dissemination (LDAD) data, and manual data flow are special cases. These special cases vary in how data arrives at EDEX but they follow the standard pattern once the data gets to EDEX. The following describes the standard data flow.

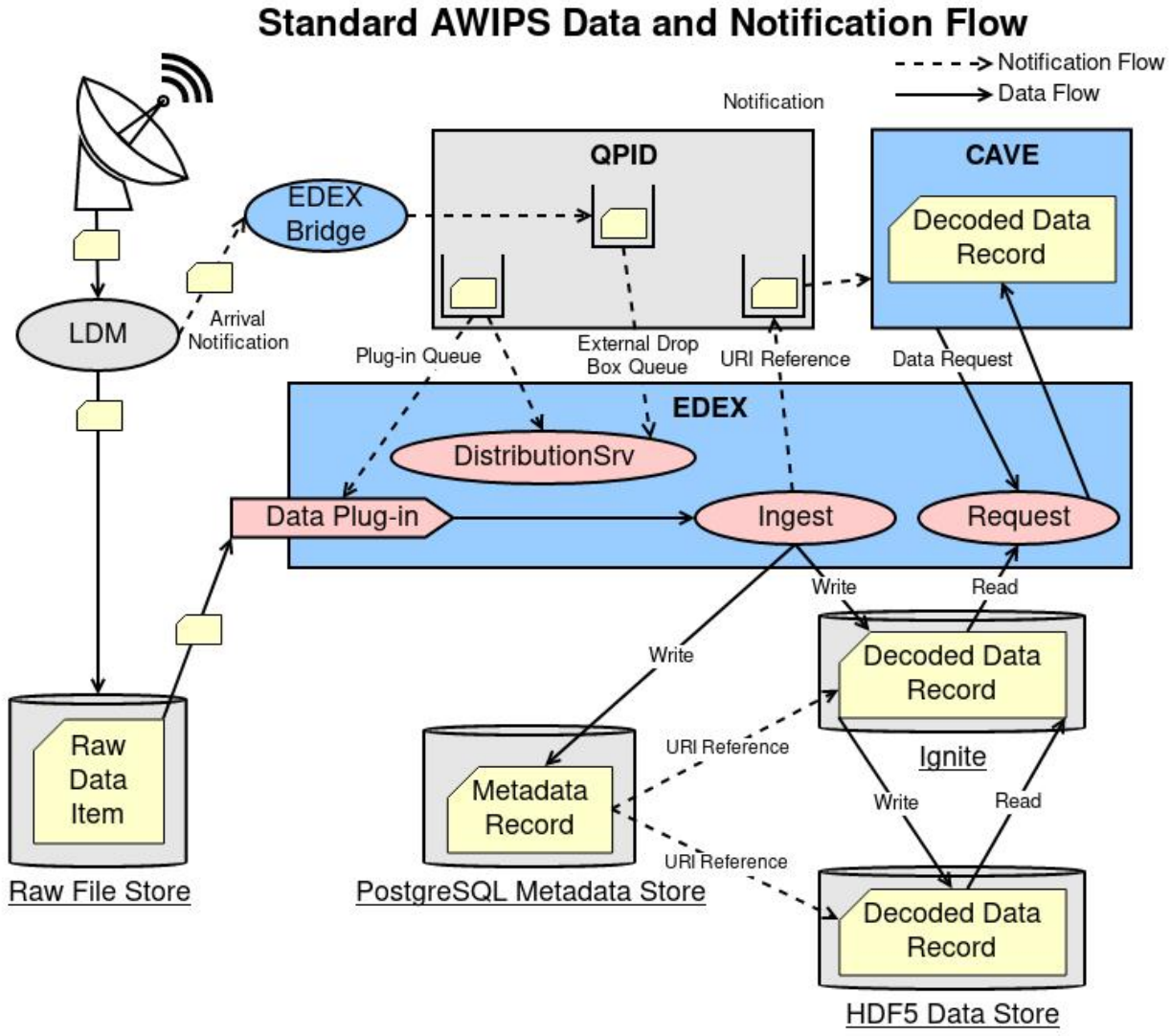

Figure 5-1. Standard AWIPS Data and Notification Flow

Figure 5-1 shows the major steps and components involved in the data flow that originates from the SBN.

- LDM (Local Data Manager) with the SBN module interfaces directly with the SBN satellite receiver over a multicast UDP (User Datagram Protocol) interface from a DVB-S. The LDM control file (pqact.conf) determines the location and filename of where the raw data gets written to, often to a VM (virtual machine) mounted directory such as /data_store. These files are raw data files that have not been decoded.

- The EDEX Bridge process interfaces directly with LDM to watch for new data arrivals. If specified in the LDM control file, once each data file has completed writing to disk then EDEX Bridge will send a notification message over JMS to the External Dropbox queue to be consumed by EDEX.

- The Distribution Service in EDEX listens continuously to the External Dropbox queue. The job of the Distribution Service is to route the data arrival notification to the appropriate plugin queue to and start the decode and ingest pipeline. The Distribution Service's routing is controlled by the plugin's distribution XML file, which is stored in the localization file system. Contained within the distribution XML are WMO header and file name regular expressions filters that specify the files that the plugin is designed to decode and store.

- The data plugin inside EDEX receives the notification sent by the Distribution Service. The plugin uses its decoder and other data specific support capabilities to extract metadata and records from the raw file. The raw file is accessed directly from the raw file store. As part of this process, each piece of decoded data receives a unique URI known as a dataURI.

- The decoded data is sent to the Persist Service which sends the data to be cached by Ignite. While holding the data for quick retrieval, Ignite will simultaneously send the new data to PyPIES which stores the data to HDF5 files based on the unique URI.

- The metadata of the decoded data is sent to the Index Service and is stored as PostgreSQL records with specific fields identified to form the URI reference.

- EDEX generates a log message and sends out a notification with URI of the newly decoded and stored data. Each CAVE (and potentially other EDEX instances or applications) listens for these notifications to know immediately when new data has been processed.

AWIPS II Architecture

AWIPS II is a client/server plugin-based architecture. It consists primarily of a server application named EDEX (Enterprise Data EXchange) and a desktop application named CAVE (Common AWIPS Visualization Environment). The AWIPS II design is based on a Service Oriented Architecture (SOA) where the applications utilize services and have no knowledge of the underlying implementations of those services. The applications communicate with one another with HTTP (Hypertext Transfer Protocol) and JMS (Java Messaging Service) and only communicate with established services.

Other applications provide some of the services and components of the server architecture as separate processes. However, these other applications can mostly be considered as FOSS (Free and Open Source Software) that just run out of the box without AWIPS II modifications or development. These other server components are:

- LDM: Provides data feed from the SBN (Satellite Broadcast Network)
- Qpid: Serves as the JMS message broker
- Postgres: Stores decoded metadata and data
- Ignite: Caches decoded data for fast retrieval
- PyPIES: Stores decoded data

EDEX Architecture

The EDEX server application is a SEDA (Staged Event Driven Architecture) application that is built upon Apache Camel (http://camel.apache.org/) and Spring (http://projects.spring.io/spring-framework/). This type of architecture is event driven, where code is executed when events occur. Events include JMS messages, HTTP requests, file arrival messages, cron-based events that occur periodically on a timer, and more.

EDEX instances can be clustered to improve performance and they will automatically distribute the workload of tasks. Each EDEX instance is started in a specific mode. A mode indicates a set of responsibilities and event routes that this EDEX instance will be supporting. The most commonly used modes are:

- request: Responsible for providing access to the data through service requests
- ingest: Responsible for decoding and storing data. Also supports some post-processing and cleanup tasks.
- ingestGrib: Responsible for decoding and storing grib data (both grib1 and grib2 format)
- ingestDat: Responsible for generating post-processed data to support the DAT (Decision Assist Tools) suite

CAVE Architecture

CAVE is an OSGi (https://www.osgi.org/) (Open Services Gateway initiative) plugin-based application built on top of the Eclipse RCP (https://eclipse.org/) (Rich Client Platform). Using Eclipse RCP, CAVE detects what plugins are installed and provides the perspectives, menu items, preferences, dialogs, etc that are contributed by each installed plugin. Many of CAVE's capabilities are associated with Eclipse 'Perspectives.' A perspective consists of a unique UI (user interface) and functionality. Examples of perspectives are D2D (Display Two Dimensional), GFE (Graphical Forecast Editor), Hydro, MPE (Multipoint Precipitation Editor), NCP (National Centers Perspective), and Localization. Eclipse RCP also provides many capabilities automatically, such as the ability to drag and drop tabs to reorganize a window layout.

CAVE uses SWT (https://www.eclipse.org/swt/) (Standard Widget Toolkit) to provide native operating system widgets so that user interfaces retain the look and feel of each operating system. For displays of maps, weather data, etc, CAVE uses OpenGL (http://www.opengl.org/) (Open Graphics Language) to take advantage of the rendering capabilities of the graphics card.

Layers and Interfaces

The AWIPS II architecture follows SOA principles through the use of layers and interfaces. Each component has knowledge of the interfaces it needs but not of the implementation details behind those interfaces. This makes the code more adaptable and maintainable should the underlying implementations need to change in the future. Code should not be dependent on specific implementations to work.

Examples:

- JMS communication: Use the established JMS interfaces (https://docs.oracle.com/javaee/7/api/javax/jms/package-summary.html). Components do not need to know that the underlying JMS implementation is Qpid (Queue Processor Interface Daemon) which uses the AMQP (Advanced Messaging Queuing Protocol) protocol to send and receive messages.
- IDataStore storage: Use the DataStoreFactory and IDataStore methods. Components do not need to know that the underlying caching scheme is Ignite, nor that the data store implementation is PyPIES (Python Process Isolated Enhanced Storage) and that it stores data to the HDF5 (https://www.hdfgroup.org/HDF5/) (Hierarchical Data Format) file format.
- Graphics rendering: Use the IGraphicsTarget interface. Components do not need to know that the primary rendering implementation uses OpenGL.

JMS and QPID

For information on JMS and messaging topology, see the following:

- **JMS Overview:** https://docs.oracle.com/javaee/7/tutorial/jms-concepts.htm (https://docs.oracle.com/javaee/7/tutorial/jms-concepts.htm)
- **JMS APIs - javax.jms:** https://docs.oracle.com/javaee/7/api/index.html?javax/jms/package-summary.html (http://docs.oracle.com/javaee/5/api/)
- **Camel JMS configuration:** http://camel.apache.org/jms.html (http://camel.apache.org/jms.html)
- **Spring JMS Overview:** https://docs.spring.io/spring-framework/docs/5.1.x/spring-framework-reference/integration.html#jms (https://docs.spring.io/spring-framework/docs/5.1.x/spring-framework-reference/integration.html#jms)
- **EDEX:** A single Qpid connection per JVM, with one session per thread. Usually each session has one consumer. If a thread/route sends to multiple other queues (i.e., distribution) it will have a producer for each destination it sends to. Pooling is setup for each of the JMS API objects. Camel/Spring will create and close the resources for every message, so pooling was setup to reuse JMS resources. These can be found at **com.raytheon.uf.common.jms**.
- **CAVE:**
    - ***Receive:*** All pulls should go through com.raytheon.uf.viz.core.notification.jobs.NotificationManagerJob. NotificationManagerJob will only have one connection, with many sessions. Each queue/topic is listened to async via MessageListener interface. Each message is delivered async via Eclipse Job.
    - ***Send:*** No standard framework. Look up **com.raytheon.uf.viz.core.comm.JMSConnection** and use JMS calls directly. See **com.raytheon.viz.warngen.comm.WarningSender.java**

GeoTools and JTS Use - Best Practices

A 2D Coordinate Point or Position

The basic building block of all the geospatial logic in AWIPS II is the concept of a 2-dimensional position which is represented by a numerical x and y. Many simple geographic positions are expressed using latitude and longitude (often shortened to Lat/Lon) where longitude is represented by the x value and latitude is represented by a y value. Positions on the screen and inside the application's canvas are also represented by a numerical x and y. Many different classes can be used to represent the same idea of a position, and unfortunately due to all the different libraries it is impossible to use a single representation. Below is a list of many of the classes used in AWIPS II to represent x,y points.

- **org.locationtech.jts.geom.Coordinate**: This is a good default representation for geospatial usage. It is the building block of all higher level JTS types so it is common in code that is taking advantage of JTS data structures. It is also the most common representation for Lat/Lon values.
- **org.locationtech.jts.geom.Point**: This is just a wrapper around a Coordinate object to use it as a JTS Geometry. JTS Geometries provide many useful algorithms and this class must be used for interacting with many of those algorithms. Outside of code using JTS to interact with Geometries this class should be avoided since Coordinates are simpler.
- **org.opengis.geometry.DirectPosition,org.geotools.geometry.DirectPosition2D**: This is used extensively when math transforms are being used to reproject data between different coordinate systems. This is very commonly used for coordinates in a projected coordinate system or points in a grid (more on that later), it is not as common for Lat/Lon points or display points.
- **java.awt.Point:** This class is limited because it uses an int for x,y values. It is very rarely used in AWIPS II. It must be used sometimes to interact with AWT specific code and occasionally it is used to represent a grid point that has been rounded to an int value.
- **java.awt.geom.Point2D**: This class is not common in AWIPS II, it is the base class of DirectPosition2D but it is not common to use this type directly, only when it is necessary for interacting with AWT libraries.
- **org.eclipse.swt.graphics.Point:** This is the SWT equivalent of the AWT point. It should never be used in Edex/Common code and should only be used in CAVE when interacting with SWT directly.
- **double[]**: This representation can be appealing because it does not lock you into any specific library for representation. The downside is that it is not clear whether this is a single point, which could be x,y or x,y,z or if this is a sequence of points and it is often a source of confusion what to use when a method accepts or returns an array for a coordinate. This only place that an array should be used for a single point is if it is absolutely necessary to use an existing function call that takes an array. When representing many points the efficiency of a raw array may make it a useful option, just ensure it is clear whether the array contains x,y points or x,y,z points and consider using a class like PackedCoordinateSequence to give the data some structure.
- **double x; double y;** It is often tempting to use separate variables for the x and y components. This is fine for local variables and private methods but in general for reusable and maintainable code it is better to use one of the other representations, code refactors often move logic to new methods and it causes problems when the value is two variables that cannot be returned together. Separate x and y variables also tends to clutter method signatures, especially if the logic starts getting complex.

The same set of classes can be used to represent a variety of different positions, for example a Coordinate for a Lat/Lon position is very different from a Coordinate for a position on a computer monitor. To avoid confusion it is very important to indicate what type of position an Object represents, when using geospatial position always indicate which CRS is used or which variable contains the CRS used for the coordinate (CRS will be described more later). It is common for variables representing Lat/Lon values to have names containing "ll" or "latLon", it is very important to provide descriptive names and/or comments for variables representing a position.

Many of the coordinate representations have a third, z component for supporting a 3D implementation. Most AWIPS code will ignore the z component of any class that has it, all coordinates should be assumed to be 2D unless stated elsewhere specifically. Most classes and libraries within AWIPS use a separate variable for any elevation information if it is available.

JTS and Geometries

AWIPS II uses the JTS library to provide the data representation and many algorithms for 2D geometries. The most basic geometry class is com.vividsolutions.jts.geom.Geometry, there are subclasses to represent things like Points, Lines and Polygons. Geometries are created using a GeometryFactory, usually in AWIPS II new GeometryFactories are created when they are needed using the default constructor. JTS provides extensive javadoc and there is also further documentation and examples easily found on the internet of how to use JTS.

Coordinate Reference Systems

AWIPS II and the geotools library are written with the assumption that the earth is a 3-dimensional ellipsoid but weather data and displays are usually represented in a 2-dimensional space. A Coordinate Reference System (CRS) is used to describe how to translate the different 2D spaces onto the earth and it is also possible to translate data from one CRS onto another. In order to combine different data types or display any data in AWIPS II you must know the source CRS for any data and what CRS you want to use for the display, and then it is possible to use CRS.findMathTransform() to get a MathTransform which can be used to reproject the data. Simple Lat/Lon data often doesn't include a CRS so DefaultGeographicCRS.WGS84 can be used as the CRS for simple Lat/Lon data.

The basic idea of using a MathTransform to convert data to a different projection is very straightforward and usually works well 90% of the time, however if you spend much time developing geospatial code then you will inevitably run into times when it doesn't work. Here are two common examples of the types of problems that come up:

1. If you look at a globe and draw a circle over the dateline (180° longitude) then reproject this circle onto a standard Lat/Lon map that is split at the dateline it becomes 2 half circles. To make things worse since the circles were connected, they will now be connected all the way across the world. If you fill in the circle with color then it turns into a streak across the entire world. This has become known as a world wrap problem, although they are most common along at 180° longitude it is possible for other projections to get this type of problem anywhere in the world. The geotools library does not offer much help with this problem so the WorldWrapCorrector has been developed to split geometries along to fix this.
2. If you have a standard Lat/Lon map of the entire world and you look at Antarctica then it is roughly rectangular. If you try to make a polygon to represent this you will need points on the bottom corners so that there will be straight edges and a straight line along the bottom. If you take that polygon and you try to put it on a globe (or a southern polar stereographic map) then you will run into problems. The points that were at the corners are now both at the south pole, in fact they are the same point which makes it an invalid polygon. The 2 edges are also the same line and the bottom line has just collapsed into a single point, you no longer have a useful polygon. If you start with a valid Antarctica on a globe and transform to a Lat/Lon map then you start off with a world wrap problem, however even if the line is split correctly it still is not possible to generate a valid polygon because there are no corner points. There is not currently a good way of handling this sort of problems near the pole. Often this type of problem arises when data is displayed on a projection where it is not very useful, in these cases it is often sufficient to detect the problem and skip the problematic data.

Envelopes

ReferencedEnvelope. When working with spatial data we tend to think of rectangular areas bounded by a lower left and upper right lat/lon. This is fine as long as you work in a single map projection. However, when changing projections (say, from unprojected lat/lon or mercator) to a conic projection (like Lambert Conformal), that rectangle gets "curved" so now the bounding lat/lon envelope must be expanded to include all the necessary points. The

org.geotools.geometry.jts.ReferencedEnvelope class can be used to help with this problem. See **com.raytheon.uf.common.geospatial.MapUtil getBoundingEnvelope** method to see how to convert an envelope in one projection to another.

Grid Geometries

GridGeometry. AWIPS II uses a lot of gridded data. This data consists of a rectangular array of data values normally at evenly spaced points in a particular projection. The lat/lon of each sample is called a gridPoint. When rendering this data as an image the data for a gridPoint is spread over a rectangular area called a gridCell. The relationship between gridPoints and gridCells is not well defined in most NWS documents (e.g., the Gridded Binary (GRIB) specification). AWIPS II assumes that the gridPoint is at the center of the gridCell. This seems to make the math work out to match that in AWIPS I in most cases. The functions in **com.raytheon.uf.common.geospatial.MapUtil** that transform coordinates from grid coordinates to lat/lon and vice versa allow you to specify the PixelOrientation as CENTER, LOWER_LEFT, LOWER_RIGHT, UPPER_LEFT, or UPPER_RIGHT to allow you to get the appropriate value for your use.

Advanced Problems

Float Precision

Be very careful when converting coordinates between a float and a double. Problems often occur if a coordinate is converted back and forth because values that should be equivalent are not. Sometime this is handled by using approximations instead of equals but it is always better if you know specifically which primitive to use and consistently keep the precision the same.

Point-in-Polygon Queries

Can be sped up immensely by using PreparedGeometry. See **com.raytheon.viz.gfe.ui.zoneselector.ZoneSelectorResource setGeometry** and contains methods.

CRS Identifiers

Developers who are familiar with other gis software may have seen CRS represented using standard identifiers such as EPSG:4326 or EPSG:3857. Identifiers like these are rarely used in AWIPS II because most projections used in AWIPS II are using custom parameters that are not defined by the EPSG. When transferring or storing a CRS AWIPS II will use well known text (WKT) instead of an identifier because the WKT contains a full definition of the projection parameters. If you are implementing capabilities that need to use an identifier in AWIPS II it can easily be translated to a CoordinateReferenceSystem object using CRS.decode(String). Here is an example of some CRS WKT so that you will know what it is when you see it.

```
PROJCS["Lambert_Conformal_Conic_1SP",
  GEOGCS["WGS84(DD)",
    DATUM["WGS84",
      SPHEROID["WGS84", 6378137.0, 298.257223563]],
    PRIMEM["Greenwich", 0.0],
    UNIT["degree", 0.017453292519943295],
    AXIS["Geodetic longitude", EAST],
    AXIS["Geodetic latitude", NORTH]],
  PROJECTION["Lambert_Conformal_Conic_1SP"],
  PARAMETER["semi_major", 6371200.0],
  PARAMETER["semi_minor", 6371200.0],
  PARAMETER["central_meridian", -95.0],
  PARAMETER["latitude_of_origin", 25.0],
  PARAMETER["scale_factor", 1.0],
  PARAMETER["false_easting", 0.0],
  PARAMETER["false_northing", 0.0],
  UNIT["m", 1.0],
  AXIS["Easting", EAST],
  AXIS["Northing", NORTH]]
```

Bulk Coordinate Transformations

If you are trying to use a MathTransform on many points (thousands) then it is inefficient to transform each point individually, instead it is faster to copy all the data into a large double[] and transform all the points at once. If possible the same array should be used for the source and destination of the transform so that memory usage is minimized.

Displaying Large Datasets

For many types of data it is common to have more detailed data available than can be displayed at once. There are some common techniques for breaking apart large data sets so that the data can be displayed quickly while maintaining accuracy.

1. Decimate the data into smaller, less detailed versions of the product
2. Requesting only subsets of the decimated data
3. Dynamically updating the display to respond to pan/zoom by the user

The result is that the user has the data they want to see but each individual request is only for a relatively small amount of data. This allows the requests to process quickly and the entire process can occur in the background without detrimental impact to the user.

For example in the maps database there are points outlining each state, and California alone needs 50,000+ points to describe the details of the coastline. Trying to load all of the states at once would request and render a prohibitively large number of points. If you look at a map of the entire continental United States then you want to see 48 states but it is impossible to actually see all the details in the database. In this example California is only a few thousand pixels on the screen so it is unnecessary to draw 50 thousand points in the space available on the screen. The maps database contains numerous different versions of each state that have been simplified to remove some of the fine details. The smallest version of California has only 1500 points and would be much more appropriate for a view of the entire country. If the user zooms in on California then AWIPS II dynamically requests more detailed versions of California. As you zoom in the number of points for each state increases but some states will no longer be visible and will be removed from the request, so states like Florida quickly disappear which keeps the requests from getting large.

Large imagery data uses the same technique. For example when we ingest satellite data we get a very high resolution product, and the product can have more pixels of data then there are pixels on a computer monitor. AWIPS II stores multiple versions of the image at progressively lower resolution, for example if a product is 8192x8192 then it might store another version at 4096x4096 and another at 2048x2048. For requesting subsets of images AWIPS II uses the concepts of tiles. At a specific decimation level the data is divided into fixed size tiles, for example a 4096x4096 image could be divided into 64 512x512 tiles. Then only the tiles that are actually on the display need to be requested. As the user pans additional tiles can be loaded and as they zoom tiles from a different decimation level can be loaded. This concept of decimating and tiling map imagery is very common in web maps and more information can be found online. A good overview of webmap tiling is available from mapbox (https://www.mapbox.com/help/how-web-maps-work/).

Using the DAF from Python

The DAF can be used from either **Java or Python**. Since the vast majority of field development is done in Python, that will be the focus of this document.

The core of the DAF lies in the **DataAccessLayer** class, so any script that will request data via the DAF must first import this class.

```
from ufpy.dataaccess import DataAccessLayer
```

Next, a **DefaultDataRequest** object is obtained, its parameters are set, available times are obtained, and the data is requested. The following is a simple grid request of 500MB temperature from the GFS40.

```
req = DataAccessLayer.newDataRequest()
req.setDatatype('grid')
req.setParameters('T')
req.setLevels('500MB')
req.setLocationNames('GFS212')
times = DataAccessLayer.getAvailableTimes(req)
grids = DataAccessLayer.getGridData(req, times[:10])  # Get the first 10 times
```

How Does Ingested Data Get Into CAVE?

CAVE retrieves data through HTTP services provided by EDEX. By using the HTTP protocol, CAVE can either connect to EDEX over the internet or connect locally over a LAN using the same interface. To improve performance the requests and responses between CAVE and EDEX are serialized into binary using Dynamic Serialize which is a serialization protocol built on top of the Apache Thrift FOSS package. **Figure 5-2** displays the CAVE-to-EDEX interface.

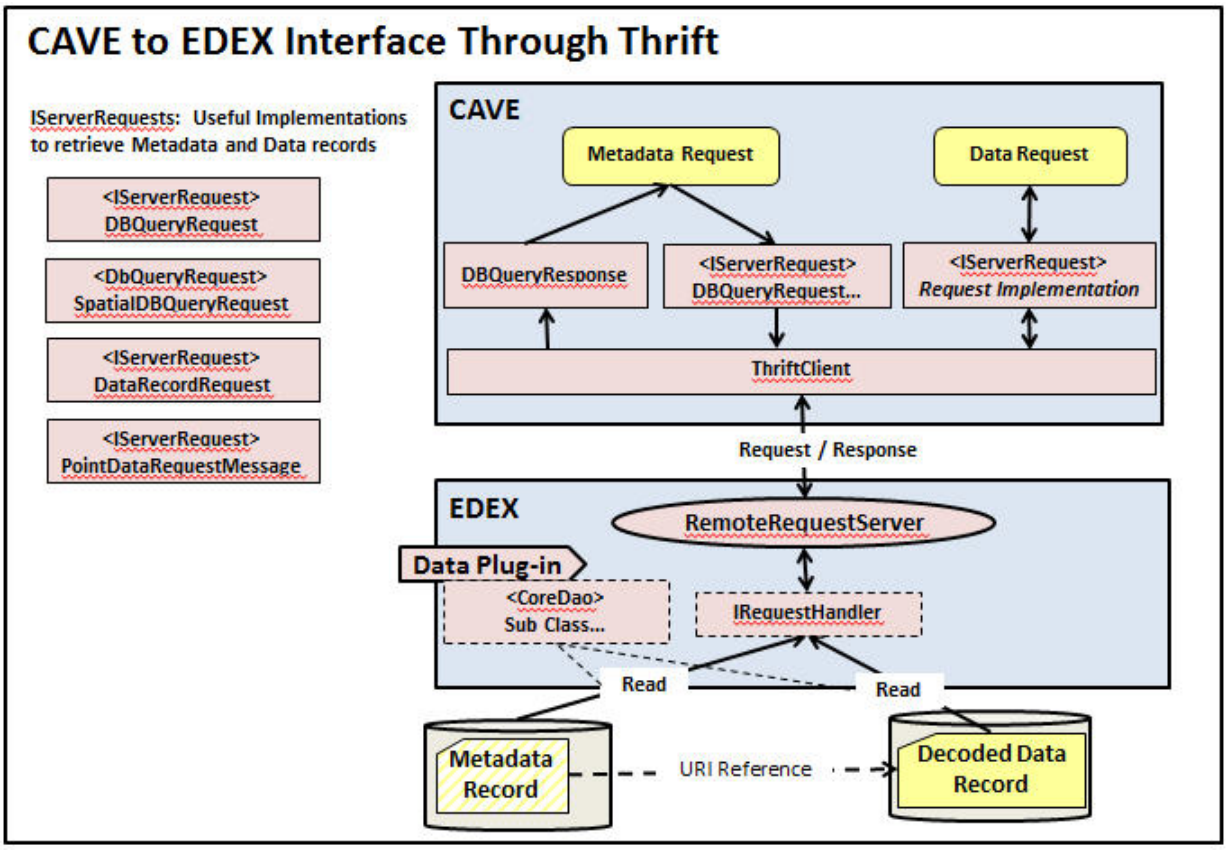

Figure 5-2. CAVE to EDEX Interface Through Thrift

- CAVE data requests can originate through user actions such as menu selections or automatically through ingest event notifications as described earlier.
- CAVE may send one or many requests to load data. In some scenarios, CAVE may first request metadata about what data is available in the system and then determine which data to request based on configurations, time matching, or other parameters. In other scenarios, CAVE may make a single request for each frame of data or a single request for all data that meets constraints.
- An implementation of the *IServerRequest* interface is used for all CAVE requests to EDEX. AWIPS II contains many implementations, a few of which are shown in Figure 5-2. All CAVE IServerRequests eventually go through the CAVE class ThriftClient to get to EDEX. ThriftClient will send the request over to the HTTP service.
- EDEX receives the HTTP requests and processes them through the RequestServiceExecutor service. The service uses *IRequestHandler* implementations to process requests. Many handler implementations exist in AWIPS II to deal with the various ways metadata and data need to be returned to CAVE. After the handler processes the request, the response is serialized and sent back to CAVE.

CAVE Graphics Tips

Keep the following tips in mind when working with graphics in CAVE:

- Dispose of anything with a dispose method.
    - This includes Wireframe Shapes, Images, Fonts, and shaded shapes.
    - Failing to dispose may leak memory or graphics memory. It is very difficult to identify leaking graphics memory but it will slow down CAVE.
    - For VizResources dispose of graphics objects in disposeInternal or whenever you no longer need an object and might be losing a reference to that object.
- Avoid creating graphics resources every time you paint.
    - Things like fonts, wireframe shapes, shaded shapes, and images should be created when they are needed and reused as long as they remain unchanged.
        - If you have a different image or shape for every time in a resource you should keep around all the different resources, you should not dispose and re-create every time the user changes frames; this can lead to very slow looping.
- Use bulk rendering whenever it is possible.
    - Most methods of IGraphicsTarget have methods that draw a single item and methods that draw a group of items all at once; using the grouped methods will increase graphics performance considerably.
    - The single draw methods are only intended to be used if you are drawing only a few things.
        - In general, if a single resource is calling **IgraphicsTarget.draw*** more than a dozen times, you should try to determine if any of the draw methods can be used in bulk instead.
        - It is almost never a good idea to be calling
        - **IgraphicsTarget.draw*** inside a loop; instead, add items to a List and call draw all at once.
    - You can combine multiple drawLine and/or drawRectangle calls by creating a wireframe shape and adding lines to that.
        - wireframe shapes are more efficient than DrawableLine objects because they are compiled to move the data onto the graphics card.
- Use I*ImageCallback objects to retrieve data for images.
    - Callbacks were designed to let the target efficiently manage memory so that large amounts of data can be loaded without running out of java heap space.
    - Read the javadoc on IColorMapDataRetrievalCallback and IRenderedImageCallback for more information
- Use **IWireframeShape.allocate** whenever possible.
    - If you have an idea how much memory you will need after you create the shape, then allocate before adding any points.
        - If you need more space than allocated, it will allocate extra memory for you without errors.
        - All unused space will be freed once compile is called.
- Use **IwireframeShape.addLabel** only when it is applicable.
    - The reason a wireframe shape contains labels is so that the lines are not drawn on top of the labels. There will be a gap in the lines where the label is. An example of this is contours.
    - If you are drawing lines and labels that are not related, it is much better to handle them separately. This would be the case for graphs and charts.
- Remember that you are not guaranteed to be the only resource drawn.
    - To know if you need more optimization, you should ensure that your resource can pan, zoom, and loop smoothly.
    - If your resource is good in a single pane, you should try it in a four panel; many unoptimized resources slow down significantly when switched to four panels.
- If in D2D, load your resource in some side panels as well. Start everything looping and make sure you can still smoothly pan and zoom.

Derived Parameters

The derived parameters framework is designed as an extendable way for calculating custom weather parameters from existing data. It can combine different weather parameters, from different layers in the atmosphere and even from different sources to calculate almost anything you might want to see. The most extensive use of derived parameters is for grid data; however, it is also used for point data.

The XML Files

Derived parameters is controlled largely by xml configuration files that contain instruction on how to derive parameters. These xml files will be in **localization/derivedParameters/definitions/*.xml**. There is one file per parameter and typically the name of the file is the same as the parameter abbreviation. Here is an example of the contents of DpD.xml

```
<DerivedParameter unit="K" name="Dew point depression" abbreviation="DpD">
   <Method name="Difference">
       <Field abbreviation="T"/>
       <Field abbreviation="DpT"/>
   </Method>
</DerivedParameter>
```

This definition defines a parameter named Dew point depression, which is known internally as DpD with units K. To calculate Dew point depression you take the difference of the T parameter and the DpT parameter, which is temperature and dewpoint.

The Python Files

The actual mathematical and logical operations that can be performed with derived parameters are completely configurable and extendable using python scripts. These python scripts will be in **localization/derivedParameters/functions/*.py**. In the previous section dew point depression was calculated using a Difference method. This method is defined in a python script **Difference.py.** For these scripts the file name is always the method name that is used in the XML.

A script must provide a function definition named execute that contains the logic for that method. The values are passed into python as numpy numeric arrays. The numpy library includes many common operations that can be useful for doing calculations quickly. It is also possible to use any features of the python language to calculate derived parameters.

Advanced XML

The DpD example of XML was very basic; there are many additional XML attributes that can be used to control how derived parameters work and where the data comes from.

The DerivedParameter element is at the root of the XML document for any derived parameter definition. All of the important attributes were given in the DpD example. The abbreviation attribute is used within derived parameters as an id for a parameter. The name attribute is something nicely formatted for display to the user. The units attributes is used for unit conversion and style rules. A DerivedParameter may contain many Method elements, when a derived parameter is requested each method is tried until one is found that is valid and for which the data type has all available fields.

For a Method element, in addition to the name you can also provide several other attributes, including the following:

- **Levels.** Limit which levels that method applies to; for example, if you specify levels="500MB" and the user requests data on the 700MB level, then that method will be skipped over. The valid values for this attribute are controlled by the LevelMappingFile.xml. Each key can serve as a value for levels, or you can provide a comma separated list of these keys. In addition to providing specific levels you can also provide a master level name; for example, levels="MB"

will apply that method to only MB levels. If no levels attribute is provided then the method can be applied to any level for which the fields are available.
- **Models.** This is simply a space-separated list of the sources for which the method is valid.
- **dtime and ftime.** These are time modifiers. Both are boolean attributes. When either of these is set to true, then fields can specify a time shift that will be used to request data from a different time than the derived parameter. This is useful for doing a parameter change over time or an accumulation over time. The difference between dTime and fTime is that fTime will only apply the time shift to forecast time so ref time must be the same for all fields and dTime will grab any data with a shifted valid time(ref time + forecast time).

There are two types of fields that a method can have:

- A **ConstantField element**, which has a single attribute, value, which is a number to use for that argument; and
- A **Field element**, which is used to guide derived parameters in selecting data to use in a method. A Field element can have several attributes:
    - **abbreviation** specifies which parameter to request for the field; this attribute is required.
    - **level** is used to specify which level load data for this field. It must be a single key from the LevelMappingFile.xml. If no level is provided then when a derived parameter is requested it will use data on whatever level is being requested, when a level is provided it uses that level instead.
    - **model** is used to import data between different sources; it must contain a single valid source.
    - **timeShift** is used to request data from a different time than the derived parameter. The time Shift is provided in seconds, a negative value will request past data and a positive value will request future data, usually only useful with forecast data.

Using localization derived parameters XML definitions can be overridden by a site or a user. When an override is provided the methods in the override are evaluated first before using the base files. The base files are not ignored; they are just lowered in priority.

All **Style Rules** are managed by the StyleManager. Each instance of the StyleManager can provide the style rules and preferences for those rules. To get the Style Rules from the StyleManager you need to provide the StyleType and the MatchCriteria. The StyleType is an enum of available types. There are several places in the code where the MatchCriteria are created. StyleRule preferences are defined in xml configuration files. Most configuration file names end in **StyleRules.xml**. For examples, search the baseline for ***StyleRules.xml.**

```
 StyleRule sr = StyleManager.getInstance().getStyleRule(StyleType.Imagery, matchCriteri
a);

ImagePreferences prefs = sr.getPreferences();
```

ArrowPreferences are used to set the scale of a GriddedVectorDisplay.

GraphPreferences are used to set label and line preferences for a graph.

ImagePreferences are used to set preferences on the CAVE display image.

In AWIPS II, gridded icon displays can be configured using derived parameters. For example, the PTyp parameter displays as icons and is defined in the **PTyp.xml** derived parameter file. Many of the definitions for this parameter use the PTyp method, which is defined in PTyp.py. Within PTyp.py the input parameters are combined and then mapped to very specific integer values. These values are what determines the symbols to display. These values map to the AWIPS II Weather symbols font, which is defined in **/awips2/cave/etc/plotModels/WxSymbols.svg.** In order to change what symbols display, all you need to do is create a site-level **PTyp.xml** that contains a new method definition to map to the symbols you want. Within the definition you can use any of the existing python functions or create a new one that maps exactly how you like it.

The Derived Parameter Tree and Inventory

In code the two most important data structures used within derived parameters are DataTree and Inventory. The DataTree is a data structure that maps a source, parameter, and level to a LevelNode. LevelNodes are the leaves of the tree and they contain the information for getting data for that source/parameter/level combination. An Inventory object holds the DataTree, and dynamically populates it with derived definitions when they are requested. The Inventory object is typically created by an IDataCubeAdapter that is initialized when data is requested for a plugin.

The AbstractInventory class is meant to provide a base Inventory implementation which plugins can extend to use derived parameters. The most important method to implement is createBaseTree. This method is used on initialization to determine what base parameters a plugin can provide as arguments to derived parameter. The leaves of this base tree should be AbstractRequestableLevelNodes that are specific to that plugin. An implementation of AbstractRequestableLevelNode will need to be capable of requesting data for a datatype. The two responsibilities of an AbstractRequestableLevelNode are to be able to time query available data and to be able to request data. The actual data request is handled by creating AbstractRequestableData objects. These objects are very similar to a PluginDataObject; they contain metadata about a specific record, the source/parameter/level and a datatime. They also contain the logic needed for requesting the raw data when it is needed.

Derived parameters are added to the DataTree in the AbstractInventory.walkTree method. This method allows you to provide a specific set of sources, parameters, and levels and they will be resolved to AbstractRequestableLevelNodes using the DataTree and derived parameter definitions. When calling walkTree you should try to be as specific as possible in what is requested to ensure it returns quickly and does not create unneeded derivations. For more information on how walk tree functions, see the java doc on that method.

Derived parameters are added to the DataTree as an instance of AbstractRequestableLevelNode. The most important such class is DerivedLevelNode, and its operation provides a basic overview of how all these nodes function. When a DerivedLevelNode is created it is supplied with other nodes that serve as the arguments to derived parameters. These other nodes can be other derived parameters or they can be the base nodes for a data type. When a request for a time query is made to a DerivedLevelNode, it first time queries each of its dependency nodes, combines these times and returns only times for which all dependencies are available. When a data request is made on a DerivedLevelNode it first requests data for all the dependency nodes, and then passes this data to the correct python script and retrieves a result record that can be returned. Because derived nodes have dependency nodes and some of those nodes can be derived nodes with their own dependencies, the whole thing forms a tree like structure, so each level node on the DataTree is its own derivation tree.

The way grid handles derived parameters is using request constraints. Whenever a time query is made to the data cube adapter, the adapter passes these constraints to the inventory which uses them to find all possible source/parameter/level options that match those constraints and uses walkTree to get all matching level nodes. The data cube adapter time queries each node and returns the result. When data records are requested a similar process is followed, to get RequestableDataObjects which are wrapped in GribRecords.

CAVE - Right-Clicking on the Legends

Each legend displayed in the bottom right corner of the editor can correspond to different maps and resources, such as county boundaries or plots. Right-clicking on individual legends enables a pop-up menu that can display different menu items to the user, such as Change Color and Line Style. Developers can add more menu items to the pop-up menu and update the capabilities of each resource.

Adding Menu Items

Menu items can be added to the pop-up menu by updating the file **com.raytheon.viz.ui/plugin.xml** (specifically, the extension for the point **com.raytheon.viz.ui.contextualMenu**). Each menu item corresponds to a contextualMenu. For example,

```
  <extension
        point="com.raytheon.viz.ui.contextualMenu">
    ...
    <contextualMenu
            actionClass="com.raytheon.viz.ui.cmenu.ChangeColorAction"
            capabilityClass="com.raytheon.uf.viz.core.rsc.capabilities.ColorableCapabili
ty"
            name="Change Color"
            sortID="10"/>
    ...
</extension>
```

Notice the attribute actionClass. The value for this attribute should point to a child class that extends the abstract class AbstractRightClickAction. In AbstractRightClickAction, the key methods that need to be overwritten are getText and run. The method getText returns the actual text that will be displayed in the pop-up menu. The method run executes when the menu option is selected. Refer to ChangeColorAction for an example.

Another important attribute is the capabilityClass. The value of the capabilityClass points a child class that extends the abstract class AbstractCapability. In the above example, the attribute is set to **com.raytheon.uf.viz.core.rsc.capabilities.ColorableCapability.** Refer to this class for an example. Setting the capabilityClass allows the capability to be available when referenced by the action class when a getCapability is called.

CAVE - Right-Clicking In Editor

Right-Clicking in the editor enables a pop-up menu that can offer different options to the user, such as Show Product Legends, Sample, Zoom, and Lat/Lon Readout. It is important to note that the pop-up menu will appear when the right mouse button is held down. A simple click will toggle the first menu item, that is, if the mouse click functionality has not been overwritten, such as in WarnGen. It might be beneficial to add more menu items common to the user in the pop-menu. This addition can make work for the user convenient and timely.

If a developer wants to add more items, there are two important classes that are used:

1. **com.raytheon.viz.ui.cmenu.AbstractRightClickAction**; and
2. **com.raytheon.uf.viz.d2d.ui.perspectives.D2DPerspectiveManager**.

The AbstractRightClickAction object corresponds to each individual menu item while D2DPerspectiveManager manages which AbstractRightClickAction objects to add to the pop-up menu.

AbstractRightClickAction

For each menu option, an action class needs to be created that extends the abstract class AbstractRightClickAction. The two key methods that need to be overwritten are getText and run. The method getText returns the actual text that will be displayed in the pop-up menu. The method run executes when the menu option is selected. Refer to LatLonReadoutAction for an example.

D2DPerspectiveManager

In the D2DPerspectiveManager, child classes of the AbstractRightClickAction are created. However, it is the method addContextMenuItems that actually determines which AbstractRightClickAction to add to the menu. For example, as seen with the following, legend modes can be used to determine what kind of ChangeLegendModeAction to add.

```
 @Override
public void addContextMenuItems(IMenuManager menuManager,
        IDisplayPaneContainer container, IDisplayPane pane) {
    ...
    D2DLegendResource ld = null;
    ...
    if (container instanceof SideView == false) {
        LegendMode mode = null;
        if (ld != null) {
            mode = ld.getLegendMode();
            if (mode != null) {
                switch (mode) {
                    case NONE: {
                        menuManager.add(getLegendAction(LegendMode.PRODUCT, ld));
                        menuManager.add(getLegendAction(LegendMode.MAP, ld));
                        break;
                    }
                    case PRODUCT: {
                        menuManager.add(getLegendAction(LegendMode.HIDE, ld));
                        menuManager.add(getLegendAction(LegendMode.MAP, ld));
                        break;
                    }
                    case MAP: {
                        menuManager.add(getLegendAction(LegendMode.HIDE, ld));
                        menuManager.add(getLegendAction(LegendMode.PRODUCT, ld));
                        break;
                    }
                }
            }
            menuManager.add(sep);
        }
    }
    ...
}
```

CAVE Maps

Importing Shapefiles

Local shapefiles can be imported into the maps database using the automation tool. Files should be staged in the following location where LLL is the WFO, e.g., OAX:

```
/awips2/edex/data/utility/edex_static/site/LLL/shapefiles
```

The shapefiles should be added in a manner similar to the following:

```
shape_desc/shapefile.(dbf|shp|shx)
```

The directory name of *shape_desc* above will determine the table name into which the shapefiles will be imported. For example, the following shapefiles will create **mapdata.oax_county** schema in the maps database:

```
/awips2/edex/data/utility/edex_static/site/OAX/shapefiles/OAX_County/OAX_County.dbf
```

```
/awips2/edex/data/utility/edex_static/site/OAX/shapefiles/OAX_County/OAX_County.shp
```

```
/awips2/edex/data/utility/edex_static/site/OAX/shapefiles/OAX_County/OAX_County.shx
```

To import the above shapefiles staged into the database, run the following:

```
./config_awips2.sh shp OAX
```

This option will also call the config_ffmp_shapefiles script to load the FFMP shapefiles.

For more details see https://collaborate.nws.noaa.gov/trac/siteconfig/wiki/ADAM (https://collaborate.nws.noaa.gov/trac/siteconfig/wiki/ADAM).

How to Query Maps Database

- **com.raytheon.uf.common.geospatial.SpatialQueryFactory.** Use this class' static method create() to obtain an instance of a class that implements IspatialQuery.
- **com.raytheon.uf.common.geospatial.ISpatialQuery.** This interface contains many overloaded query(...) methods that all return an array of SpatialQueryResult[]. It also contains a dbRequest(String sql, String dbname) that can be used to execute more general sql queries.
- **com.raytheon.uf.common.geospatial.SpatialQueryResult.** This is a data class that contains an instance of **com.vividsolutiions.jts.geom.Geometry** and a mapping of its attributes.
- **com.raytheon.edex.plugin.warning.gis.GeospatialDataGenerator** The static method queryTimeZones(...) is one example of using the above classes (see **Figure 4-4**). It performs a query to get timezone information (lines 449-452) and then modifies the attributes to contain the information.

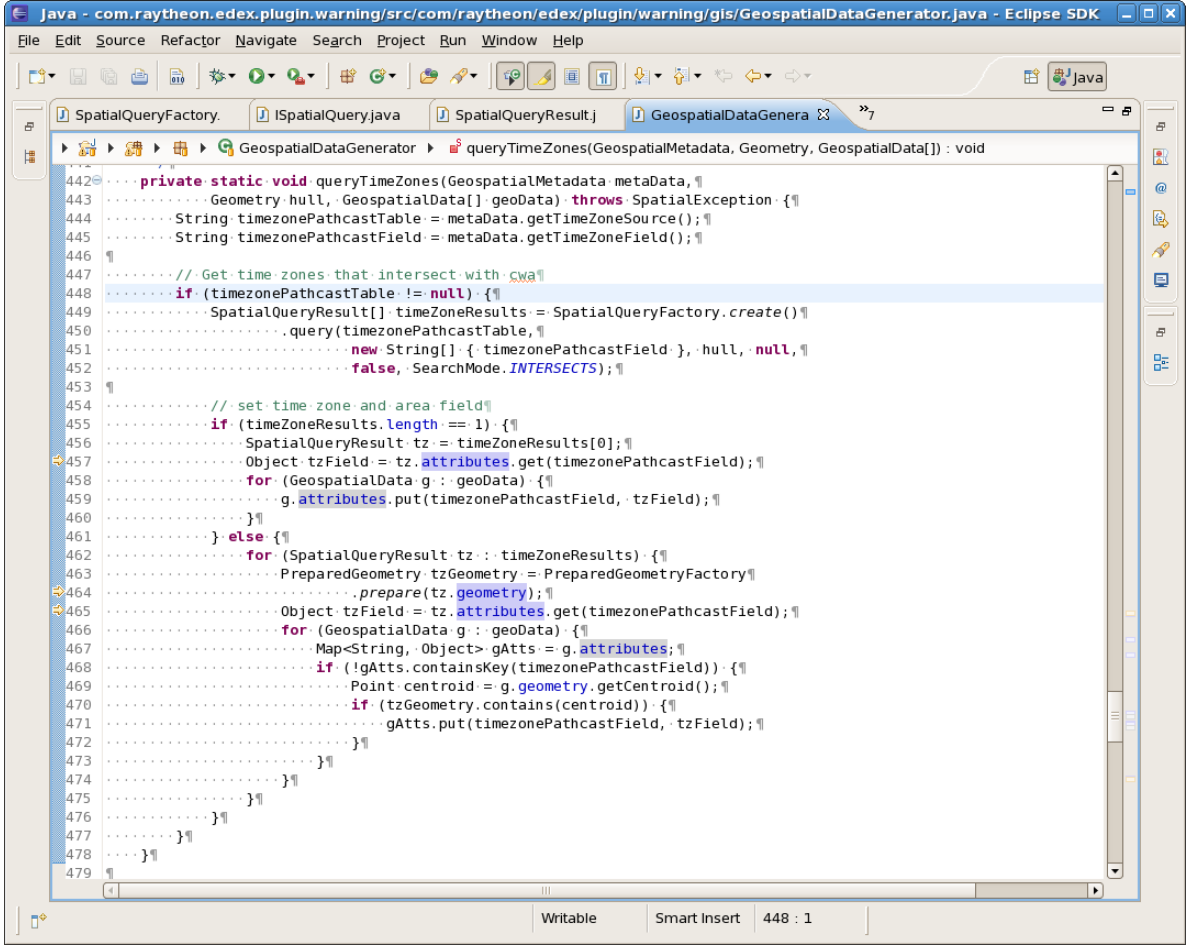

Figure 4-4. Geospatial Data Generator

CAVE Features

CAVE uses features projects much differently than EDEX and resembles how Eclipse RCP intended them to be used. CAVE, includes a feature project for groups of similar plugins; this is one of the first projects that should be created when developing new plugins. An example of this includes **com.raytheon.uf.viz.d2d.core.feature**, which is the feature project for all core D2D projects. Another example is **com.raytheon.uf.viz.cots.feature,** which is the feature project for COTS projects in CAVE. Feature projects contain two files, **build.properties** and **feature.xml**. The build.properties file just states what should be included in a binary build and what should be included in a source build. It will always have the **feature.xml** file listed. The **feature.xml** file contains a list of plugins that should be included in the feature as well as a list of dependencies on other features that this feature has. This dependency list should not only include features it will directly depend on but also the features that its dependencies depend on and so on. Because this method of using feature projects is only meant for CAVE, only viz and common plugins should be added to features. If an EDEX plugin needs to be added, more than likely the needed code will need to be moved to the common project.

Creating

In most cases, a developer of plugins will want to create a feature project for their plugins. To do this, from Eclipse, select **File->New->Project...** Select the "Plug-in Development" folder and choose "Feature Project." The "New Feature" wizard will open and a project name will need to be entered. The naming convention usually looks something like:

```
<creating_entity_url>.uf.viz.<component_name>.feature
```

Example: **com.raytheon.uf.viz.thinclient.feature**

In this example the creating_entity_url is "com.raytheon" and the component_name is "thinclient." Once a feature name has been entered, change "Feature Name" to "<component_name> Feature" and select "Finish" at the bottom of the wizard. Now that the feature project is created, open the **feature.xml** file and it should open in the Eclipse graphical editor's "Overview" tab. Switch to the "Plug-ins" tab; it is here that the plugins the feature was created for can be added. Next, switch to the "Dependencies" tab and add the features that are required. The most common features that all other features depend on are:

- **com.raytheon.uf.viz.eclipse.feature**
- **com.raytheon.uf.viz.cots.feature**
- **com.raytheon.uf.viz.common.core.feature**
- **com.raytheon.uf.viz.core.feature**

The developer needs to determine the full list of plugins that are depended on and not in these common features or the feature being created. It is not just the plugins that are directly depended on that must be gathered; the dependencies of those plugins also must be determined, and so on until an entire hierarchical dependency tree can be seen. At this point the features that the dependency plugins are in will need to be added as dependencies to the developer's feature.

Modifying

There may be cases when a new plugin(s) needs to be added to an existing feature. Do this rarely, and with extreme caution. It is important to begin by getting a list of the new plugin's dependencies because you must be sure not to add dependencies to the feature project that will cause a cycle. Once you have a list of dependencies for your new plugin, trace through the entire dependency tree and get a list of the features your plugin depends on. Then, check to see if your plugin depends on any features that are not currently dependencies of the feature you want to add to. If there are no additional dependencies, you may add the plugin. If there are additional dependencies, proceed only with high caution. You should probably look into adding a ***new*** feature project for your plugin, but if you still wish to add the plugin to the feature project, you need to

build a complete dependency tree of the additional feature(s) your plugin depends on. If none of the dependencies in the tree can be linked back to the feature you want to add to, you may add the plugin. Otherwise, you must create a separate feature for your plugin.

Building/Deploying

Once a feature project has been created, it must be set up to be used when running from Eclipse and built for distribution. To run CAVE out of Eclipse with the feature enabled, open the **feature.xml** file in the project **com.raytheon.viz.feature.awips. developer** and switch to the "Included Features" tab. Here the new feature project can be added as an included feature and the plugins referenced in it will be used next time a "Synchronize/Run" from the **developer.product** file is done. In order to build a feature/group of features manually to be deployed, an Eclipse Update Site project must be created. In Eclipse, go to **File->New->Project...**, and select "Plug-in Development/Update Site Project." Give the update site project a name like:

```
<creating_entity_url>.uf.viz.<component_name>.site
```

Select "Finish" and an Eclipse project should be created with a single file, **site.xml**. Open **site.xml** and select the "Site Map" tab. Add the feature project(s) that should be built/deployed by the site by selecting "Add Feature..." Note that only the feature projects created by the developer should be added. Also note that now that a feature project has been added, more options appear in the **site.xml** editor. Once added, the features can be built for deploy at any time by selecting the "Synchronize..." button, then the "Build All" button. Once the build is finished, there will be more folders/files in the plugin. The contents of the update site plugin can now be zipped up or copied directly to a remote server directory to be installed to CAVE via the p2 director.

CAVE Alert Observer

This is a discussion of how to be notified when an alert has arrived. This can be used to trigger getting new data in order to update a GUI's display. This is handled by using the ProductAlertObsever static methods to add and remove classes that implement the IAlertObserver interface.

com.raytheon.viz.alerts.observers.ProductAlertObserver is the class with static methods for adding and removing observer listeners that implement the IAlertObserver interface. The two static methods are:

- `addObserver(String pluginName, IAlertObserver obs)`
- `removeObserver(String pluginName, IAlertObserver obs)`

The ProductAlertObserver handles multiple observer lists based on the pluginName.

com.raytheon.viz.alerts.IAlertObserver is the interface an observer class must implement in order to register with ProductAlertObserver. It contains a single method:

- `alertArrived(Collection<AlertMessage> alertMessages)`

The implementing class can iterate though the alertMessages to determine what action it must perform.
com.raytheon.uf.viz.cored.AlertMessage contains the alert's data URI (dataURI) and a mapping of the decoded String (decodedAlerts).

Example. The example in **Figure 4-1** shows how AvnFPS updates the viewer tab that contains the Global Forecast System (GFS) Model Output Statistics (MOS) Guidance information for a site. The tab name (MAV) is configurable, so it may change.

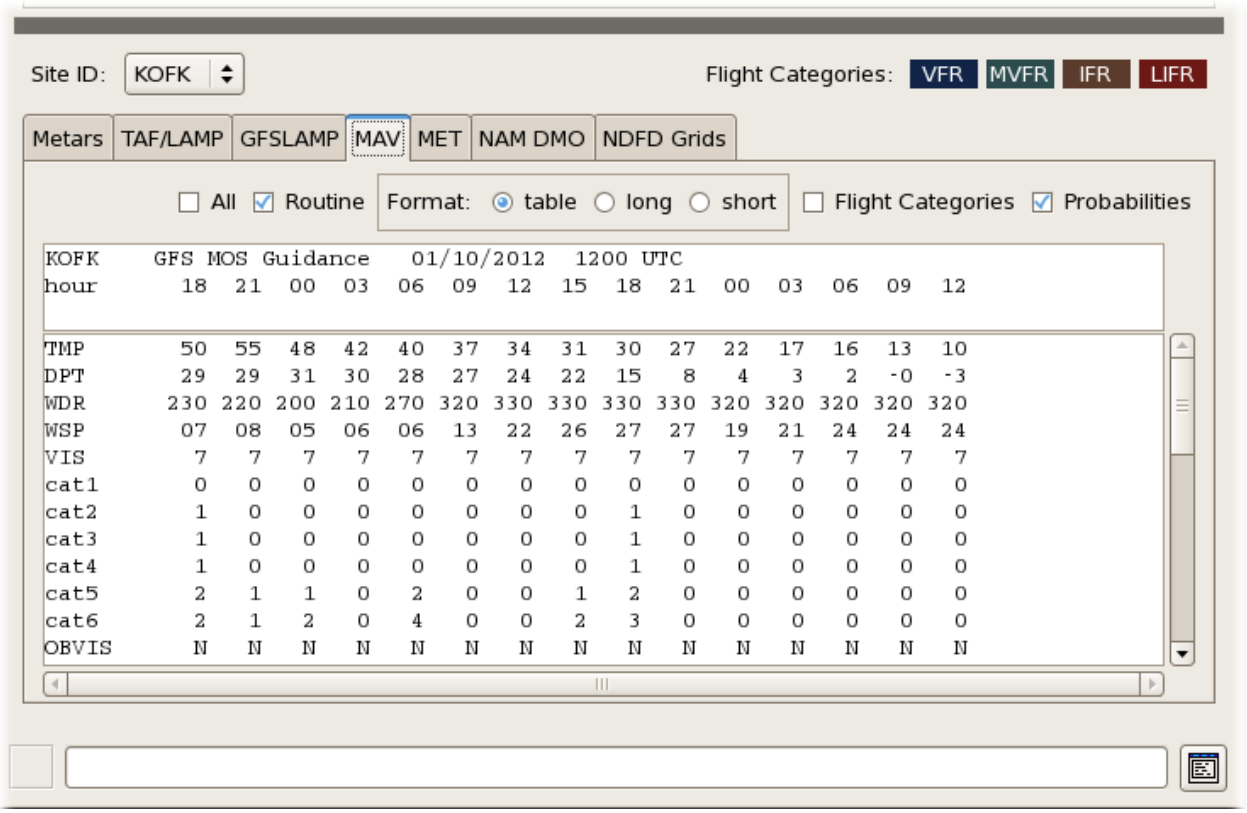

Figure 4-1. Example: CAVE Alert Observer

Determining the pluginName to use in order to register an observer can be tricky. You need to determine what DAO was used to obtain the data and look up its bean information in the appropriate xml file. For our example, this is done by the BuferMosGFSData DAO. Looking at the spring configuration file **bufrmos-common.xml** (see **Figure 4-2**), find the following bean definition, which has the pluginName bufrmosGFS.

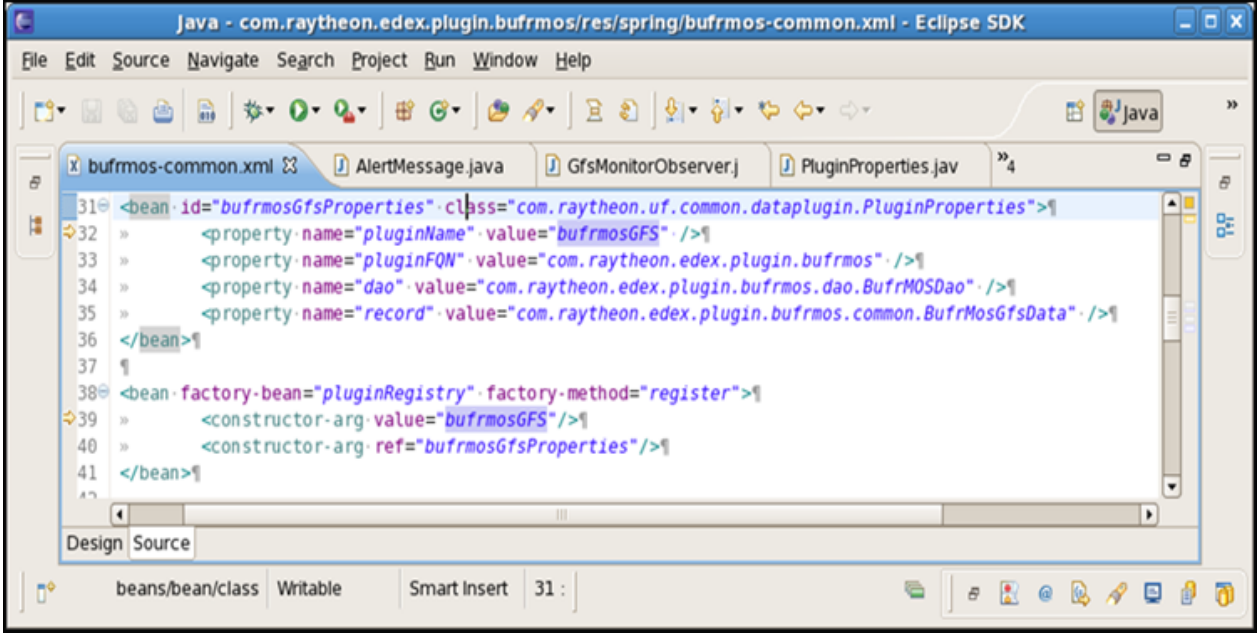

Figure 4-2. Spring Configuration File (bufrmos-common.xml)

com.raytheon.viz.aviation.monitor.GfsMonitorObserver is the class that implements the IAlertObserver interface for updating the tab. It contains a static element pluginName that is set to "bufrmosGFS". The alertArrived method determines which site, if any, of the sites it can display needs to have its cache data updated. The currently selected site's display is also updated. See **Figure 4-3** for an illustration.

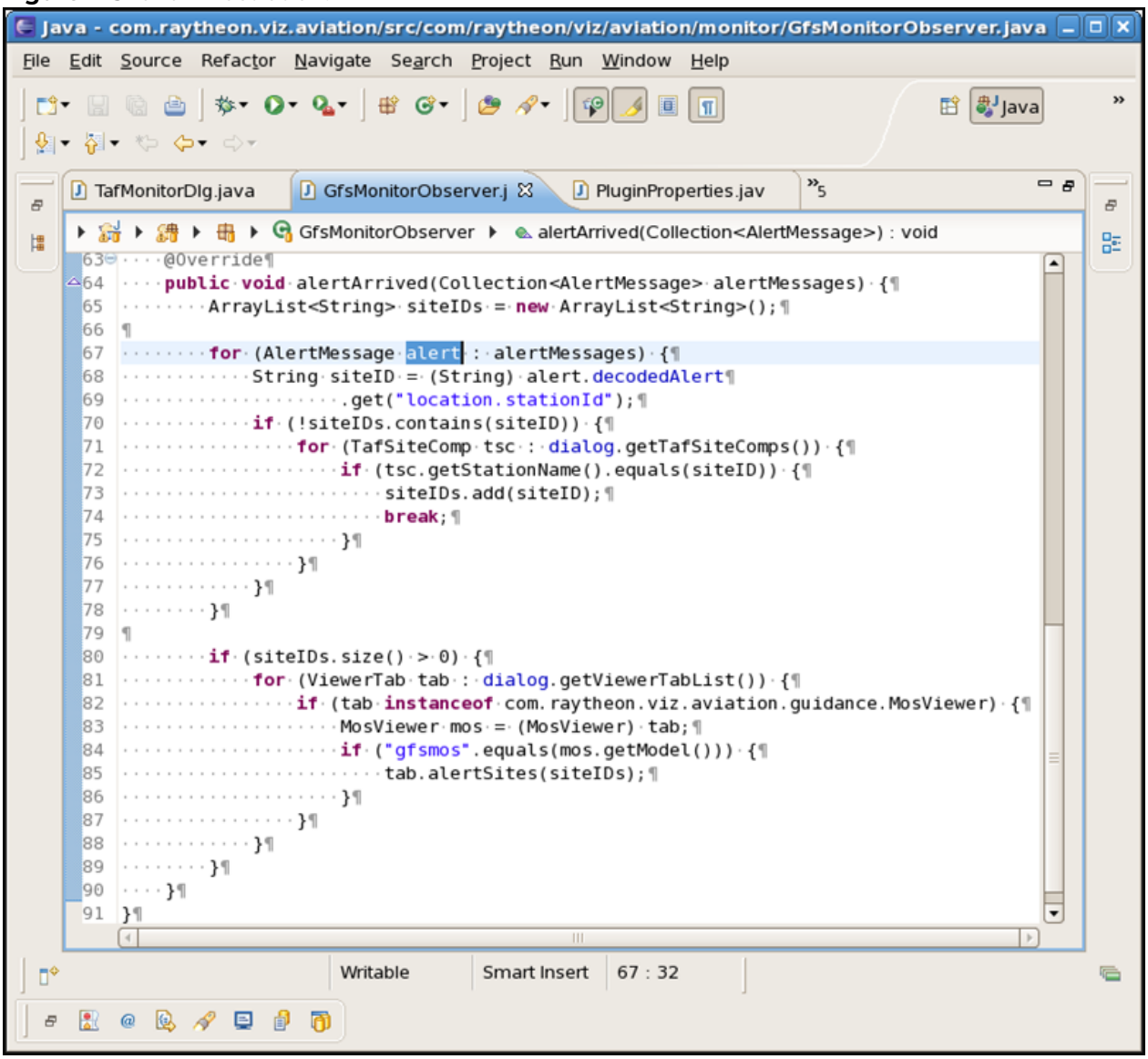

Figure 4-3. GFS Monitor Observer

com.raytheon.viz.aviation.observer.TafMonitorDlg is that class that controls the dialog display. Its setupMonitoring method registers the observer:

```
  gfsObserver = new GfsMonitorObserver(this);
ProductAlertObserver.addObserver(GfsMonitorObserver.pluginName, gfsObserver);
```

Its cleanupMonitoring method does the unregister:

```
  ProductAlertObserver.removeObserver(GfsMonitorObserver.pluginName, gfsObserver);
```

Menu Customization

index.xml

CAVE automatically searches for **index.xml** files in localization under **menus/*.** This allows a developer to add a menu simply by adding an **index.xml** menu under this location using the Localization perspective and having CAVE pick it up on the next restart.

Using the Localization Perspective

In CAVE, select the "Open Perspective" button -> Localization. This is both the preferred and the easiest method of editing menus under CAVE -> Menus.

Command Menu Items

- **xsi:type** = "command"
- **commandId** = the command that was defined in the plugin.xml
- **menuText** = the text to be seen in the menu
- **id** = a unique id that describes the menu item

Example:

```
<contribute xsi:type="command"
       commandId="com.raytheon.uf.viz.radarapps.rps.rpsListEditor"
       menuText="RPS List Editor..." id="${icao}RPSListEditor" />
```

Bundle Menu Items

- **xsi:type** = "bundleItem"
- **file** = name of the bundle in localization to load
- **menuText** = the text to be seen in the menu
- **id** = a unique id that describes the menu item

Example:

```
<contribute xsi:type="bundleItem" file="bundles/DefaultRadar.xml"
       menuText="0.5 Z" id="${icao}058bitZ">
   <substitute key="icao" value="${icao}"/>
   <substitute key="product" value="94"/>
   <substitute key="elevation" value="0.5--0.5"/>
</contribute>
```

Title Menu Items

- **xsi:type** = "titleItem"
- **titleText** = the text to be seen in the menu
- **id** = a unique id that describes the menu item

Example:

```
<contribute xsi:type="titleItem" titleText="------ Applications ------"
       id="${icao}Applications" />
```

Separators

- **xsi:type** = "separator"
- **id** = a unique id that describes the menu item

Example:

```
<contribute xsi:type="separator" id="${icao}applicationsSeparator"/>
```

Submenus

- **xsi:type** = "subMenu"
- **menuText** = the text to be seen in the menu

This surrounds the types that you want to go inside that submenu.

Example:

```
<contribute xsi:type="subMenu" menuText="${icao}">
```

Including Other Menu Files

Other files can be included within menus, and can either be whole submenus or just in the same menu.

- **xsi:type** ="subinclude"
- **submenu** = name of the sub-menu
- **fileName** = path of the file in localization

Example:

```
  <contribute xsi:type="subinclude" fileName="menus/radar/baseReflectivityMotion.xml" />
<contribute xsi:type="subinclude" subMenu="${icao} four panel"
        fileName="menus/radar/baseRadar4Panel.xml" />
```

Variable Substitution

Variable substitution allows for a single variable to be substituted across all levels inside the xml files. For instance :

```
<substitute value="koax" key="icao"/>
```

Any time that "${icao}" is used from this point on in the calling for xml files, "koax" will then be substituted and used for the value. Certain plugins generate a single file (**index.xml**) and have all the substitutions in there allowing for dynamic values in the menus.

Automatically Customized Menus

- **Radar.** Changing the radarsInUse.txt file will regenerate menus on next CAVE restart. This file has sections for each type of radar (local, dial, Aggregation Service Routers (ASR), Air Route Surveillance Radar (ARSR), terminal). This will change what shows up in the top menus as well as under Radar -> Dial Radars.
- **Satellite.** Based on the site that CAVE is localized to, the satellite menus will change to reflect East CONUS vs. West CONUS or non-CONUS.
- **Upper Air.** Very similar to the radar menus, this is configured based on the raobSitesInUse.txt file.

How to Write Dialogs for CAVE Classes

This is a guide to creating a CAVE dialog that does not block the User Interface (UI) thread. When a dialog is open that blocks the UI thread alerts, other critical information will not be displayed in a timely manner. Only the main dialog, CAVE, or the top dialog of a standalone product should be blocking.

Problems with Blocking Dialogs

When a blocking dialog's open() is performed, a return does not happen until the dialog is disposed. This makes it easy to perform the logic for any results returned by the open(). The problem is that, apart from this dialog, the main active dialog (CAVE or a standalone dialog) must be a blocking dialog. The UI thread has problems handling more than one blocking dialog. Popping-up dialogs, such as an Alert, are queued up by the UI thread to be opened after the non-main blocking dialog's open() returns. Thus, a forecaster will not see or hear an alert until after the blocking dialog is disposed. Having the blocking dialog minimized does not help.

To get around this problem, CAVE has the classes CaveSWTDialog and CaveJFACEDialog, which can be extended to make non-blocking dialogs. These two classes are explained in the following sections Converting to CAVESWTDialog and Converting to CaveJFACEDDialog.

Finally, a blocking dialog may not be modal, and a modal dialog does not have to be blocking. A modal dialog prevents its parent from being changed while it is open. Normally, with this behavior, a modal dialog also blocks because nothing much can be done while it is open. With CaveSWTDialog, a dialog can be made modal and non-blocking, preventing the problems associated with a blocking dialog but still having the behavior of a modal dialog.

Converting to CAVESWTDialog

Eclipse contains two useful dialog classes; both are named Dialog. To aid in converting dialogs that extend these classes, CAVE has two classes. If you are converting a class that extends the Dialog in the **org.eclipse.jface.dialogs**, see the following section Converting to CaveJFACEDDialog.

Guidance on converting a dialog that extends the Dialog in the **org.eclipse.swt.widgets** package to a CaveSWTDialog follows.

- Verify the MANIFEST.MF for the imports **com.raytheon.viz.ui**
- Change the class declaration for example:

  ```
  public class TheDialog extends Dialog  … {
  ```

  to

  ```
  public class TheDialog extends CaveSWTDialog … {
  ```

- Look for a class variable such as `Shell shell` and remove it because it will mask the variable set up by CaveSWTDialog.
- Look for the open() method:

  ```
  public Object open() {
  ```

  and convert to:

  ```
  @Override
  protected void initializeComponents(Shell shell) {
  ```

  - This is a method called by CaveSWTDialog to generate the dialog when its open() is called the first time.
    - In this method you may find:

      ```
      Shell parent = getParent();
      display = parent.getDisplay();
      ```

        - Most likely there is no need for the parent shell. You can get the display from the shell:

```
display = shell.getDisplay();
```

    - The old open() will contain lines such as the following, which need to be removed:

```
 shell = new Shell(parent, SWT.DIALOG_TRIM | SWT.PRIMARY_MODAL);
shell.setText("The Dialog");
```

    - The SWT constant argument should be moved to the second argument of the constructor's super and the text can also be set there:

```
public TheDialog(Shell parent, ...){
   super(parent, SWT.DIALOG_TRIM | SWT.PRIMARY_MODAL, CAVE.NONE);
   setText("The Dialog");
```

        - Notice that the third argument on the super call is for CAVE's dialog information. For now it has the placeholder CAVE.NONE, which leaves the dialog in its default blocking mode. This will be changed later.
    - What normally follows in the old open() is the code to set up the dialog's display. For the blocking dialog there will be something like:

```
 shell.pack();
shell.open();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch()) {
        display.sleep();
    }
}
```

        - This is the blocking loop from the old open(). This code needs to be removed.
        - Any code after this loop handles cleanup such as disposing of fonts preparing any return values. This should be handled by overriding the disposed() method and placing the code there.
    - If the old open() returns a non-null value, use the setReturnValue(returnValue) method to provide the value it should return when disposed. See Get Results from a Dialog Using the ICloseCallback Interface, for more details.
- If you have buttons that need to close the dialog use the close() method. This will eventually call the disposed() method.
- At this point you should have a CaveSWTDialog derived dialog that is blocking and works the same as the old Dialog. This would be a good time to test it in all the places it is called prior to making it a non-blocking dialog.
- There are many ways in which a dialog can be created and opened. This will follow a typical example. Others should be similar. Assume we have a method in the parent for handling the dialog and the old method has the following pattern:

```
private void handleTheDialog() {
   TheDialog theDialog = new TheDialog(shell, …);
   theDialog.open();
}
```

- Here, no results of the dialog are used. Because the dialog is currently blocking, the open() does not return. If the dialog is non-modal, it is possible with this example to display more than one instance of the dialog. To prevent this, you may see code such as the following:

```
  private TheDialog theDialog = null;  // A class variable

…

private void handleTheDialog(){
    if (theDialog != null){
        theDialog = new TheDialog(shell, …);
        theDialog.open();
        theDialog = null;
    }
}
```

    - This will display only the one instance of the dialog when it is blocking. However, when the dialog becomes non-blocking, the open() immediately returns allowing multiple instances of the dialog.
- Look for shell listeners that clean things up when the shell is closed. Instead override the disposed() method and do the cleanup there. Most likely you will then be able to get rid of the listener:

```
  shell.addShellListener(new Shell Adapter()){
     @Override
     public void shellClosed(ShellEvent event){
         // Code here move to the disposed() method.
         …

     }
});
```

- With a CaveSWTDialog derived dialog the open() handles re-creating a dialog once it is disposed. If it is currently open or hidden, it will bring the dialog to the top. Thus only one instance of the dialog needs to be created:

```
  private TheDialog theDialog = null;

…

private void handleTheDialog {
    if (theDialog != null || theDialog.isDisposed()){
        theDialog = new TheDialog(shell, …);
        theDialog.open();
    } else {
        theDialog.bringToTop();
    }
}
```

    - This allows a single instance of the dialog to be created and forces creation of a new dialog with everything within the dialog set to its initial state.
    - To make the dialog non-blocking go to its constructor, change the **CAVE.NONE** to **CAVE.DO_NOT_BLOCK**.
    - **Note**: The Dialog may obtain information from the parent dialog used by its constructor or initialization methods. For example, a spell checker may get information from a StyledText area in the parent dialog. The text may change between opening of the dialog so changes will need to be done prior to displaying the dialog. This can be done by overriding this method:

```
  @Override
protected void preOpened() {
    super.preOpened(); // Must do this
    // Any setup prior to dialog display may go here.
}
```

        - When the dialog open() is called, this method is always invoked just prior to displaying the dialog.
    - See Get Results from a Dialog Using the ICloseCallback Interface, for guidance on how to obtain and use the results from a dialog.

Converting to CaveJFACEDDialog

Eclipse contains two useful dialog classes both named Dialog. To aid in converting dialogs that extend these classes, Cave has two classes. If you are converting a class that extends the Dialog in the org.eclipse.swt.widgets see the section on Converting to CAVESWTDialog (EConverting_to_CAVESWTDialog).

Guidance on converting a dialog that extends the Dialog in the **org.eclipse.jface.dialogs** package to a CaveJDACEDialog follows.

Normally this type a dialog is modal blocking and expects results to be returned. Blocking open() returns an integer result of Window.OK if something is to be performed with the result. To perform the conversion, take the following steps.

- Verify the MANIFEST.MF for the plugin imports **com.raytheon.viz.ui**
- Change the class to extend CaveJFACEDDialog instead of Dialog.
- You should now have a blocking dialog that replaces the old dialog.
- Now look at the constructor and look for the following pattern:

```
  public TheJfaceDialog(Shell, shell,  Object resultsObject, …) {
     super(shell);
     …
     this.resultsObject = resultsObject;
     …
}
```

To simplify your return callback, create a getter method for the resultsObject.

- Like the CaveSWTDialog, the open() method will open the window or, if it is already displayed, bring it to the top. In the parent where the dialog is used, you can make a blocking version using the following code pattern:

```
  ...
private TheJfaceDialog theJfaceDialog = null;  // class instance variable
…
private void handleTheJfacedDialog {
    if (theJfaceDialog = null) {
        ResultObject resultObject = new ResultObject(...);
        theJfaceDialog = new JfaceDialog(getShell(), resultObject,...);
        theJfaceDialog.setBlockingOnOpen(true);
    }

    int state = theJfaceDialog.open();
    if (state = Window.OK) {
        // get the result and do the update here.
        ResultObject result = theJfaceDialog.getResultObject();
        ….
    }
```

- Converting this to non-blocking is covered in the next section (Get Results from a Dialog Using the ICloseCallback Interface).

Get Results from a Dialog Using the ICloseCallback Interface

Both the CaveSWTDialog and the CaveJFACEDialog use the ICloseCallback interface to obtain values from a non-blocking dialog after it is closed. For both types of dialogs there is a setCloseCallback method. When a non-null interface is passed in via this method it will be called when the dialog is closed. The interface must implement a single method:

```
public void dialogClosed(Object returnValue) {…}
```

With a dialog that extends CaveJFACEDialog the returnValue will be an instance of type Integer. With the CaveSWTDialog, the returnValue will be whatever was passed to the last call of setReturnValue(object). If never called, it will return a null.

The following takes the blocking example in the CaveJFACEDialog and converts it to a non-blocking dialog. CaveSWTDialog is handled in a similar manner where you test to see if the returnValue is an instance of what is needed:

```
 ...
private TheJfaceDialog theJfaceDialog = null;  // class instance variable
…
private void handleTheJfacedDialog {
    if (theJfaceDialog = null || theJfaceDialog.isDisposed()) {
        ResultObject resultObject = new ResultObject(...);
        theJfaceDialog = new JfaceDialog(getShell(), resultObject,...);
        theJfaceDialog.setBlockingOnOpen(false);
        theJfaceDialog.setCloseCallback(new IcloseCallback() {
            @Override
            pubic void dialogClosed(Object returnValue) {
                if (returnValue instanceof Integer) {
                    int value = (Integer) returnValue;
                    if (value = Window.OK) {
                        // Perform the update here
                        ResultObject resultObject = theJfaceDialog.getResultObject();
                        …
                    }
                }
                theJfaceDialog = null;
            }
        });
        theJfaceDialog.open()
    } else {
        theJfaceDialog.bringToTop();
    }
}
```

- **Note:** The setBlockingOnOpen(false) forces the open() to be non-blocking. This is only for dialogs that extend CaveJFACEDialog.
- The line theJfaceDialog = null is optional for dialogs with complex setups. It may be easier to set it to null to force a new instance of it to be created the next time it is needed.
- With a CaveSWTDialog type dialog, the line if (returnValueinstanceof Integer){ can check for the expected return type object. As long as the setReturnValue is only called when the dialog has something to return, the result will be null when the dialog is canceled and nothing will be performed by the callback.

Making a Non-blocking Dialog a Standalone Blocking Dialog

Once a dialog is converted to non-blocking, it will work great in CAVE. Take the TextWorkstationDlg for example. **Its constructor is:**

```
public TextWorkstationDlg(Shell parent) {
   super(parent, SWT.DIALOG_TRIM | SWT.MIN | SWT.RESIZE,
           CAVE.PERSPECTIVE_INDEPENDENT | CAVE.INDEPENDENT_SHELL
           | CAVE.DO_NOT_BLOCK);
   setText("Text Workstation");
   …
}
```

Notice the additional CAVE constants in the super's third argument, which generate an independent dialog with proper window trimmings for a full-blown window dialog. The only change was to add:

```
| CAVE.DO_NOT_BLOCK
```

This works great when running in CAVE. Now take a look at TextWorkstationComponent where it is set up to run as a standalone component:

```
public class TextWorkstationComponent extends AbstractCAVEComponent {
   …
   @Override
   protected void startInternal(String componentName) throws Exception {
       SerializationUtil.getJaxbContext();
       TextWorkstationDlg textWorkstationDlg = new TextWorkstationDlg(
               new Shell(Display.getCurrent()));
               textWorkstationDlg.open();
   }
   …
```

When this is run by the plugin, the open() no longer blocks so the method returns right away. This results in the dialog flashing on the screen and then the program exits. To prevent this from happening, a new class was created to extend AbstractCAVEDialogComponent. It contains an additional method to perform the blocking. So, the above will work with the following changes:

```
public class TextWorkstationComponent extends AbstractCAVEDialogComponent {
   …
   @Override
   protected void startInternal(String componentName) throws Exception {
       SerializationUtil.getJaxbContext();
       TextWorkstationDlg textWorkstationDlg = new TextWorkstationDlg(
               new Shell(Display.getCurrent()));
       textWorkstationDlg.open();
       blockUntilClosed(textWorkstationDlg);
   }
   …
```

SWT

SWT uses the native widgets of the operating system. The life-cycle of the widgets' Java object mirrors the life-cycle of the native widget that it represents. When the Java widget is created, the native widget is created, and when the Java widget is destroyed, the native widget is destroyed. The design avoids the issues of calling methods on a code object before the underlying widget has been created.

Display Object

The Display object is the connection between the application SWT classes and the underlying windowing system. The Display class is windowing-system dependent and may have additional methods available on some platforms.

- Each application will have only one Display object.
- The "User-Interface" thread that creates the Display object is the thread that executes the event loop.
- An important task of the Display class is the event-handling mechanism.
- The Display class keeps a collection of the registered events from the operating system level event queue and delivers the events to the registered listener.
- The Display object forms the GUI foundation but doesn't display any graphics to the screen.

Shell Object

The shell object represents a window/dialog. A shell can be either a top level shell or regular dialog shell. The Shell follows the SWT pattern of passing in a parent and style into the constructor.

Top-Level Shell

A top-level shell:

- Takes a Display object as the parent.
- Will show up as a separate application on the operating system's task bar.
- Can be minimized to the operating system's task bar.

Regular/Dialog Shell

A regular/dialog shell:

- Takes another shell as the parent.
- Will not show up as a separate application in the operating systems task bar.
- Will be minimized when the parent dialog is minimized.
- Can be set up to block the parent dialog.

Disposing of Widgets/Objects

SWT works directly with the native graphics resources. Each SWT resource consumes a GUI resource. Because all GUI resources are limited across all platforms, a timely release of resources are vital. SWT widgets have to be disposed of manually because the Java garbage collector never guarantees a timely release so it is considered to be a poor manager of GUI resources.

When widgets are created, a parent widget is passed into the constructor (example parent widget would be a Shell, Composite, or Group). The lifetime of the parent component constrains the lifetime of the child component. So when that parent is disposed of the child get disposed of as well.

These are the rules for disposing widgets:

- If you create, you dispose it.
    - Because native resources are created when an SWT object is created, the object needs to be disposed when it is no longer used.
- If you do not call the constructor to get a resource, then you **must not** dispose of the resource.

    - Why? Because the resource does not belong to you. It is considered "borrowed."
- Disposing of the parent will dispose the child.
    - Calling the dispose() method for every object would be very time consuming.
    - Since each widget has a parent, disposing of the parent will take care of the children.
    - This ensures that all of the resources get disposed.

Layout Overview

Layouts provide a layer between the widgets in a Composite and the Composite itself. They define where to place widgets in a Composite.

You set the Composite's layout by using the setLayout() method, and there can only be one layout per composite.

Composites can be nested and each Composite can have a layout independent of the other Composites.

Types of Layouts

FillLayout

- FillLayout is the simplest layout.
- The widgets are placed in a single column or a single row and are all the same size.
- There are two possible styles for the FillLayout: SWT.HORIZONTAL and SWT.VERTICAL.
- You can configure the FillLayout by setting member data (marginHeight, MarginWidth, spacing, etc.).

RowLayout

- RowLayout is similar to FillLayout as it places widgets in a single row or column.
- It does not force the widgets to be the same size.
- If widgets will not fit on a single line they wrap to the next line or column.
- RowLayout uses the RowData class to configure the setting for the layout.
    - Each widget must have its own instance of the RowData object.
    - Reusing the same RowData object will yield undesired results.
- Like the FillLayout, RowLayout also has member data that can be set to fine-tune the placement of the widgets.

GridLayout

- GridLayout offers more flexibility than RowLayout or Fill Layout. GridLayout is the most commonly used layout in AWIPS II.
- GridLayout arranges the widgets/Composites in a grid pattern.
- Widgets are added left to right, top to bottom.
- GridLayout uses the GridData class to configure the setting for the layout.
    - Each widget must have its own instance of the GridData object.
    - Reusing the same GridData object will yield undesired results.
- Like the FillLayout and RowLayout, GridLayout also has member data that can be set to fine-tune the placement of the widgets.
- The two most commonly used attributes of GridLayout are the number of columns and a flag to determine if the grid cells should be forced to be the same width or height.
- A widget may span multiple rows or columns.
- A widget can fill the remaining space horizontally, vertically, or both.

StackLayout

- The StackLayout stacks all of the Composites on top of each other (think of a deck of cards where only one card is visible).
- Only the top Composite is visible.
- All of the stack layers occupy the same amount of space.

FormLayout

- FormLayout is the most complex of all the layouts.
- Like other layouts, FormLayout uses a layout data class (FormData).
- FormData is crucial when using the FormLayout.
    - If FormData is not used, then all of the widgets will be placed on top of each other.

- FormData uses the FormAttachment class to control widget sizing and placement.
- Up to four FormAttachment instances can be set in the FormData object of the widget.
- Each instance of the FormAttachment corresponds to one side of the widget:
    - Top, Bottom, Left, and Right.
- FormAttachment defines the following:
    - How widgets position themselves with respect to the parent Composite or to other widgets within that Composite.
    - How the side of the widget it belongs to positions/sizes itself to the object it is attached to (parent Composite or other widget).

Composite/Group Overview

Composite and Group are containers used to hold widgets and other Groups or Composites objects. These containers can have a layout applied to them to dictate how to arrange other containers and widgets.

Composite

A ***Composite*** is the most commonly used container in AWIPS II. It features include the following:

- A Composite can have a border to show the boundaries.
- A Composite's background can have different colors.
- A Composite can only have one layout.

A ***ScrolledComposite*** is a container just like a Composite. It has a defined area that will scroll horizontally/vertically when the widgets will not fit in the boundaries of the ScrolledComposite.

Group

A Group is the same as a Composite except that a Group has a border and a title that appears in the top-left corner of the Group. Groups are used to group widgets visually.

The border of the Group can be altered using a "hint" when constructing the object.

SashForm

A SashForm is a container that can have other containers added. A divider in the SashForm allows the user to resize how the space is divided.

On certain operating systems, the SashForm widget is not visible and is only represented by what appears to be "dead space." One approach to making it stand out is to color the background of the SashForm.

Widget/Control Overview

Widgets are objects that are placed on a dialog that the user interfaces with. A Control subclasses the Widget object. "Widget" and "Control" are terms that are used interchangeably, so, for this documentation, the term "Widget" will be used because Controls inherit from the Widget class.

Widget characteristics include the following:

- Widgets have a parent (usually a Shell or a Composite/Group).
- Most widgets cannot be sub-classed (cannot be extended). Check the Javadoc of the widget to determine if it can be sub-classed.
- All widgets have a setData() and getData() methods. The setData() method stores a plain Java object and the getMethod() will retrieve the object.
- Controls can have a ToolTipText. A ToolTipText is a box that appears when the mouse hovers over a Control. It usually displays information about the control.

The following are commonly used widgets:

- Button
    - Buttons can display text, an image, or both.
    - Button types are determined by setting a "hint" when creating the widget.
    - Buttons can change the font of the text.

    - On some operating systems, the foreground and background colors can be changed.
    - Common button types include:
        - Push. A single click push button widget. SWT.PUSH is passed in when creating the widget.
        - Arrow. Like a push button but displays an arrow icon in the button. SWT.ARROW is specified in the constructor along with one of the following: SWT.UP, SWT.DOWN, SWT.LEFT, or SWT.RIGHT.
        - Check. Displays a checkbox and text that is used to display an on/off state. SWT.CHECK is passed in when creating a widget.
        - Radio. Displays a radio button and text and displays an on/off state. Once a radio button is selected, only selecting another radio button can unselect it. SWT.RADIO is passed in when creating a widget.
        - Toggle. A toggle button is a cross between a push button and a check-box. It maintains an on/off state once it is clicked. SWT.TOGGLE is passed in when creating a widget.
- Canvas
    - A canvas is a widget that is specifically designed for graphics operations.
    - A canvas can draw lines, shapes, and text.
    - Canvases can receive mouse events.
    - In AWIPS II, canvases have been used to draw custom controls.
- Combo
    - A combo widget is a hybrid of a text and a list widget.
    - Combo boxes allow users to choose from a list of choices or the user can enter text not found in the list.
    - Combo boxes do not take up as much room as a List control because it hides its information until it is displayed.
    - Only one item at a time may be selected.
    - There are three styles available for combo widgets:
        - SWT.DROP_DOWN. A combo box where the list "drops down" to show the available items. A user can type in the combo widget. The item typed in does not automatically get added into the list. Selecting an item from the list will erase the item that was typed in.
        - SWT.READ_ONLY. Restricts the user from typing in any inputs.
        - SWT.SIMPLE. On certain operating systems, this will make the list always visible. However, this does not work on the Linux platform, and SWT.SIMPLE works exactly like SWT.DROP_DOWN.
- Label
    - This is a non-editable widget that displays text or an image.
    - It cannot display an image and text at the same time.
    - Labels can have a border.
    - The foreground, background, and font can be changed on a label.
- List
    - Using hints, a List can have single or multiple selections.
    - A List does not specify a border by default so a border needs to be specified when the object is created.
        - Lists will only display strings. In most cases, a List object would be paired with an array of data where the index in the data array would match the index in the list.
        - List boxes can be created with horizontal and/or vertical scrollbars. If no scrollbars are specified, the text will be hidden when the control is resized to be smaller than the area of the text.
- ProgressBar
    - A ProgressBar is a widget that is used to visually show progress.
    - ProgressBars can be horizontal of vertical
        - Horizontal - progression moves from left to right
        - Vertical - progression moves from bottom to top
    - There are two types of ProgressBars:
        - SWT.SMOOTH

            - Slowly fills the bar until full, and updates based on what the "selection value" is set to.
            - When using the "smooth" style ProgressBar, the ProgressBar is usually updated by actions that occur in a separate thread.
            - Trying to update the ProgressBar in the event loop will yield an all-or-nothing result as the GUI does not update until the task is complete.
        - SWT.INDERTERMINATE
            - The ProgressBar indicator moves back and forth forever until the ProgressBar is hidden or removed.
- Scale
    - The Scale widget is a lot like the Slider widget as it allow the user to slide a "tab" up and down a scale to adjust a value.
    - Unlike Slider, Scale does not have arrow buttons on each side of the control.
    - "Ticks" or "Hashes" are located on both sides of the Scale (Windows platform only).
    - You should specify minimum and maximum values for the Scale.
        - **Note:** When changing the min and max values, make sure the min is never set to a number higher than the max before the max is set. When the min value is set higher than the max value, at run time the code will reset the minimum value. The same thing goes for setting the max value lower than the min value.
        - Example: If min is 0 and max is 100 and you want the min to be 1000 and the max to be 2000, first set the max to 2000 and then set the min to 1000.
    - Scale can be positioned horizontally or vertically.
- ScrollBar
    - ScrollBars appear and function like Sliders.
    - ScrollBars have a movable thumb that is used to:
        - Scroll the contents of the widget.
        - Visually represent the position.
    - Arrows are located at the end of the ScrollBar to increment or decrement the ScrollBar.
    - You do not actually create ScrollBars as they are built into widgets.
    - To access a ScrollBar from a Widget, use the getVerticalBar() or the getHorizontalBar().
- Slider
    - The Slider control in SWT looks a lot like a ScrollBar.
    - The Slider can be in a horizontal or vertical position.
    - You can set the minimum and maximum values of the Slider.
    - When setting the maximum value of the Slider, you must take into account the size of the thumb bar.
        - Example: If you want to have the Slider go from 0 to 100, you must add the thumb size to the maximum, i.e., value.
        - slider.setMaximum(100 + slider.getThumb()).
- Spinner
    - Spinner is a control that allows the user to enter and modify numerical values.
    - Integer or decimal values can be used.
    - The Spinner control has up and down arrow buttons that allow the user increment/decrement the value.
        - The amount that is incremented/decremented is configurable.
    - Minimum and maximum values of the Spinner can be specified.
- StyledText
    - The StyledText widget is a more advanced version on the Text widget.
    - The StyledText widget allows a user to type information into a text field.
    - StyledText widgets do not have a border by default.
        - Use SWT.BORDER to make the Text control have a border.
    - StyledText widgets have cut, copy, and paste methods to conveniently cut, copy, paste text from/into the widget.
    - StyleRange:
        - A StyledText widget uses the StyleRange object to specify styles for a range of text in the StyledText widget.
        - StyleRange can change the background and foreground colors, font, font style, underline, and strikeout.

        - StyleRange is also used to identify sections of text. StyledText can contain an array of StyleRanges, each representing a section/range of text.
- TabFolder & TabItem
    - TabFolders allow several "pages" of information to be stacked on top of each other, and the "pages" are accessed by clicking on the individual tabs.
    - Depending on the platform, tabs can be displayed on the top or bottom TabFolder.
        - Top is the default.
        - The location of the tabs is specified by using a style when creating the TabFolder.
    - Each tab on a TabFolder is a TabItem.
    - A TabItem can contain an image, text, or both.
    - When creating a TabItem, the parent is the TabFolder.
    - The TabItem is also the area that displays Composites/Widgets.
    - All of the TabItems will be the same size. The TabItem with the most area will determine that size for all TabItems.
    - To add contents to a TabItem you use the setControl() method.
        - **Note**: The setControl() method takes a single control as an argument.
        - Use a Composite (which is subclassed to Control) to display multiple controls/layouts in a TabItem.
- Table & TableItem
    - Tables display data in a tabular format.
    - Tables can have table columns, which can display an image, text, or both.
    - Each row in a table is a TableItem object.
        - A TableItem is an array of data that is displayed in a table row.
        - Each "cell" of the TableItem can have its font and foreground/background colors changed.
    - A TableEditor can be used to add widgets like Button, Combo, and Label to the table.
    - The Table is a very simple widget, and anything other than basic functionality provided by SWT must be handwritten. For example, JAVA Swing uses abstract table models that can be highly customized and can be swapped out of a table, it also takes care of the table columns. In SWT, the Table widget requires a lot of extra coding to take care of managing the data.
    - The Table widget has a virtual capability that will only load the data that is displayed in the table. The data will not be loaded until the table is scrolled.
- Text
    - The Text widget allows a user to type information into a text field.
    - The Text widget can be a single line or multiple lines.
    - Text widgets do not have a border by default.
        - Use SWT.BORDER to make the Text widget have a border.
    - Text widgets have cut, copy, and paste methods to conveniently cut, copy, and paste text from/into the widget.
    - The Text widget features a password style that replaces the text with asterisks or a symbol when the user types into the field.
    - You can change the font of the text.
- Tray & TrayItem
    - The Tray widget represents the system tray from the operating system.
    - The TrayItem widget represents icons that can be placed on the system tray or task bar status area.
    - TrayItems can have images, tool tips, and popup menus.
- Tree & TreeItem
    - Trees provide a selectable user interface object that displays a hierarchy of items and issues notification when an item in the hierarchy is selected.
    - The children that may be added to instances of Tree must be of type TreeItem.
    - Using a VIRTUAL style creates a Tree whose TreeItems are to be populated by the client on an on-demand basis instead of up-front. This can provide significant performance improvements for trees that are very large or for which the TreeItem population is extensive (for example, retrieving values from an external source).

Menu & MenuItem

Three types of menus are available in SWT:

1. **Bar menus**. Typically displayed at the top of the parent window (**SWT.BAR**).
2. **Dropdown menus**. Menus that drop down from a bar menu, a popup, or another dropdown menu (**SWT.DROP_DOWN**).
3. **Popup menus.** Menus that will display at the mouse cursor location and disappear when the user selects an item (**SWT.POP_UP**).

Submenus are menus that appear/popup off of an existing menu item that displays an arrow on the right side of the menu. Submenus appear when the mouse hovers over the MenuItem the submenu is associated with.

MenuItems that have submenus have a cascade style (**SWT.CASCADE**).

MenuItems can have a radio or check style.

- A menu item with a radio style behaves like a radio button (**SWT.RADIO**).
- A menu item with a check style behaves like a check box button (**SWT.CHECK**).

A popup menu is just a menu that is assigned to a widget (like a Button or List). Popup Menus:

- "Pop up" when the right mouse button is clicked.
- Can contain cascading (dropdown) menus, check menu items, radio menu items, and separators.
- Can be associated with a Shell, composites, or widgets.

Events & Listeners

SWT offers two types of listeners: ***untyped*** and ***typed.***

- Untyped:
    - Untyped listeners can lead to smaller code.
    - An untyped event listener can be registered to listen for any type of event. SWT has two classes for untyped event:
        - An interface Listener.
        - An event class named Event.
- Typed:
    - Typed listeners lead to more modular designs.
    - Typed listeners use classes and interfaces specific to each possible event.
    - For example, to listen for a button click, register a SelectionListener implementation with the button using the button's addSelectionListener() method.
    - All typed events ultimately derive from a common class: TypedEvent.
    - Many event classes have a boolean member called "doit" that you can set to false to cancel the processing of that event.
    - SWT provides implementations of every listener interface that has more than one method. The names of these classes end in Adapter.

Font Overview

Instances of the Font class manage operating system resources that define how text looks when it is displayed.

Fonts may be constructed by providing a device and either name, size, and style information, or a FontData object that encapsulates this data.

Application code must explicitly invoke the Font.dispose() method to release the operating system resources managed by each instance when those instances are no longer required.

"System Fonts" returns a reasonable font for applications to use. On some platforms, this will match the "default font" or "system font" if such can be found. This font should not be freed because it was allocated by the system, not the application.

Typically, applications that want the default look should simply not set the font on the widgets they create. Widgets are always created with the correct default font for the class of user-interface component they represent.

Color Overview

Instances of this class manage the operating system resources that implement SWT's Red, Green Blue (RGB) color model. To create a color you can either specify the individual color components as integers in the range 0 to 255 or provide an instance of an RGB.

Application code must explicitly invoke the Color.dispose() method to release the operating system resources managed by each instance when those instances are no longer required.

"System Colors" returns the matching standard color for the given constant, which should be one of the color constants specified in class SWT. Any value other than one of the SWT color constants which is passed in will result in the color black. This color should not be freed because it was allocated by the system, not the application.

Built-in SWT Dialogs

SWT has built-in convenience dialogs. Dialogs are used to get quick inputs from the user.

Some of the available dialogs include the following:

- Message Box Dialog
    - Message boxes are used to display messages and to get confirmation from the user.
    - Message boxes display icons along with messages.
        - Icon styles may be different between platforms
        - If an icon is not supported then a default icon is used
    - Message boxes have different button styles that determine which buttons will be displayed on the message box.
- Color Selection Dialog
    - The Color Selection Dialog is a dialog that allows a user to select a color.
    - The look and feel of the Color Dialog is different between each platform.
    - When a color is selected a RGB value is returned.
    - You can set the title bar text and color before the dialog is displayed.
    - When creating a new color (using an existing color object that is not null) you have to dispose of that color object first.
- Directory Selection Dialog
    - The Directory Dialog is an easy way to browse directories.
    - You can set the Directory Dialog's title bar and starting directory before the dialog is opened.
    - A customizable message can also be displayed in the Directory Dialog.
- File Open/File Save Dialog
    - The File Dialog is used for selecting files for opening or saving.
    - The type of dialog depends on the style specified at creation.
        - **SWT.OPEN.** Open dialog.
        - **SWT.MULTI.** Open dialog that can select multiple files.
        - **SWT.SAVE.** Save dialog.
    - Both the Open and Save dialogs can have file filters to restrict the file types for opening and saving. Two sets of data are used:
        - A String array of "file types."
- Microsoft Excel Spreadsheet Files (*.xls)
    - A String array of file extensions.
- "*.xls"
    - **Note:** The list of file types and the file extensions must match. If they do not, then the correct files will not display according to the file name.
- Font Selection Dialog
    - The font dialog allows the user to choose from the available fonts.
    - The user can specify the following:
        - Font type
        - Size
        - Font style (Regular, Bold, Italic, Bold Italic)
        - Color
        - Effects (Strikeout, Underline)

CaveSWTDialog

The CaveSWTDialog was created to simplify setting up dialogs and to provide additional built-in features for CAVE.

Base class for CaveSWTDialog does not require the Eclipse workbench to have started (CAVE does not have to be running to use). In 99% of cases, do not do not extend this class except for rapid prototyping or if you have perspective independent standalone components. This extends CaveSWTDialogBase and allows for perspective dependent dialogs, which requires the workbench to be running.

Always use this class over CaveSWTDialogBase unless you have a standalone component that uses dialogs.

Gotchas

Three things to watch out for are:

1. Not disposing of Color, Image, Font (creates memory leaks).
2. Certain widgets do not translate well across multiple platforms like the Scale widget.
3. The hints that are provided to widgets may not work depending on the operating system.

SWT References

Several websites offer help in understanding how to use SWT. They also provide example code snippets. Here are two helpful links:

- http://www.eclipse.org/swt/ (http://www.eclipse.org/swt/). The Eclipse website; provides code snippets and the Javadoc for the SWT classes.
- http://www.java2s.com/Tutorial/Java/0280__SWT/Catalog0280__SWT.%20htm (http://www.java2s.com/Tutorial/Java/0280__SWT/Catalog0280__SWT.%20htm). Website with SWT training.

RCP Framework

CAVE is built off the Eclipse RCP framework. Many of the things available in Eclipse are also available for use in CAVE.

Views

A view is something that can be detached or attached to the CAVE window. It functions similar to a dialog, the difference being that it can be attached or detached. For an example, see **ProductBrowserView.java** in **com.raytheon.uf.viz.productbrowser.**

Views must extend ViewPart. They must also have an extension point defined in the plugin.xml for that plugin, which is defined like the following :

```
 <extension
       point="org.eclipse.ui.views">
   <view
         allowMultiple="false"
         category="com.raytheon.viz.ui"
         class="com.raytheon.uf.viz.productbrowser.ProductBrowserView"
         id="com.raytheon.uf.viz.productbrowser.ProductBrowserView"
         icon="icons/browser.gif"
         name="Product Browser"
         restorable="true"/>
</extension>
```

Because a view is much like a dialog, you are able to use all the Standard Widget Toolkit (SWT) controls inside of it.

Perspectives

A perspective is basically everything that you see. Different perspectives can be made for different things, as we see in having D2D/GFE/Hydro/Multi-programming Executive (MPE)/Localization, etc. This includes menus, views, editors, as well as other things. Basic new perspectives can be made as follows.

See LocalizationPerspective.java and the plugin.xml inside the **com.raytheon.uf.viz.localization.perspective** plugin.

Perspective Java classes must implement IPerspectiveFactory. This will force the user to override createInitialLayout, which then the user can then add views and editors to their liking.

The **plugin.xml** will need to define the perspective as follows:

```
 <extension point="org.eclipse.ui.perspectives">
   <perspective
         class="com.raytheon.uf.viz.localization.perspective.LocalizationPerspective"
         id="com.raytheon.uf.viz.ui.LocalizationPerspective" name="Localization"
         icon="icons/localization.gif"
         singleton="true">
   </perspective>
</extension>
```

Perspectives then need to create a class that extends **AbstractVizPerspectiveManager.java**. In this class the perspective will be created and managed. See **LocalizationPerspectiveManager.java**. The **plugin.xml** must also define this as follows:

```
 <!-- Viz Localization Perspective Manager -->
<extension point="com.raytheon.viz.ui.perspectiveManager">
    <perspectiveManager perspectiveId="com.raytheon.uf.viz.ui.LocalizationPerspective"
            class="com.raytheon.uf.viz.localization.perspective.LocalizationPerspectiveM
anager"
            name="LocalizationPerspectiveManager">
    </perspectiveManager>
</extension>
```

Editors

Editors are used to allow users to edit items, files, or anything really. Editors are tied very tightly with a perspective, and often tied with Views as well. To create an editor the following needs to be done:

- For CAVE editors, see any class that extends AbstractEditor.

Extension Points

Extension points can be used to contribute functionality by plugins that are not included in the MANIFEST.MF file. A good example of how extension points work is used in the **com.raytheon.uf.viz.productbrowser** plugin. For each plugin that wants to contribute data to this plugin, something must be added to its **plugin.xml** file. This allows for the ProductBrowser plugin to receive data from the other plugins without actually having a dependency added for the other plugin.

For example:
com.raytheon.uf.viz.productbrowser

```
 <extension-point id="dataDefinition" name="dataDefinition" schema="schema/dataDefinitio
n.exsd"/>
```

Defines an extension point on the dataDefinition.exsd file.
com.raytheon.viz.radar plugin.xml (for adding radar data to the product browser)

```
 <extension
       point="com.raytheon.uf.viz.productbrowser.dataDefinition">
   <dataDefinition
           name="radarProductBrowserDataDefinition"
           class="com.raytheon.viz.radar.RadarProductBrowserDataDefinition" >
   </dataDefinition>
</extension>
```

This defines that the **com.raytheon.uf.viz.productbrowser.dataDefinition** extension point will use the **com.raytheon.viz.radar.RadarProductBrowserDataDefinition** class.
In the Java class, doing the following will recurse all the extensions and get each class that was defined in the individual **plugin.xml** files.
ProductBrowserView.java

```
 IExtensionRegistry registry = Platform.getExtensionRegistry();
IExtensionPoint point = registry
       .getExtensionPoint(ProductBrowserUtils.DATA_DEFINITION_ID);
if (point != null) {
    extensions = point.getExtensions();
} else {
    extensions = new IExtension[0];
}
```

Because all of these classes extend **AbstractProductBrowserDataDefinition.java**, we can use the same function call and get all the data to populate the product browser tree.
ProductBrowserView.java

```
 for (IExtension ext : extensions) {
    config = ext.getConfigurationElements();
    for (IConfigurationElement element : config) {
        try {
            AbstractProductBrowserDataDefinition<?> prod = (AbstractProductBrowserDataDe
finition<?>) element
                    .createExecutableExtension("class");
            String productName = prod.populateInitial();
        }
    }
```

This will make calls into each individual class for each of the functions call prod.populateInitial() and return the String for each.

Plugins

Plugins can be added to the RCP application simply by including them in the feature.xml for CAVE, or by including them in a feature.xml that is included by the AWIPS feature.xml file.

To create a new plugin, go to File -> New -> Project... -> Plug-in Project. Name the project according to the correct naming convention, leaving everything else default, and click Finish.

By default, the only thing that is created is an **Activator.java** class. This class is first called when the plugin is activated or first used.

SWT/JFace

Eclipse RCP is based on SWT/JFace components, for which there is more documentation in **CAVE_SWT.odt**.

Uframe feature.xml

The **feature.xml** is a file that allows plugin providers a means to make collections of plugins that logically go together. In addition to collecting the plugin names together, these names define the dependencies of the feature project.

com.raytheon.edex.feature.uframe
The **com.raytheon.edex.feature.uframe** project groups together all of the projects that are required to build and deploy the EDEX uframe subsystem successfully.

```
 <feature
       id="com.raytheon.edex.feature.uframe"
       label="Uframe Feature"
       version="1.0.0"
       provider-name="RAYTHEON">

   <description url="http://www.example.com/description">
       [Enter Feature Description here.]
   </description>

   <copyright url="http://www.example.com/copyright">
       [Enter Copyright Description here.]
   </copyright>

   <license url="http://www.example.com/license">
       [Enter License Description here.]
   </license>

   <plugin
           id="com.raytheon.edex.common"
           download-size="0"
           install-size="0"
           version="0.0.0"
           unpack="false"/>
</feature >
```

This extract shows the required elements of the file. Although only a single plugin is referenced here, **com.raytheon.edex.common**, the actual number is nearly 300 required plugins. This highlights another aspect of the **feature.xml**. By mentioning the plugin in the feature.xml the Eclipse environment is able to detect missing plugins. This is useful in the development environment as new plugins are added, Eclipse may issue an error alerting the developer of the need to import the specific project. At build and deployment, the feature is used to ensure that all required projects are available in the source baseline.

Logging Configuration

Configured Via XML Files

XML Files for EDEX

- **log4j.xml**
- **log4j-ingest.xml**

Logging Levels

- Level Names:
  - TRACE
  - DEBUG
  - INFO
  - WARN
  - ERROR
  - FATAL.
- TRACE is lowest level and FATAL is highest level.
- A logger set to log at a certain level will log that level and all higher levels. Example: logger set to WARN level will log all WARN, ERROR, and FATAL messages, but not TRACE, DEBUG, or INFO levels.
- Logging level is inherited from a parent logger.

Additivity

- Additivity allows logging statements to be forwarded to all the appenders in that logger as well as the appenders higher in the hierarchy.
- Set to true by default.

Appenders

An appender is an output location. All loggers log to one or more appenders.

- Layouts
  - Appenders use layouts to format the log file's name and the output
  - The PatternLayout is standard with the log4j distribution
  - Uses conversion patterns to format the output.
    - The conversions patterns are closely related to the print function in C.
    - Literal text can be inserted within the conversion pattern.

XML Entries Explained

Start by creating appenders. Here is the radar log appender:

```
  <!-- radar log -->
<appender name="RadarLog" class="org.apache.log4j.rolling.RollingFileAppender">
    <rollingPolicy class="org.apache.log4j.rolling.TimeBasedRollingPolicy">
        <param name="FileNamePattern" value="${edex.home}/logs/edex-${edex.run.mode}-rad
ar-%d{yyyyMMdd}.log"/>
    </rollingPolicy>

    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%-5p %d [%t] %c{1}: %m%n"/>
    </layout>
</appender>
```

Appender

- name is RadarLog.
- Java class is RollingFileAppender.

RollingPolicy

- Determines how the log files will roll over.

- AWIPS II uses the TimeBasedRollingPolicy.
- Requires a FileNamePattern option to be set.
- The value FileNamePattern should consist of the name of the file and a %d conversion specifier.
- Uses Java's SimpleDateFormat.
- The %d conversion specifier determines when the log will roll over.
- AWIPS2 log files roll over daily.
    - %d{yyyyMMdd}
    - FileNamePattern value is the path to the log file and the file's name.

Layout

- Uses the PatternLayout.
- The conversion pattern is the format of the output line.
- ConversionPattern value above is %-5p %d [%t] %c{1}: %m%n.
    - %5-p is the log priority left justified at 5 spaces.
    - %d is the date time stamp in this format: 2011-12-14 17:11:16,509.
    - [%t] is the name of the thread running surrounded by [].
    - %c{1} prints the category of the logging event, where the number means to print the corresponding number of right most components. 1 prints just the rightmost component.
    - The colon is just literal text to signify the start of the log text.
    - %m is the actual message.
    - %n is a platform dependent line separator character.

```
  <logger name="com.raytheon">
    <level value="INFO" />
</logger>

<logger name="com.raytheon.edex.plugin.shef">
    <level value="DEBUG" />
</logger>
```

Logger

- name is the package name to be logged.
- Any code inside the com.raytheon package will be logged at level INFO.
- The logger com.raytheon.edex.plugin.shef overrides the value set at the **com.raytheon** level and uses the DEBUG level logging for log entries inside the **com.raytheon.edex.plugin.shef** package.
- Add an appender to a logger with the **.**
- If no appender-ref listed the logger will use the default logging.

Logging in the Java Code.

- Define the status handler:

```
  private static final transient IUFStatusHandler statusHandler = UFStatus.getHandler(Cla
ssName.class);
```

- Call any of the statusHandler.handle() methods to send the message.
- Messages go to log file and AlertViz for notification.
- Notifications are configurable in the Alert Visualization Configuration dialog.
- Additional Info available via the log4j website: http://logging.apache.org/log4j/index.html (http://logging.apache.org/log4j/index.html)

AWIPS II deploy-install.xml

To understand what the **deploy-install.xml file** is and how to use it, you will need to be familiar with **ant**. Ant, an Apache project, is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The primary use case of Ant is to build Java applications. Ant supplies a number of built-in tasks allowing users to compile, assemble, test, and run Java applications.

The AWIPS II deploy-install.xml file (provided that you have **ant** installed) will allow you to deploy EDEX directly from your Eclipse workspace during development provided that you have installed AWIPS II Standalone software package (installing and configuring AWIPS II Standalone is outside the scope of this document).

Within your workspace, the deploy-install.xml file can be found in the build.edex directory. To run deploy-install.xml from within Eclipse, right click on the file and select: Run As -> **Ant Build...** in the context menu that is displayed. The "Edit Configuration" dialog will be displayed. Select the "Main" tab in the Edit Configuration dialog and look for the Arguments field. There are a few arguments that you will have to provide before you can use deploy-install; a few are required and others are optional. One of the required arguments is install.dir; if you are using the standard ADE setup this argument should always be set to "/awips2/edex": **-Dinstall.dir=/awips2/edex**. Other arguments that you can specify include:

- **-Dupdate.python**
- **-Dlocalization.sites**

The update.python argument expects a yes / no value. If you set update.python to "yes", **deploy-install.xml** will update the ufpy and dynamicserialize site-packages in your python install. However, in order for this update to work, you must have the pythonPackages project in your workspace and the pythonPackages project must have dynamicserialize and ufpy sub-directories. If the pythonPackages project is not present in your workspace, deploy-install will fail.

The **localization.sites** argument expects nothing, a single site identifier or a comma-separated list of site identifiers. When a localization site is specified, deploy-install will copy the files from the associated localization project in your workspace to your EDEX installation. (**WARNING:** This will overwrite any localization files that are already present for the site.) The localization project(s) for any site that you specified must be in your workspace; if not, deploy-install will fail.

Once you configure **deploy-install.xml**, you will be able to bypass the configuration step completely and immediately run deploy-install by right clicking on the file in Eclipse and select: **Run As -> Ant Build** in the context menu that is displayed. As deploy-install is running, it will log information in the Eclipse console so that you will be able to determine if deploy-install was successful or if it failed.

Clustering

This is an EDEX-only concept, implemented via database row locks. All clustering goes through **com.raytheon.uf.edex.database.cluster.ClusterLockUtils**. The general conops is to use an easily identifiable name field that is specific to your overall flow and then to use the details column to specify the unit of work. The unique combination of the name and details provide the specific database row to lock. ClusterLockUtils is used directly to cluster lock specific pieces of code, for example, **com.raytheon.edex.plugin.gfe.config.GFESiteActivation.java**, where the name is "GFESiteActivation" and task details is "Initialization: OAX". The different lock calls to ClusterLockUtils allow for customization of other only returning once a lock is granted, the timeout of when to override a current lock, and overriding of IClusterLockHandler can you give you custom control of how the extrainfo column is used. The state of locks can be viewed in postgres. It is stored in the metadata database, awips schema, cluster_task table.

There are clustered camel contexts to emulate singleton services, so an entire set of routes is only running on a work machine in the cluster based on the context name. The clustered context needs to be registered with the clustered camel context manager and the context set to **not** auto-start, for example, **purge-spring.xml**.

Note: Changing the extrainfo column of a ClusteredContext to a different host/jvm will cause that service to switch to the designated jvm at the next sync interval (usually 20 seconds).

There are clusteredquartz endpoints for periodic kickoff of work that can be run on any system, but the work unit should only happen once (example: **gfe-request.xml**).

If CAVE ever needed cluster locking, a Thrift request would need to be sent to EDEX to interface with ClusterLockUtils on the client's behalf.

Request JVM

Thrift Request and Handler API

EDEX supports a request-handler API that allows client applications (like CAVE) to send data that will be processed by the EDEX server and optionally return results back to the client. Due to its use of Dynamic Serialize, Java-based and Python-based clients can interact with the server through this API.

Creating a Request

To create a new request type, create a new class that implements the IServerRequest interface (see **com.raytheon.uf.common.serialization.comm.IServerRequest**). Because this class will be sent to the server via Dynamic Serialize/Thrift, you must also annotate your new class with the Dynamic Serialize annotations. The class itself should have the **@DynamicSerialize** annotation and any fields of the class that will be needed to process the request should have the **@DynamicSerializeElement** annotation. Also, any fields marked as **@DynamicSerializeElement** will need associated getters and setters.

The following sample code demonstrates a very simple request type.

```
  // ASampleRequest.java
@DynamicSerialize
public class ASampleRequest implements IServerRequest {
    @DynamicSerializeElement
    private long userId;

    @DynamicSerializeElement
    private String siteId;

    @DynamicSerializeElement
    private String message;

    public long getUserId() {
        return userId;
    }

    public String getSiteId() {
        return siteId;
    }

    public String getMessage() {
        return message;
    }

    public void setUserId(long userId) {
        this.userId = userId;
    }

    public void setSiteId(String siteId) {
        this.siteId = siteId;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Creating a Request Handler

To process requests of your new request type, you will need to do two things:

1. Create a class that implements the IRequestHandler<YourNewRequestType> interface (see **com.raytheon.uf.common.serialization.comm.IRequestHandler**).
2. Register your request handler with the request handler registry.

To implement the IRequestHandler interface properly, your new handler class must implement a method named handleRequest, which accepts your request as the only argument and returns an Object. This return value does not have to be typed Object because primitives, Strings, and user-defined classes are also acceptable. The only requirement is that the return type supports serialization via Dynamic Serialize.

The following sample code demonstrates a request handler for the request type from the previous section.

```
 // ASampleRequestHandler.java
public class ASampleRequestHandler
        implements IRequestHandler<ASampleRequest> {
    @Override
    public String handleRequest(ASampleRequest request) {
        StringBuilder retVal = new StringBuilder();
        retVal.append("User ");
        retVal.append(request.getUserId());
        retVal.append(" from site ");
        retVal.append(request.getSiteId());
        retVal.append(" says ");
        retVal.append(request.getMessage());
        return retVal.toString();
    }
```

To register your new handler, you will have to alter the EDEX plugin's spring request XML (this will be in an XML file named ***-request.xml**) file and add the following:

```
 <!-- samplePlugin-request.xml -->
<bean id="sampleHandler" class="com.raytheon.edex.plugin.sample.handlers.ASampleRequestH
andler"/>

<bean factory-bean="handlerRegistry" factory-method="register">
    <constructor-arg value="com.raytheon.uf.common.dataplugin.sample.requests.ASampleReq
uest"/>
    <constructor-arg ref="sampleHandler"/>
</bean>
```

So, you create a bean for the request handler, then register it with the handlerRegistry, and, by specifying your request type in constructor argument, tell the server to send all requests of that type to your handler bean.

Sending the Request with Java from CAVE

In order for the client to send a request, developers should use the ThriftClient class (see **com.raytheon.uf.viz.core.requests.ThriftClient**). This will automatically send your request to the configured EDEX server. Just call the static method sendRequest and pass in the request you want to send, and the server's response will be returned. If the request handler threw an exception while processing your request, sendRequest will throw this exception back to the caller.

Sending the Request with Python

Since the Request/Handler API communicates using Dynamic Serialize, pure Python clients can also interact with EDEX using the same request types that Java does. AWIPS II provides a ufpy Python package, which includes a ThriftClient class for communicating with EDEX. However, any requests you wish to send through ThriftClient must be converted to pure python classes. Classes

within the dynamicserialize.dstypes Python package have already been converted for use in baseline tools. Further information on converting Java classes to Python is covered in the documentation on Dynamic Serialize.

AWIPS II Data Purging

The purging in AWIPS II is based largely on the rule-based purging scheme in use currently by AWIPS I. Due to some fundamental architecture differences between AWIPS I and AWIPS II, the AWIPS II data purging model differs in some regards.

Configuration

By default, the purge routine runs off of a quartz timer once an hour at 30 minutes past the hour. This value may be changed by modifying the purge.cron entry in the **/awips2/edex/conf/spring/project.properties** file. The purge component is configured in the res/spring/purge-spring.xml file located in the **com.raytheon.uf.edex.purgesrv** plugin. This file defines several beans and camel routes, including the aforementioned quartz timer job, used in the purge process.

As will be explained later, each plugin is responsible for purging its own data in whatever manner it chooses. Each plugin is responsible for assigning a data access object (DAO) in their <plugin_name>-common.xml spring configuration file in the <plugin_name>Properties bean. If a plugin does not specify a custom DAO to use, the default plugin DAO (**com.raytheon.uf.edex.database.plugin.PluginDao**) is used. Specifically, the purge behavior is defined in two methods on the data access object. These are purgeExpiredData and purgeAllData. The default plugin DAOs implement the default rule-based purge routine. Plugin-defined DAOs may override these methods to define their own custom purge behavior.

Purge Execution Flow

The quartz timer sends a message to the **com.raytheon.uf.edex.purgesrv.PurgeSrv** bean defined by spring. The PurgeSrv then retrieves all the registered plugins from the PluginRegistry. A loop then delegates the purging of data to the plugin by calling the purgeExpiredData method on the DAO. As mentioned above, plugins may use the default purge routine or define their own.

Default Purge Behavior

If a plugin chooses to use the default purge behavior, the plugin must define rules for how and what to purge. The plugin should contain a file called **<plugin_name>PurgeRules.xml located in the utility/common_static/base/purge** folder. If a plugin does not define this file, the default purge rule will be used to purge their data. The default purge rule is defined in **defaultPurgeRules.xml** located in the **utility/common_static/base/purge** folder of the **com.raytheon.uf.edex.database** plugin. Currently, the default rule is to purge all data with reference times older than one day based on the current time.

A purge is identified by an id field. The id consists of the plugin name and a purge key. The key field defines what fields in the plugin record class to examine to determine if the data should be purged. For example, the grib plugin defines the following rule:

```
  <rule>
     <id>
         <pluginName>grib</pluginName>
         <key>modelInfo.modelName=ETA</key>
     </id>
     <versionsToKeep>2</versionsToKeep>
</rule>
```

The key field in the id is identified as modelinfo.modelName=ETA. This means that the purger will examine the modelInfo field of GribRecord (the record class assigned to the grib plugin) and subsequently look at the modelName field of the modelInfo field to make its purge decision. In this case, this rule is saying to keep two versions of grib records whose modelInfo.modelName field is ETA. Or, in other words, keep two runs of the ETA grib model.

A plugin may specify a plugin default rule. This rule is used to prevent data that may not get examined by the defined rules from not getting purged. If a plugin does not specify a plugin default rule, then the global default rule mentioned earlier is used. A plugin default purge rule is defined as follows:

```
  <rule>
    <id>
        <pluginName>grib</pluginName>
        <key>default</key>
    </id>
    <versionsToKeep>2</versionsToKeep>
</rule>
```

The plugin name is specified as the plugin that this rule applies to. The key is specified as default. This rule is saying to keep two versions of all data not addressed by other defined rules. Taking grib as an example: Say a new model, or an unknown model, starts to be ingested by EDEX. Obviously, no rule has been defined for this data, but we do not want the data to persist forever and fill the disk. The default purge rule kicks in and this data is purged in a reasonable way until a specific rule can be defined.

If a plugin, such as grib, defines purge rules based on fields in the class (in the case of grib modelInfo, modelName) *and* the plugin stores HDF5 data, then an additional file ***must*** be present for purge to operate correctly. This file is called **<plugin_name>PathKeys.xml** and is located in the **utility/common_static/base/path/** directory of the plugin. This file is read by the purger to tell it the fields on which this plugin is basing its purging. This file also determines the layout of the HDF5 data in the HDF5 data store. For grib, the contents of the file are as follows:

```
  <pathKeySet>
    <pathKey>
        <key>modelInfo.modelName</key>
        <order>0</order>
    </pathKey>
</pathKeySet>
```

The key field is the record class field to use when persisting HDF5 data. The order is the order in which these fields should be appended when determining the HDF5 path. In this case, the modelInfo.modelName field from the GribRecord class is used. Examining the HDF5 directory for grib shows this:

```
 grib
|-- AK-NamDNG5
|-- AK-RTMA
|-- AKWAVE239
|-- AKwave10
|-- AKwave4
|-- AUTOSPE
|-- AVN
|-- AVN203
|-- AVN211
.
.
.
```

You can see that the model name is used as the directory name. Expanding that out a little more, you can observe that the actual HDF5 files then reside in those directories:

```
 grib
|-- AK-NamDNG5
|   |-- AK-NamDNG5-2012-03-22-06-FH-000.h5
|   |-- AK-NamDNG5-2012-03-22-06-FH-003.h5
.
.
.
```

Important Note: *A plugin may not use more than one key for defining purge rules.*

This means that for grib, you cannot have rules with different keys meaning you can have one rule with key modelInfo.modelName and another rule with key modelInfo.genprocess. This is due to how the purge routine was designed. Because plugins have wide latitude for defining how their data is persisted, concessions had to be made on what the purge routine was capable of doing The purger examines the pathKeys.xml file to determine what to look at in the database. Then, based on that key, it determines the list of refTimes matching that criteria. As an example, the grib plugin uses modelInfo.modelName as its key. Therefore, the purger will first determine all the unique modelNames found in the database. Then, it will find all of the unique reference times for each of those modelNames. An example representation of the lists follows:

- ETA (2012-03-22-00, 2012-03-22-06, 2012-03-22-12)
- ETA218 (2012-03-22-00, 2012-03-22-06)

The purger then uses these lists to determine what data to purge. If multiple keys were allowed, the purger would potentially be making thousands of queries to the database to determine all the reference times that apply to those keys and then have to find out if multiple rules apply. The logic has the potential to get extremely complex and more importantly, time consuming. In this case, we are keeping two versions of the ETA model, meaning that all ETA data with refTime 2012-03-22-00 is deleted from the database and the HDF5 directory. This also adds efficiency to deleting HDF5 data. Instead of calling many deletes to HDF5 directory to cherry pick specific pieces of data from each file, which could get very time consuming, entire files may be deleted. Essentially, the pathKeys file makes the HDF5 data get organized in manner that facilitates fast purging.

The default purge routine relies on a plugin using the DefaultPathProvider. If the plugin does not use this path provider, the purge routine may fail.

Purge Rules

A purge rule may specify the following parameters:

- **versionsToKeep.** The number of versions for this key to keep. Note that a version is a reference time.
- **period.** Max period between the current time and the oldest time stamp of files to keep; defaults to 0 which means do not time purge. The leading tilde (~) on the period means to calculate from the latest time instead of the current time
- **delta.** Data with a time stamp separated by less than this from the next newest file will not be kept. Defaults to zero, which means do not consider time separation. If a leading equals (=), keep only files an exact multiple of this delta time, if a leading tilde (~), keep only the one file closes to an exact multiple of this delta time.
- **round.** Round times by this before deciding whether to purge. Defaults to zero, which means do not round. The rounding time interacts with the delta, but not the period. If a leading plus sign (+), add the time instead of rounding by it. If consecutive data round to the same time, then if one is kept, they will all be kept.
- **LogOnly.** Do not actually purge by this entry, only log what would have been purged
- **modeTimeToWait.** Time period to wait after the insert time of the latest data to purge normally; this allows the most recent file to be completed before the oldest is purged

PluginRegistry

The EDEX PluginRegistry provides a means of setting various property values that are specific to each plugin. These property values are stored in a single object, PluginProperties, which is keyed using the pluginName. **Figure 3-4** provides a class diagram of the relationship.

The PluginProperties class used in PluginRegistry exposes important properties that are used while creating a plugin at startup as well as providing information that will be used during the lifetime of the plugin.

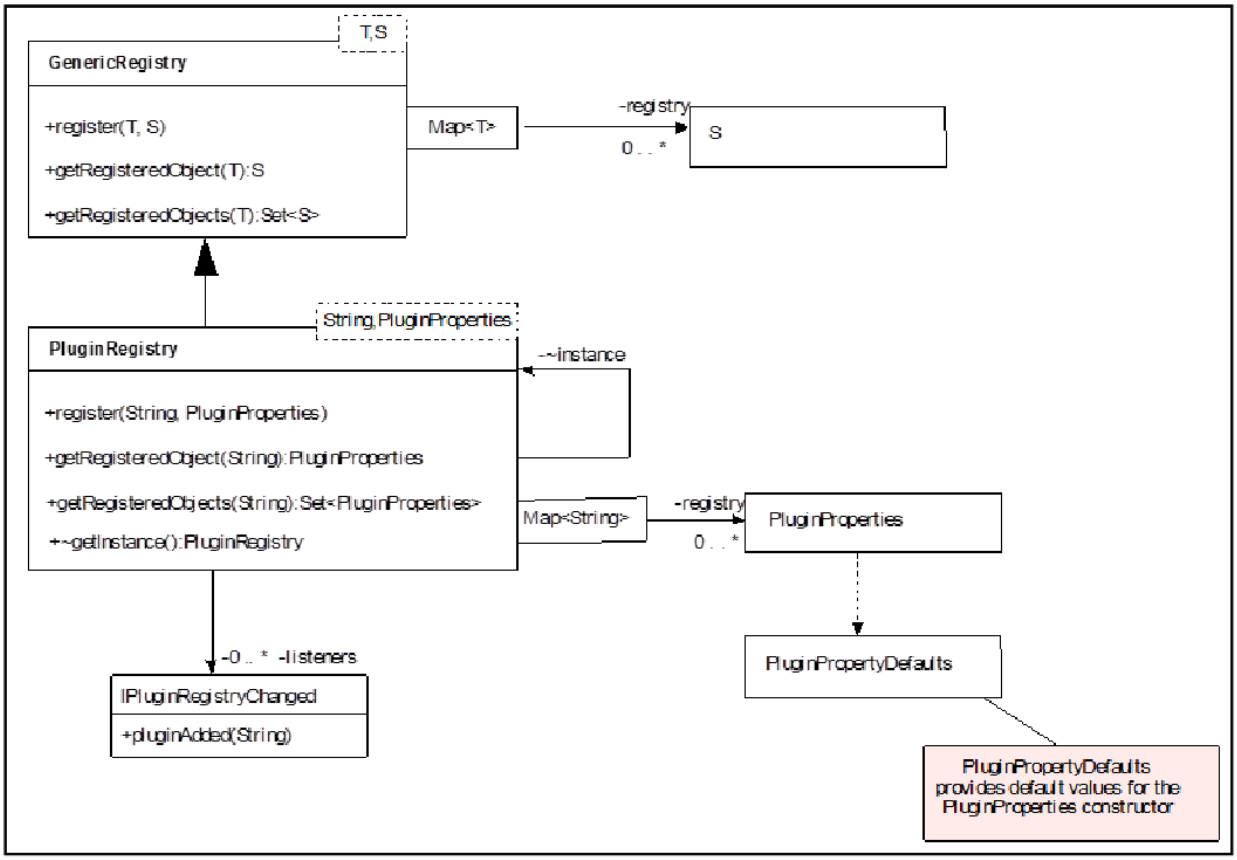

Figure 3-4. Plugin Registry

Properties Exposed by PluginProperties

The properties exposed by PluginProperties are:

- **pluginName.** The short name of the project.
- **pluginFQN.** The fully qualified name of the project.
- **Database.** The database that should be used.
- **Record.** The fully qualified name of the record object to be registered.
- **dao.** The Dao (Data Access Object) that implements store behavior for the record object.
- **Initialize.** An initializer class that performs any initialization required while the plugin is being registered.
- dependencyFQNs
- **pathProvider.** A class that provides a path to the HDF repository for the record object.
- **Compression.** The type of compression to be used on the data.
- **initialRetentionTime.** Use of this property is deprecated.

The values for database, initializer, dao, initialRetentionTime, and pathProvider are set to default values that are declared in the class PluginPropertyDefaults in the "edex.xml" startup configuration.

The following properties show some typical values for the properties. The "record" property is important as the initialization tasks use this record to construct the build table SQL required when a table is initially created in the database.

Given a sample decoder plugin named mytest the following properties could be set as follows

```
 <bean id="mytestPluginName" class="java.lang.String" >
    <constructor-arg type="java.lang.String" value="mytest" />
</bean>

<bean id="mytestProperties" class="com.raytheon.uf.common.dataplugin.PluginProperties" >
    <property name="pluginName" ref="mytestPluginName" />
    <property name="pluginFQN" value="com.raytheon.uf.common.dataplugin.mytest" />
    <property name="dao" value="com.raytheon.uf.common.dataplugin.mytest.dao.MyTestDao"
/>
    <property name="record" value="com.raytheon.uf.common.dataplugin.mytest.MyTestRecor
d" />
</bean>
```

The properties defined are then registered with the pluginRegistry and any initialization actions occur at this time.

```
 <bean factory-bean=="pluginRegistry" factory-method="register" >
    <constructor-arg ref="mytestPluginName" />
    <constructor-arg ref="mytestProperties" />
</bean>
```

Plugin Startup

Figure 3-5 shows both the system initialization and the plugin initialization involved with PluginRegistration. At startup the **edex.xml** configuration file begins by creating a set of default plugin properties. These properties are later used to populate initial properties in the PluginProperties constructor as each new plugin is defined. When values for certain properties are not explicitly specified by a plugin, these defaults ensure that important properties always have meaningful values.

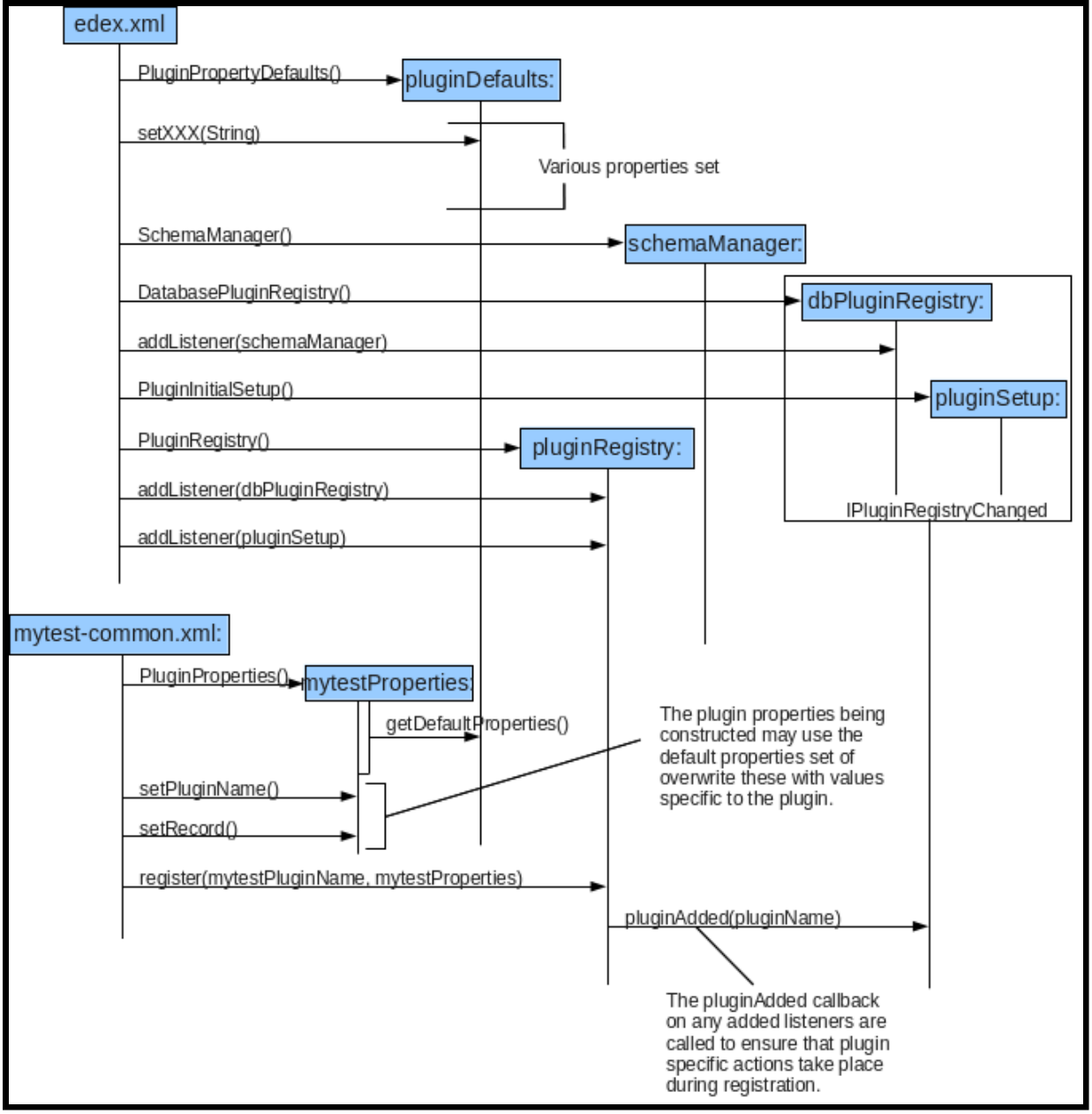

Figure 3-5. Plugin Startup: System Initialization and Plugin Initialization

The **edex.xml** next creates callback classes that will later use specified plugin properties to perform additional initialization as follows.

1. The **schemaManager** bean is created, followed by the **dbPluginRegistry**. This **schemaManager** is then added to the **dbPluginManager** as a listener for *registryChanged* events in the **dbPluginRegistry**.
2. The **dbPluginRegistry** is in turn added as a *registryChanged* listener on **pluginRegistry** after it has been created.
3. Other *registryChanged* listeners may be added to either the **dbPluginRegistry** or **pluginRegistry** in a similar fashion.
4. Later, as plugins are defined and registered with the **pluginRegistry**, the *pluginAdded* method on any listeners in the **pluginRegistry** are called. For the above example:
    - A plugin is defined and registered with the **pluginRegistry**.
    - The **pluginRegistry** then calls the *pluginAdded* method on the **dbPluginRegistry**.
    - The **dbPluginRegistry** performs its tasks then calls the *pluginAdded* method on the **schemaManager**.
    - The **schemaManager** uses information in the pluginProperties to perform various database specific actions during startup. Most common would be creating the DDL for the table for initial creation and then creating that table if it does not exist.

The following extract is typical of the definitions for the items mentioned above.

```
 <!-- Create the default properties →
<bean id="pluginDefaults"
        class="com.raytheon.uf.common.dataplugin.defaults.PluginPropertyDefaults">
    <property name="database" value="metadata" />
    <property name="initializer" value="com.raytheon.edex.plugin.DefaultPluginInitialize
r" />
    <property name="dao" value="com.raytheon.edex.db.dao.DefaultPluginDao" />
    <property name="initialRetentionTime" value="24" />
    <property name="pathProvider" ref="defaultPathProvider"/>
</bean>
<bean id="pluginRegistry"
        class="com.raytheon.uf.edex.core.dataplugin.PluginRegistry"
        factory-method="getInstance"/>
<bean id="dbPluginRegistry"
        class="com.raytheon.uf.edex.database.DatabasePluginRegistry"
        factory-method="getInstance"/>

<!--  schemaManager initializes database tables db plugin is registered -->
<bean id="schemaManager" class="com.raytheon.edex.db.purge.SchemaManager"
        factory-method="getInstance" />
<!-- Add the schemaManager as a listener on the dbPluginRegistry -->
<bean factory-bean="dbPluginRegistry" factory-method="addListener">
    <constructor-arg><ref bean="schemaManager"/></constructor-arg>
</bean>
<!-- This causes the data plugin's database tables to be created when a plugin is regist
ered -->
<bean id="dbPluginRegistryListenerAdded" factory-bean="pluginRegistry"
        factory-method="addListener">
    <constructor-arg><ref bean="dbPluginRegistry"/></constructor-arg>
</bean>
<!-- Runs the data plugin's initializer when a plugin is registered -->
<bean id="pluginSetup" class="com.raytheon.edex.plugin.PluginInitialSetup" />
<!-- Note the "depends-on" reference requires that the bean  "dbPluginRegistryListenerAd
ded" be defined prior to this bean being created. -->
<bean factory-bean="pluginRegistry" factory-method="addListener"
        depends-on="dbPluginRegistryListenerAdded">
    <constructor-arg><ref bean="pluginSetup"/></constructor-arg>
</bean>
```

EDEX Decoder Plugins

Generic Decoder

EDEX decoders provide the capability of transforming incoming data, coded or not, into a canonical data form that is later persisted to a data store. By providing the data in a canonical form, only a single data access object needs to be provided to retrieve these data. The decoders within the ingest component may receive data from outside sources, coded weather data for example, or as the output of other decoders to be transformed into a different format.

A simple implementation of a decoder is a class that exposes an "action" method that will be called by the "camel" subsystem. When designing such a class some of the following rules and conventions must be observed.

The camel Spring "wiring" XML allow the developer to expose the decoder action method to camel. The following is a snippet from a configuration XML of a typical decoder.

```
  <bean id="mytestDecoder"
        class="com.raytheon.uf.edex.plugin.mytest.MyTestDecoder"
        depends-on="mytestPluginName">
    <constructor-arg ref="mytestPluginName" />
</bean>
```

The bean name *mytestDecoder* is the decoder's name reference. This will be used when a reference to the bean is required. The "**class**" argument indicates that class that will be created, and "**depends**" on indicates that *mytestPluginName* must be defined before this bean will be created. Note that when the bean is created and used, there is a single instance. The above "bean" definition would use the following java code:

```
  public MyTestDecoder(String name) {
    pluginName = name;
}
```

Note that the only correspondence between the "bean" constructor argument and the Java code is that the constructor parameter takes a single argument of type String. Camel uses reflection to determine the correct constructor to be called.

It is important to recognize that a single instance of the class is created. That means that any attributes of the class are common to an invocation of the bean methods. Any data that is declared within a method is visible to that method only, however any data declared at the class level is visible to any invocation of any method. So it is best to ensure that thread safety is a high design priority.

The following reference actually uses the defined bean and indicates that the method "*decode*" will be invoked as the action method of the class. Note in particular that there are no arguments to the method. Camel uses reflection to examine the decoder's decode method and determine how to transform the incoming data if that is possible.

```
  <bean ref="stringToFile" />
<bean ref="mytestDecoder" method="decode" />
```

As an example, given the above XML, the "bean" *stringToFile* converts a byte array or String to a File reference given the assumption that the byte array/String represents a fully qualified file name. Then Camel examines the bean *mytestDecoder*'s decode method to determine what parameter(s) it may take. The following code outlines some possible variations:

- ```
  public PluginDataObject [] decode(File file)
  ```

  The file reference created by stringToFile is passed unchanged.

- ```
  public PluginDataObject [] decode(byte[] data)
  ```

The contents of the file reference are first read into a byte array and this is then passed to the decode method.

- ```
  public PluginDataObject [] decode(String data)
  ```

  The contents of the file reference are first read and converted to a String and this String is passed to the decode method.

The first method, a direct File reference, is useful when dealing with large files. This allows the decoder to keep a minimum of data in memory at a time. The developer should keep in mind that the file being referred to should not be deleted or modified as it may be used by other clients not known to the developer.

Using the second method, the entire contents of the file are read as a byte []. This is useful for smaller data files as the developer has the data readily available and has no concern regarding file manipulation. Like the first method, this allows the developer access to the raw data. However, unlike the first method, the developer does not have access to the underlying file reference.

The third method makes the assumption that the underlying data is of a String type. Although useful, this method must be used with care. The Java runtime uses certain encoding to transform raw data into a String. If any data within the raw data does not map correctly then, the result will not be accurate or even rendered unusable.

The action method "decode" declares a return type of an array of PluginDataObject. The return value should be a not-null reference with zero or more entries. Returning zero entries allows the data to be processed fully by "downstream" processes as empty is valid. The not-null array keeps an exception from being thrown if a downstream process is expecting an array of PluginDataObject.

The *strategy* design pattern is commonly used with designing decoder classes. **Figure 3-2** illustrates a situation where several parser classes may be used depending on the type of the data.

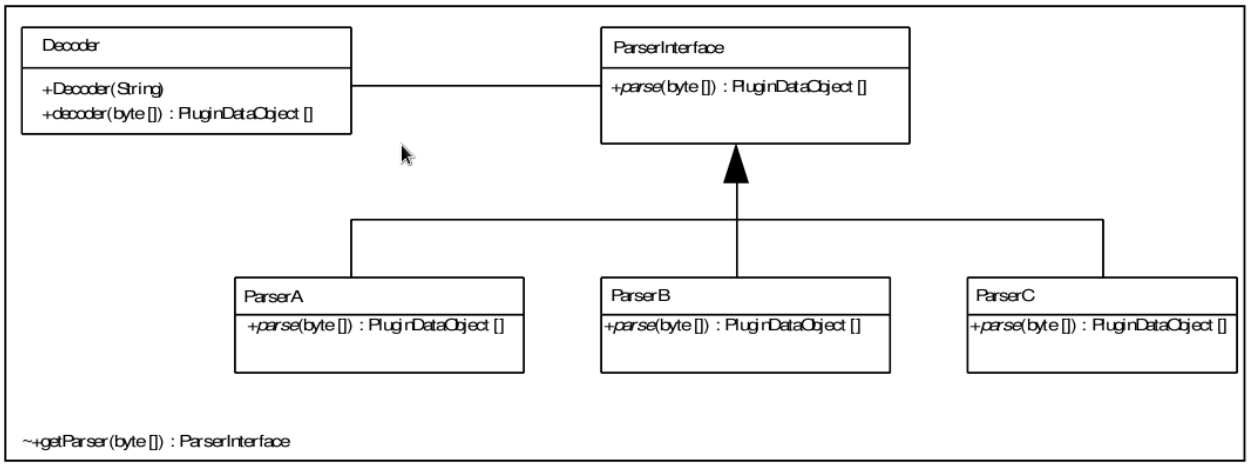

Figure 3-2. Parser Classes

Implementing the parsing services behind the ParserInterface decouples the Decoder "frontend" from the actual parser implementation. This makes software maintenance easier as the Camel interfaces need little or no changes when changes occur within the Parser class. This division of labor also allows the Parser to be tested in a stand-alone mode.

Figure 3-3 shows the sequence of events that are typical during the lifetime of a decoder class. At startup camel creates a bean instance of the decoder. At that point the decoder instance should be ready to begin receiving messages via its exposed "action" methods.
  ```

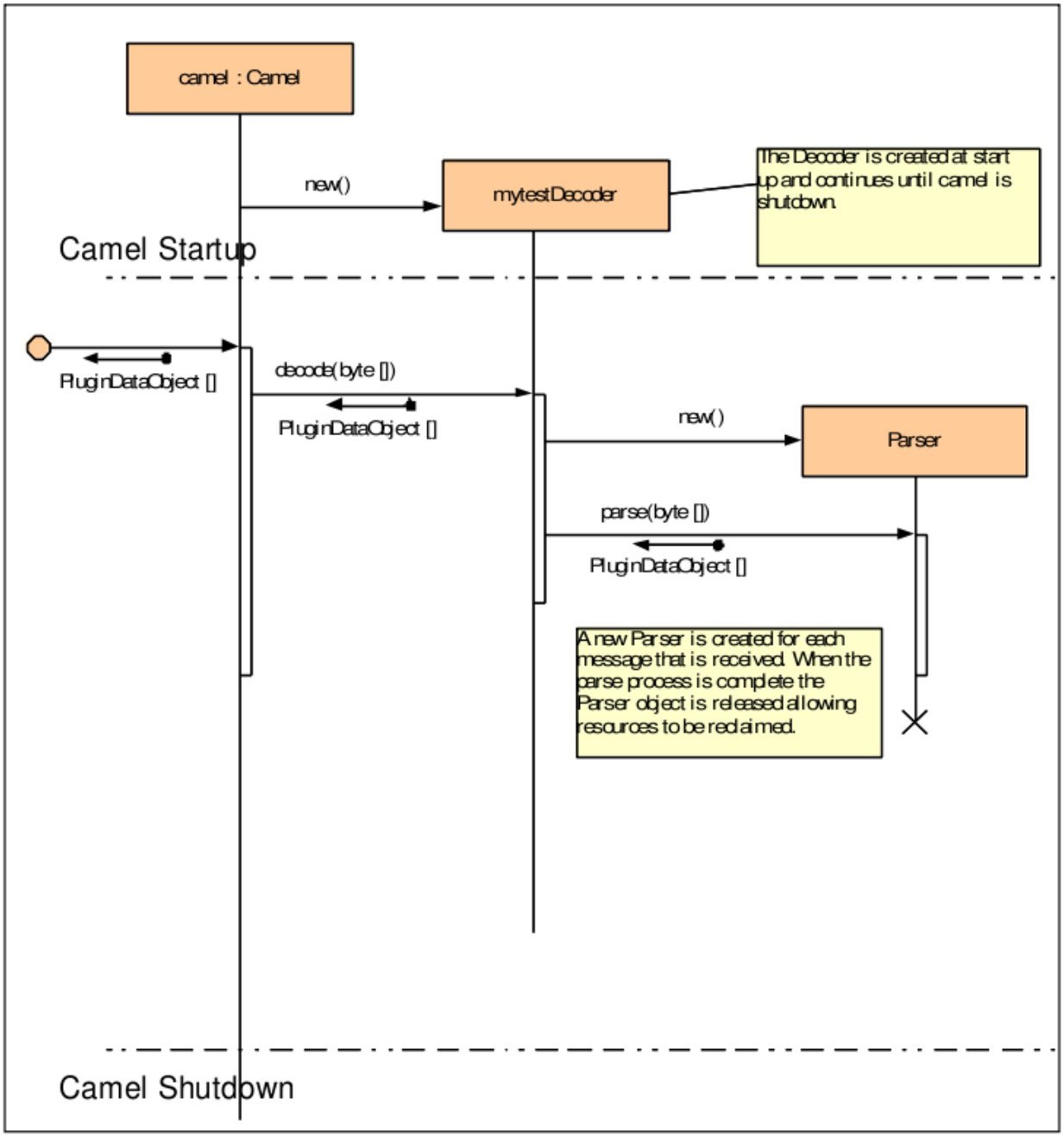

Figure 3-3. Typical Sequence of Events During Lifetime of a Decoder Class

Between Startup and Shutdown a simple sequence occurs. Data, from some source, is received by camel and, after being identified as data to be processed by the *mytestDecoder*, is routed to the this decoder. The "bean" *mytestDecoder* has identified **decode** as its action method, which is called with the incoming data. For each new data message a new Parser object is created, its parse method is then called to decode the data fully and return an array of PluginDataObjects representing the decoded data. This is then returned to camel to be processed further downstream. Note that because of the way this occurs, the decoder should make no assumptions about how the data is to be used by downstream consumers. The data object representing the decoded data should be the only output of the decoder.

Camel-Spring Configuration xml

Normally the Spring configuration is split into two sections. The first, named "plugin"-common.xml, usually contains information that should be "available" prior to the plugin specific information being defined. The following shows a simple, yet complete definition for the "mytest" plugin.

```
 <beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:amq="http://activemq.apache.org/schema/core" xmlns:xsi="http://www.w3.org/200
1/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://activemq.apache.org/schema/core http://activemq.apache.org/schema/core/ac
tivemq-core.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/came
l-spring.xsd">

    <bean id="mytestPluginName" class="java.lang.String">
        <constructor-arg type="java.lang.String" value="mytest" />
    </bean>

    <bean id="mytestProperties" class="com.raytheon.uf.common.dataplugin.PluginPropertie
s">
        <property name="pluginName" ref="mytestPluginName" />
        <property name="pluginFQN" value="com.raytheon.uf.common.dataplugin.mytest" />
        <property name="record" value="com.raytheon.uf.common.dataplugin.mytest.MyTestRe
cord" />
    </bean>

    <bean id="mytestRegistered" factory-bean="pluginRegistry" factory-method="register">
        <constructor-arg ref="mytestPluginName"/>
        <constructor-arg ref="mytestProperties"/>
    </bean>

</beans>
```

The XML attributes in the <beans...> tag is always required and identifies camel XML namespace information. </beans...>

The first bean of interest is "mytestPluginName." This bean is identified by name "id="mytestPluginName", the class that represents the value class="java.lang.String". The value that will be assigned to the bean is set using a "constructor-arg", that is, "constructor argument", which identifies the argument type and value. Once this bean is created it is commonly used by using the form ref="name", where name is the **id** associated with the bean. This is defined so that all references for pluginName are the same.

The second bean sets up some important properties associated with the plugin.

- **pluginName.** The "short" name of this plugin. Normally the pluginName bean reference.
- **pluginFQN.** The fully qualified name of the data plugin.
- **Record.** This is the fully qualified name of the Data Record that will be used to return data from this decoder.

The third bean registers the above properties with the pluginRegistry. This registry makes the defined property information available to other services using the pluginName as a key.

Once these items have been declared the plugin has been identified to the system.

The second configuration section, named "plugin"-ingest.xml usually contains information that tells camel how this plugin handles data being received by the ingest system. In the following the <beans...> preamble is not shown. </beans...>

```
 <bean id="mytestDecoder" class="com.raytheon.uf.edex.plugin.mytest.MyTestDecoder"
       depends-on="mytestPluginName">
    <constructor-arg ref="mytestPluginName" />
</bean>

<bean id="mytestDistRegistry" factory-bean="distributionSrv" factory-method="register">
    <constructor-arg ref="mytestPluginName" />
    <constructor-arg value="jms-dist:queue:Ingest.mytest?destinationResolver=#qpidDurabl
eResolver" />
</bean>

<bean id="mytestCamelRegistered" factory-bean="contextManager"
        factory-method="register"
        depends-on="persistCamelRegistered">
    <constructor-arg ref="mytest-camel" />
</bean>

<camelContext id="mytest-camel" xmlns="http://camel.apache.org/schema/spring"
        errorHandlerRef="errorHandler" autoStartup="false">
    <!-- Begin mytest routes -->
    <route id="mytestIngestRoute">
        <from uri="jms-generic:queue:Ingest.mytest?destinationResolver=#qpidDurableResol
ver" />
            <setHeader headerName="pluginName">
            <constant>mytest</constant>
            </setHeader>
        <doTry>
                <pipeline>
                <bean ref="stringToFile" />
                <bean ref="mytestDecoder" method="decode" />
                <to uri="directvm:indexAlert" />
            </pipeline>
            <doCatch>
                <exception>java.lang.Throwable</exception>
                <to uri="log:mytest?level=ERROR&amp;showBody=false&amp;showCaughtExcepti
on=true&amp;showStackTrace=true" />
            </doCatch>
        </doTry>
    </route>

</camelContext>
```

The first bean, "mytestDecoder," defines the decoder bean, its implementing class, and in this case a constructor argument referencing the pluginName.

As noted previously, this creates a single instance of the class that will be used when referenced. Note that this bean also contains the attribute

```
depends-on="mytestPluginName"
```

This ensures that all beans in "mytest"-common.xml have been defined prior to this bean being created.
The second bean, "mytestDistRegistry," registers information about this plugin with the distribution service. In this case the pluginName and a message service queue are registered. This tells the distribution service that when data for the named plugin is received, that data should be placed on the specified queue so that the data may be routed to the plugin. Generically an "endpoint" is being defined which may be written to or read from.

The first bean of interest is "mytestPluginName." This bean is identified by name "id="mytestPluginName", the class that represents the value class="java.lang.String". The value that will be assigned to the bean is set using a "constructor-arg", that is, "constructor argument", which

identifies the argument type and value. Once this bean is created it is commonly used by using the form ref="name", where name is the **id** associated with the bean. This is defined so that all references for pluginName are the same.

The first item in the route "mytestIngestRoute" declares the "from" endpoint, that is, where the message comes from. In this case the "jms" queue endpoint is referenced.

Next, the message header is modified to add a property "pluginName" with the value "mytest." This property will serve to identify where the processing was performed for later use. Next, a "doTry" section is declared. This is set so that any errors that occur will be caught (by the doCatch tag) and appropriate action can be taken, writing an error message to the log file in this example.

The actions contained with the "pipeline" tag are then executed serially. The first "stringToFile" and the second "mytestDecoder" have been described. The third tag - "to" - sends the resulting message to the endpoint "directvm:indexAlert" for further processing.

A note of warning. The data placed in the body of the message is not checked and is presupposed to be correct. If the "decoder" bean were to return an array of String instead of PluginDataObject, no error would occur within this context. The error would occur, however, in downstream processing where camel finds bean method expecting an array of PluginDataObjects and is instead presented with an array of String. An exception would be thrown, indicating that a suitable conversion could not be made.

EDEX Data Routing

indexAlert Route

The "indexAlert" route is used as a possible destination for decoded data that needs to be stored to a database table. Specifically, this route exists to persist data to the "metadata" data base and to place alerts for that data on the notification queue. The processing for indexAlert takes place as follows.

When a decoder process has completed its decode, any resulting data is placed on the **indexAlert** queue. The indexAlert route receives data from that queue and passes it through the set of processing steps as shown in **Figure 3-1.**

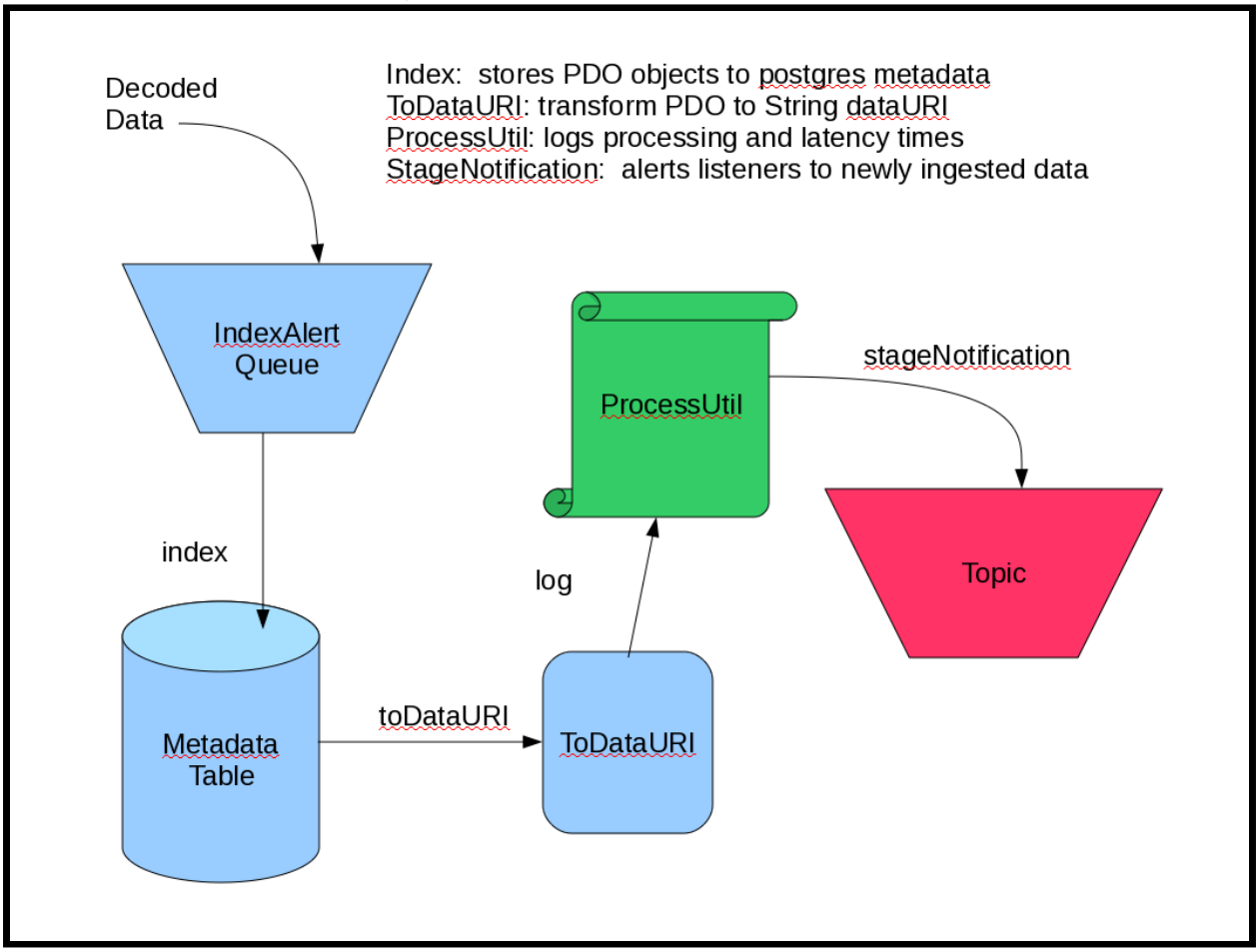

Figure 3-1. indexAlert Route

Thread Pools - Usage of Generic Decoder

Thread pooling in the EDEX system is derived from the Spring concurrent scheduling architecture, which is in turn built on the Java Concurrency API. See the following documentation.

- Spring Concurrent Scheduling Framework Documentation (http://static.springsource.org/spring/docs/2.0.x/api/index.html?overview-tree.html).
- Java Concurrency API Documentation (http://docs.oracle.com/javase/6/docs/api/index.html?java/util/concurrent/package-summary.html).

Threading allows various program execution units in the EDEX system to share time on the processors. How that sharing takes place is "hidden" by the API with the assumption that the API can best decide how to allocate resources to multiple tasks.

ThreadPools are defined within the Spring xml configuration files in the following manner.

```
  <bean id="mytestThreadPool" class="com.raytheon.uf.edex.esb.camel.JmsThreadPoolTaskExec
utor" >
    <property name="corePoolSize" value="3"/>
    <property name="maxPoolSize" value="3"/>
</bean>
```

In this example we set up a thread pool with given properties. By setting "core" and "max" PoolSize to the same value we indicate that this is the maximum number of threads that will be created for this pool. This threadPool is then set as a property on the JMS component for the plugin as follows

```
  <bean id="jms-mytest" class="org.apache.camel.component.jms.JmsComponent" >
    <constructor-arg ref="jmsIngestMyTestConfig"/>
    <property name="taskExecutor" ref="mytestThreadPool"/>
</bean>
```

Here the thread pool is injected into the plugin JMS Component as a custom executor for consuming messages from either queues or topics. Any data arriving on the component will be executed on the thread. The named JmsComponent is later used to create specific JMS inbound routes for data arriving at the plugin.
The primary purpose for this threading environment is to allow individual logging for various plugin components. Also it allows a separation of processing code. The Standard Hydrometeorology Exchange (SHEF) decoder, for example, writes its data to its own database. By using threading, this processing is kept separate from other decoders. Note that to keep the processing on the same processor, and thread of execution, subsequent endpoints must use the "directvm" endpoint.

To use the separate logging feature, an appender for the plugin is created. This appender is then associated with the specific thread pool. These actions are specific to the system Log4j configuration.

```
  <appender name="MyTestLog"
    <!-- define appender -->
</appender>

<appender name="ThreadBasedLog" class="com.raytheon.uf.edex.log.ThreadBasedAppender">
    <param name="ThreadPatterns" value="...;MyTestLog:mytestThreadPool .*;..."/>
        <!-- Other definition itemsr -->
        <appender-ref ref="MyTestLog"/>
</appender
```

Camel EDEX Adapters

Important Camel-EDEX Classes

The purpose of these classes is to provide some common utility functions that are used when processing data within the Camel ESB. In addition, several of these classes define an adapter layer that decouples EDEX from the underlying Camel infrastructure.

The Exchange interface defines a container that is used to transfer data between various Camel components.

DataUriAggregator

- Method: addDataUris
- Input: Array of String - An array of URIs to add to the collection.
- Output: void

Add one or more URIs to the internal collection of URIs.

- Method: hasUris
- Input: Object - Not used.
- Output: boolean

Does the aggregation contain any URIs?

- Method: sendQueuedUris
- Output: DataURINotificationMessage

Create a message containing the aggregated URIs.

- Method: sendPracticeQueuedUris
- Output: PracticeDataURINotificationMessage

Create a practice message containing the aggregated URIs.

FileToBytesConverter

- Method: toByteArray
- Input: File
- Output: Array of byte

Read an entire array of byte from a specified File reference.

FileToString

- Method: process
- Input: Exchange
- Output: void

Copy an incoming file from its location into the edex "../data/processing" directory structure. The FileToString class assumes that the incoming message contains a Java File reference to some existing file on the file system. The process method creates the new path in the "processing" structure if necessary, then performs a binary copy operation of the contents of the file. When the copy is complete the original file is deleted from the file system. The **dequeueTime** message property is set to the current system time in milliseconds. This time is used to provide timing instrumentation.

MessageProducer

- Method: sendAsync
- Input:
    - String - endpoint - Name of the endpoint to receive the object.

    - Object - message - The object to send.
- Output: void

- Method: sendSync
- Input:
    - String - endpoint - Name of the endpoint to receive the object.
    - Object - message - The object to send.
- Output: void

- Method: sendAsyncUri
- Input:
    - String - uri - The URI of the endpoint to receive the object.
    - Object - message - The object to send.
- Output: void

Allow objects to be sent to Camel endpoints programmatically. The two send methods, sendASync and sendSync, send a message to the endpoint. The sendSync will expect a reply from the endpoint, whereas the sendASync is "fire and forget." The sendAsyncURI sends to an endpoint URI instead of a named endpoint.

NotifySeparator

- Method: separate
- Input:
    - String - header - derived from the message header property.
    - Long - queueTime - derived from the message enqueueTime property.
    - String - body - derived from the message body.
- Output: List<Message>

Assumes that "header" is a concatenated list of notifications and that body is a concatenated list of locations. The separate method splits the concatenated lists and creates a list of Camel Messages, one for each member of the lists.

ProcessUtil

- Method: delete
- Input: String - derived from the "ingestFileName" message header property.
- Output: void

Delete the file referenced by the ingestFileName.

- Method: delete
- Input: File - A file reference to delete.
- Output: void

Delete the specified file.

- Method: log
- Input: Map - derived from the message headers. Contains various properties that are collected during processing.
- Output: void

Log information derived from various header properties that may have been set during processing. Used to generate timing statistics.

- Method: iterate
- Input: Array of PluginDataObject
- Output: Iterator An iterator to the array.

Convenience method to return an iterator to an array of PluginDataObjects. Normally used within Camel "wiring" to get items one at a time.

SetIngestHeaderFields

- Method: process
- Input: Exchange - The message container used for transport data.
- Output: void

On entry to the process method the **dequeueTime** message property is set to the current system time in milliseconds. This time is used to provide timing instrumentation. The body of the message, assumed to be a String, is read and used to create a File reference. The resulting File is checked to see if it exists on the file system. On success the "ingestFileName" property to the fully qualified path name of the file. If the file does not exist an error log message is posted and the fault flag on the outgoing message is set to **true**.

StringToFile

- Method: process
- Input: Exchange
- Output: void

The StringToFile class assumes that the incoming message contains either an array of byte or a String as its payload. In the case of a byte array it is further assumed that this data represents a String. The String object should be a fully qualified file path to some file of interest. A Java File object is created using the resulting String data and this File is then tested to determine if it exists. If it does exist, the body of the Exchange message is set to the File object. In addition, the following message properties are set.

- "ingestFileName" is set to the name of the File object created.
- dequeueTime is set to the current system time in milliseconds. This time is used to provide timing instrumentation.
- If the file represented by the input String was not found an error is logged and the fault flag on the outgoing message is set to **true**.

ToDataURI

- Method: toDataURI
- Input: Array of PluginDataObject
- Output: Array of String

The ToDataURI class converts an array of PluginDataObject and converts these to their DataURI. This function is primarily used as a step prior to notification. The notification subsystem notifies interested subscribers of new data by broadcasting the datauri only.

UUIDGenerator

- Method: generateUUID
- Output: String

Generates a random Universally Unique IDentifier (UUID) and returns the String representation of that identifier.

EDEX Camel Spring

EDEX is built upon the Apache Camel framework and is configured through the use of Spring XML files. Plugins contribute their Spring XML files inside the plugin in the res/spring folder.

EDEX Modes

Each EDEX instance can selectively start different services and plugins through the use of a command line argument to start.sh that designates the edex mode.

There are currently four supported EDEX modes, and each server starts all four, resulting in four distinct JVMs.

1. **request.** Serves http for CAVE and other clients.
2. **ingest.** Processes most of the backend work, including ingesting and storing most data types.
3. **ingestGrib.** Decodes and stores grib data that is received. This was separated from the ingest JVM to free up memory for the other ingest process.
4. **ingestDat.** Calculates and stores data types for the DAT plugins, such as FFMP and Scan.

Too many JVMs can result in the server going into memory swap. However, splitting the functionality into separate JVMs can improve efficiency and how much is affected by a failover scenario.

The services that are started with each mode are determined by the modes.xml file. As an example, here is the entry for the request mode.

```
  <mode name="request">
     <include>.*request.*</include>
     <include>.*common.*</include>
</mode>
```

When you start edex with *start.sh request*, edex scans all available plugins for files under the res/spring folder. Then, any of those files that match the regular expressions of <include> tags and do not match the regular expressions of <exclude> tags will be loaded.

Spring XML Files

To understand how EDEX uses Spring, it is helpful to understand some of Spring itself. For documentation on Spring, please see http://static.springsource.org/spring/docs/2.5.x/reference/index.html (http://static.springsource.org/spring/docs/2.5.x/reference/index.html).

An EDEX Spring XML file must always start and end with the <beans> tag that defines where the schemas are located. EDEX typically uses the Spring schema and the Camel schema.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">
```

Within the beans tag, plugins typically instantiate beans that will be used in service routes and register with specific services. For some examples:

```
<bean id="satNotifyTransform"
       class="com.raytheon.edex.plugin.satellite.notify.SatelliteNotifyTransform" />
```

This XML tag instantiates the class SatelliteNotifyTransform and identifies it for later use as satNotifyTransform.

```
<bean id="gribRegistered" factory-bean="pluginRegistry" factory-method="register"
      depends-on="levelRegistered">
   <constructor-arg value="grib"/>
   <constructor-arg ref="gribProperties"/>
</bean>
```

This XML tag calls PluginRegistry.register(grib, gribProperties), and it will not be called until the levelRegistered bean and gribProperties bean are created. Since gribProperties is passed into this call, it will be created before this call. levelRegistered is not passed into the method, but the developer determined it was necessary to occur before gribRegistered, hence the use of the depends-on attribute.

```
<camelContext id="sat-camel"
      xmlns="http://camel.apache.org/schema/spring"
      errorHandlerRef="errorHandler"
      autoStartup="false">
```

The camel contexts are the services of EDEX. Contexts consist of one or more routes.

```
<route id="satNotification">
   <from uri="vm:satNotify" />
   <bean ref="satNotifyTransform" method="transformToMessages" />
   <bean ref="serializationUtil" method="transformToThrift" />
   <to uri="jms-generic:queue:satNotification" />
</route>
```

Routes consist of a from uri, processing in the middle, and then a to uri. To understand specific options for a route, you should consult the Camel documentation at http://camel.apache.org/ (http://camel.apache.org/). In the above example, messages are pulled from the vm:satNotify queue, they are sent through the previously instantiated satNotifyTransform's transformToMessages method, then passed to serializationUtil.transformToThrift(), and finally are sent out through JMS to another queue.

Multiple routes can be chained together with the use of from and to uri's. As an example, all incoming data gets routed to the distribution service, then routed to the individual plugin's decoding service, then the persist service, then the index service, and finally the notify service. So when a developer goes to add a new data type, they only have to hook in the decoding service into the appropriate spot in the chain and the rest will work automatically.

Data Access Framework

The Data Access Framework allows developers to retrieve different types of data without having dependencies on those types of data. It provides a single, unified data type that can be customized by individual implementing plugins to provide full functionality pertinent to each data type.

Writing a New Factory

Factories will most often be written in a dataplugin, but should always be written in a common plugin. This will allow for clean dependencies from both CAVE and EDEX.

A new plugin's data access class must implement IDataFactory. For ease of use, abstract classes have been created to combine similar methods. Data factories do not have to implement both types of data (grid and geometry). They can if they choose, but if they choose not to, they should do the following :

```
throw new UnsupportedOutputTypeException(request.getDatatype(), "grid");
```

This lets the code know that grid type is not supported for this data factory.
Depending on where the data is coming from, helpers have been written to make writing a new data type factory easier. For example, PluginDataObjects can use AbstractDataPluginFactory as a start and not have to create everything from scratch.

Each data type is allowed to implement retrieval in any manner that is felt necessary. The power of the framework means that the code retrieving data does not have to know anything of the underlying retrieval methods, only that it is getting data in a certain manner. To see some examples of ways to retrieve data, reference SatelliteGridFactory and RadarGridFactory.

Methods required for implementation :

- **public DataTime[] getAvailableTimes(IDataRequest request)** - This method returns an array of DataTime objects corresponding to what times are available for the data being retrieved, based on the parameters and identifiers being passed in.
- **public DataTime[] getAvailableTimes(IDataRequest request, BinOffset binOffset)** - This method returns available times as above, only with a bin offset applied.
  Note: Both of the preceding methods can throw TimeAgnosticDataException exceptions if times do not apply to the data type.
- **public IGridData[] getGridData(IDataRequest request, DataTime... times)** - This method returns IGridData objects (an array) based on the request and times to request for. There can be multiple times or a single time.
- **public IGridData[] getGridData(IDataRequest request, TimeRange range)** - Similar to the preceding method, this returns IGridData objects based on a range of times.
- **public IGeometryData[] getGeometryData(IDataRequest request, DataTime times)** - This method returns IGeometryData objects based on a request and times.
- **public IGeometryData[] getGeometryData(IDataRequest request, TimeRange range)** - Like the preceding method, this method returns IGeometryData objects based on a range of times.
- **public String[] getAvailableLocationNames(IDataRequest request)** - This method returns location names that match the request. If this does not apply to the data type, an IncompatibleRequestException should be thrown.

Registering the Factory with the Framework

The following needs to be added in a spring file in the plugin that contains the new factory:

```
 <bean id="radarGridFactory" class="com.raytheon.uf.common.dataplugin.radar.dataaccess.RadarGridFactory" />

<bean factory-bean="dataAccessRegistry" factory-method="register">

    <constructor-arg value="radar"/>

    <constructor-arg ref="radarGridFactory"/>

</bean>
```

This takes the RadarGridFactory and registers it with the registry and allows it to be used any time the code makes a request for the data type "radar."

Retrieving Data Using the Factory

For ease of use and more diverse use, there are multiple interfaces into the Data Access Layer. Currently, there is a Python implementation and a Java implementation, which have very similar method calls and work in a similar manner. The Java implementation is primarily for use by CAVE. Local applications should use the Python interface. Plugins that want to use the data access framework to retrieve data should include **com.raytheon.uf.common.dataaccess** as a Required Bundle in their MANIFEST.MF.

To retrieve data using the Python interface :

```
 from ufpy.dataaccess import DataAccessLayer

req = DataAccessLayer.newDataRequest()

req.setDatatype("grid")

req.setParameters("T")

req.setLevels("2FHAG")

req.addIdentifier("info.datasetId", "GFS212")

times = DataAccessLayer.getAvailableTimes(req)

data = DataAccessLayer.getGridData(req, times)
```

To retrieve data using the Java interface :

```
 IDataRequest req = DataAccessLayer.newDataRequest();

req.setDatatype("grid");

req.setParameters("T");

req.setLevels("2FHAG");

req.addIdentifier("info.datasetId", "GFS212");

DataTime[] times = DataAccessLayer.getAvailableTimes(req)

IData data = DataAccessLayer.getGridData(req, times);
```

newDataRequest() - This creates a new data request. Most often this is a DefaultDataRequest, but saves for future implentations as well.

setDatatype(String) - This is the data type being retrieved. This can be found as the value that is registered when creating the new factory (See Registering the Factory with the Framework above [radar in that case]).

setParameters(String...) - This can differ depending on data type. It is most often used as a main difference between products.

setLevels(String...) - This is often used to identify the same products on different mathematical angles, heights, levels, etc.

addIdentifier(String, String) - This differs based on data type, but is often used for more fine-tuned querying.

Both methods return a similar set of data and can be manipulated by their respective languages. See DataAccessLayer.py and DataAccessLayer.java for more methods that can be called to retrieve data and different parts of the data.

Because each data type has different parameters, levels, and identifiers, it is best to see the actual data type for the available options. If it is undocumented, then the best way to identify what parameters are to be used is to reference the code.

Python Job Coordinator

The PythonJobCoordinator allows developers to easily create thread pools to execute Python code on separate threads. Jobs can be run either asynchronously (in which a listener is fired when the job is done) or synchronously (in which the calling code waits for the job to finish before moving on). Jobs will be queued as necessary, so if a pool of three is allocated and five jobs are queued, three will run simultaneously, with two running after those finish (in an order). Code can allocate as many or as few threads as desired, although starting with a single thread is recommended.

The following code is necessary to execute an existing PythonInterpreter class using this functionality. Implementation of PythonInterpreter is not demonstrated by the following code. For a basic example, look in the com.raytheon.viz.gfe.query package in the com.raytheon.viz.gfe plugin.

- QueryScript.java. This class is the interface to the Python interpreter. It creates the Jep instance, and has an execute method that will be called to run your Python code. This class will be executed on a thread from the thread pool, and has any functionality that is necessary while integrating with Python. This should be a subclass of PythonInterpreter or PythonScript.
- QueryScriptExecutor.java. This class will be instantiated every time a user wants to call a method on QueryScript. Arguments should be added to the constructor of this class that are necessary to be used in the QueryScript itself. The execute method takes the QueryScript and runs any method on it. In this case, there is only a single executor, but it is possible to have multiple Executor classes to do different things on the same PythonInterpreter class. The Executor classes must implement IPythonExecutor<I, O> where I is the QueryScript class and O is what the user expects back from the method being called.
- QueryScriptFactory.java. This class instantiates the QueryScript itself. The constructor should take anything necessary to know about in the QueryScript itself, and needs to define the name of the thread pool and how many threads should be allocated. These threads do not go away, and are only used for this procedure, so this needs to be taken into account when thinking of the number of threads. The createPythonScript() method builds a new PythonInterpreter that does not go away.

To get the following to run:

1. The first thing to do is to declare a new factory. This only needs to be done once, so it should only be constructed in a place that gets called once (for example, constructor).

   ```
   AbstractPythonScriptFactory<QueryScript> factory = new QueryScriptFactory(dataManager);
   ```

2. The PythonJobCoordinator now needs to create a new thread pool from that factory, and this should be stored off. If this is not possible, there is a getInstance() method by the name that is given in the factory.

   ```
   PythonJobCoordinator coordinator = PythonJobCoordinator.newInstance(factory);
   ```

   OR

   ```
   PythonJobCoodinator coordinator = PythonJobCoordinator.getInstance("factoryname");
   ```

3. Now, we need to declare new Executors, create a listener (if desired), and submit them to the coordinator.

```
 IPythonExecutor<QueryScript,ReferenceData> executor = new QueryScriptExecutor("evaluat
e", argMap);

IPythonJobListener<ReferenceData> listener = new IPythonJobListener<ReferenceData>() {

    @Override
    public void jobFailed(Throwable e) {
        statusHandler.handle(Priority.ERROR,
                "Unable to finish QueryScript job", e);

    }

    @Override
    public void jobFinished(ReferenceData result) {
        getActiveRefSet();
        if(!result.getGrid().equals(getActiveRefSet().getGrid())) {
            setActiveRefSet(result);
        }
    };
};

coordinator.submitAsyncJob(executor, listener);
```

OR

```
 ReferenceData data = coordinator.submitSyncJob(executor);
```

- In the first case, we are going to continue on with everything and the listener will get fired depending on whether it failed (jobFailed) or worked (jobFinished).
- In the second case, the code waits until the QueryScript job is done.
- The first case is the desired and recommended approach, as nothing will lock up when that is called.

IDataStore

IDataStore is the interface for storing and accessing raw data that has been decoded. The current implementations are PypiesDataStore and CachingDataStore; the other implementations are legacy implementations that have been superseded by the PypiesDataStore. Currently IDataStore implementations store to hdf5, but in theory the storage format could change in the future.

To access a file, you must first create the IDataStore object by using DataStoreFactory.getDataStore(File file). To add data to the file, you use IDataStore.addRecord(IDataRecord) and then call IDataStore.store(). To retrieve data, you can use any of the retrieve methods on IDataStore. If you wish to retrieve only a column, row, or selected set of points, you must use a retrieve method that takes a Request object.

PyPIES

PyPIES is **Py**thon **P**rocess **I**solated **E**nhanced **S**torage. It is pronounced like Py as in Python followed by the plural form of a dessert. PyPIES is designed to push all of our hdf5 actions into transactions that are isolated to a unique process. In short, every time you read or write to hdf5 a separate and dedicated process will handle that action at the hdf5 API level. You can think of it as a separate hdf5 service, much like postgres runs as a service and handles sql commands, PyPIES runs as a service and handles IDataStore commands.

Advantages:

- EDEX cannot crash in the hdf5 libraries
- Removed bottleneck of single-thread per process on hdf5 access
- HDF5 service and file store can reside on different machine than EDEX
- Python DynamicSerialize developed to support this

Disadvantages:

- All data must be serialized and sent to the service before it can be written

PyPIES Architecture

PyPIES is built on the following components:

- **Apache Http Server (httpd).** For serving http requests and returning responses.
    1. **mod_wsgi.** Module for httpd to serve python
    2. **werkzeug.** Python package that implements wsgi
- **dynamicserialize python package.** For serializing/deserializing requests and responses
- **h5py.** For writing/reading hdf5
- **pypies python package.** Glue that deserializes requests and processes data with h5py, then serializes responses.

Starting PyPIES

To start pypies, as root do `/etc/init.d/httpd-pypies start`

Configuring PyPIES

Pypies primary config files are apache config files. The most important is located at **httpd_pypies/etc/httpd/conf.d/pypies.conf**. This allows you to configure the number of processes that pypies will have available to process IDataStore requests. The other important file is at **httpd_pypies/var/www/wsgi/pypies.wsgi**. If either of these files has incorrect paths to python and its packages, pypies will probably not start. The last config file for apache is at **httpd_pypies/etc/httpd/conf/httpd.conf**. This files controls a number of options including port number.

EDEX is configured to use the pypies server address specified in **edex/bin/setup.env**.

You can also configure the number of connections an EDEX instance can have open to pypies at any given time, and the timeout value. The number of connections edex can open to pypies is currently set in the file **edex/etc/default.sh**. If you would like different EDEX Java Virtual Machine

(JVM) to have a different number of connections, set the value in the appropriate .sh file. The timeout value is set in edex.xml in the pypiesStoreProps bean.

PyPIES Logs

PyPIES logs to two different locations. For top level apache issues, it logs to apache's error_log which you can find at **httpd_pypies/etc/httpd/logs/error_log**. The python code that processes requests and returns responses is logging to **awips2/pypies/logs/pypies.log** by default.

Due to the nature of multiple processes attempting to write to the same file, pypies starts a separate logging service. Each individual process will log to a socket, while this separate logging service reads from the socket and writes to the file, rolling over the file at midnight each day.

Python

Python can be used in two ways, either through Java or through the command line. When using python from Java, it is a hybrid of Java and python objects; whereas from the command line, it is purely python. This section will only cover the use through Java.

The interface from Java to Python is built on Java Embedded Python (JEP). The best way to call Python from Java is to use the class PythonScript or other derivatives of the class PythonInterpreter. The javadoc explains the methods on those classes. Each separate instance of a PythonInterpreter object results in a separate, mostly sandboxed python interpreter.

Gotchas

- All access to a Python interpreter must be on the same thread that created the interpreter.
- You should call dispose() when finished with an interpreter to free up memory.
- Importing numpy in python leaks memory, so you should reuse interpreters that import numpy instead of disposing them.
- Some Python extensions may not work well in multiple interpreters (e.g. h5py).

Python/Java Code

Inside an interpreter, JEP will automatically convert primitives and Strings between Java and Python, but otherwise every Java object is wrapped in a python object known as a PyjObject. PyjObjects can be used throughout python code just like any other python object. You can also call the Java methods on a PyjObject, however, any arguments passed to them must be primitive objects or other PyjObjects.

To import a Java class, you use the from syntax:

```
from com.raytheon.uf.common.dataplugin.gfe.reference import ReferenceID, ReferenceData
```

This imports the classes ReferenceID and ReferenceData as PyjClasses. PyjClasses implement python's __call__ method, so to create an instance and call the Java constructor you just call the pyjclass, such as

```
instance = ReferenceID()
```

Transforming Between Java Arrays and Numpy Arrays

To send a primitive Java array to python, the best way is to implement the Java interface INumpyable. The implementation should provide the x and y dimensions of the numpy array, and then getNumPy() returns an Object[] of all the primitive arrays to return.

In python code, with a PyjObject of the INumpyable implementation, simply do

```
x = obj.__numpy__
```

where obj is the INumpyable PyjObject, and x will be a python list of numpy arrays corresponding to the primitive arrays returned by getNumPy().
To send numpy arrays back to Java, ensure the Java method you are calling accepts the appropriate primitive array as an argument, and simply pass it along.

Point Data

Point data is used when each data record describes data at a single point (lon, lat). Point data is designed to hold many parameters for each point with each point having the exact same parameters. Often the parameters describe a single value such as surface temperature, or humidity, but it can also be used to hold multiple values for data throughout the atmosphere.

PointDataContainer

The primary way to interact with point data is through a PointDataContainer. A PointDataContainer holds all the parameters for several different points. Most often, when interacting with a PointDataContainer you will want to use a PointDataView. A PointDataView provides a view into the container for a single record. On edex when you decode you will use PointDataContainer.append to get a view which you will populate with data. On CAVE you will request a container with the parameter you use and you will use PointDataContainer.readRandom to iterate over the data records in the container.

PluginDataObject

For point data plugins the PluginDataObject must implement IPointData. This interface is just a getter and a setter for a PluginDataView. This view will be set in the decoder and then retrieved by the Dao to store in HDF5. Usually a point data object will also have a location object with a latitude and longitude that is stored in the dataURI for querying. Everything else about the PluginDataObject should be similar to other data plugins.

Dao

The Dao for a point data plugin must extend PointDataPluginDao. This class handles all of the storage and retrieval of point data to hdf5. The PointDataPluginDao also provides a means for loading the default descriptions for your data.

Descriptions

There are two description files for each point data plugin that describe what parameters are stored.

HDF5 Data Description

The HDF5 data description for a point data plugin is found in res/pointdata/{pluginName}.xml. This file contains a description of each field stored in hdf5 for the points in the data. Here is an example:

```
 <pointDataDescription>
   <parameter name="temperature" numDims="1" type="FLOAT" unit="K" />
   <parameter name="dewpoint" numDims="1" type="FLOAT" unit="K" />
   <parameter name="windSpeed" numDims="1" type="FLOAT" unit="m/s" />
   <parameter name="windDir" numDims="1" type="FLOAT" unit="degree" />
   <parameter name="windGust" numDims="1" type="FLOAT" unit="m/s" />
</pointDataDescription>
```

Each parameter has several fields.

- **name.** The name for the parameter, this is how it is referenced throughout the system
- **numDims.** If there is one data value for each point this is one, if there are multiple values(for instance for different elevations), then this can be 2.
- **type.** STRING,LONG,FLOAT,DOUBLE,INTEGER.
- **unit.** The data units for this data, these can be used for automatic conversion later, this should be parseable by the javax.measure.units parser.

DB Data Description

The db data description for a point data plugin is usually found in res/pointdata/{pluginName}db.xml. This file contains a description of each field stored in the database that can be requested through a point data query. Here is an example:

```
 <pointDataDbDescription>
    <parameter name="latitude" queryName="location.latitude" type="FLOAT" unit="°" />
    <parameter name="longitude" queryName="location.longitude" type="FLOAT" unit="°" />
    <parameter name="elevation" queryName="location.elevation" type="FLOAT" fillValue="-
9999" unit="m" />
    <parameter name="stationId" queryName="location.stationId" type="STRING" />
    <parameter name="reportType" queryName="reportType" type="INT" />
    <parameter name="corIndicator" queryName="corIndicator" type="STRING" />
    <parameter name="dataURI" queryName="dataURI" type="STRING" />
</pointDataDbDescription>
```

Each parameter has several fields:

- **name.** The name for the parameter, this is how it is referenced throughout the system
- **queryName.** The query string to use for hibernate when requesting this parameter.
- **type.** STRING,LONG,FLOAT,DOUBLE,INTEGER
- **unit.** The data units for this data, these can be used for automatic conversion later, this should be parseable by the javax.measure.units parser.
- **fillValue.** This value can be used for numeric types which may store null in the database, but null is invalid in the Point Data Container.

Decoder

The decoder for a point data plugin is the same as for any other plugin except it stores data in a PointDataView. The decoder needs to create new PointDataContainers to store the data in, this can be easily done using the description which can be retrieved from the dao. Here is an example of how a PluginDataObject is created within a decoder:

```
 PluginDao pluginDao = PluginFactory.getInstance()
        .getPluginDao(pluginName);
PluginDataObject record = pluginDao.newObject();
PointDataContainer pdc = PointDataContainer.build(pluginDao
        getPointDataDescription(null));
PointDataView pdv = pdc.append();
record.setPointDataView(pdv);
/* populate metadata in record */
/* populate data in the pdv */
pdv.setFloat("temperature", sampleMethodToParseTemperature());
pdv.setFloat("dewpoint", sampleMethodToParseDewpoint());
```

Requesting Data on CAVE

The PointDataRequest class provides static methods for retrieving point data on CAVE. The method most often used is requestPointDataAllLevels. The javadoc for this method describes how to use this method and what to pass in. This method returns a PointDataContainer. Processing of this container is usually done in a loop similar to this:

```
 PointDataContainer pdc = PointDataRequest
        .requestPointDataAllLevels(time, plugin, parameters,
        null, requestConstraints);
for (int uriCounter = 0; uriCounter < pdc.getAllocatedSz(); uriCounter++) {
    PointDataView pdv = pdc.readRandom(uriCounter);
    float latitude = pdv.getFloat("latitude");
    float longitude = pdv.getFloat("longitude");
    // do something with the point data view here.
}
```

PlotResource2

It is possible to display any point data type on CAVE in PlotResource2 with very little java code. In order for a new plugin to work with PlotResource2 there are a few constraints on the PluginDataObject. The PluginDataObject must have a field called location that contains a location object. An existing object such as SurfaceObsLocation a custom object can be created to hold the location. This object must contain a latitude, longitude, and stationId. Additionally, these fields must be included in the dataURI for the pluginDataObject.

To display a plot an svg file is needed. This can be seen in the existing svg files in the plotModels directory in the localization perspective. The most important thing to understand is that the plotParam attribute on a text element specifies the name of a parameter directly from the EDEX description files. The important part of a plotModel file is what is within the symbol element. Here is a very simple example.

```
 <symbol
   overflow="visible" id="plotData" class="info">
   <text
       id="lat" plotMode="null" class="text" plotParam="latitude" x="0" y="0">0
   </text>
   <text
       id="lon" plotMode="null" class="text" plotParam="longitude" x="0" y="0">0
   </text>
   <text
       id="myParam" plotMode="text" plotParam="myParam" x="0px" y="0px">75
   </text>
</symbol>
```

This symbol element has three text elements. The first two are lat and lon. Notice that these have a plotMode of null which means nothing will be displayed; these two are just used so that PlotResource2 knows where to put the plot on a map. The third element plots the value of the parameter myParam at the center of the plot. To display other parameters, add more elements similar to the one for myParam. By changing the x and y values this changes where the different parameters appear in the plot.

Creating a New PluginDataObject Derived Class

Description of the PluginDataObject Base Class

All data objects derive from the **com.raytheon.uf.common.dataplugin.PluginDataObject** abstract base class. This class defines the minimum functionality for any PluginDataObject (PDO) that may be persisted to a database. In particular the class exposes the following attributes:

- **Id.** A unique record identifier automatically generated by the database.
- **dataUri.** A construction that uniquely identifies this data object within a plugin table. In the base PluginDataObject, the **dataUri** consists of the pluginName and the dataTime.
- **pluginName.** Short name of the plugin name. For example using the following plugin project name [**com.raytheon.uf.edex.plugin.myplugin**], the plugin short name is "*myplugin*".
- **dataTime.** The time that should be associated with this data object. The most common dataTime for point data is the time that the data was observed. A validTime for forecast data is also common.
- **insertTime.** The time that the data object was inserted into the database.
    - Currently the insertTime, if set, will be overwritten with the current system time if the PDO is not an IPersistable. If the instance is IPersistable, then the insertTime will remain as set.
- **messageData.** The raw data that was used to generate the data object. This attribute value is not required and may be left null.

When examining the PluginDataOject class and its associated database fields the developer may note that the **id** attribute is the only field indicated as NOT NULL. Even so the pluginName is used explicitly as the base of the dataURI and the dataTime attribute is declared as the first item in the dataURI. If these values are null then no error will occur, however a useless dataURI of "/null/null" will be generated.

Figure 2-5 shows the class diagram showing the PluginDataObject, its ancestors, and a possible descendant.

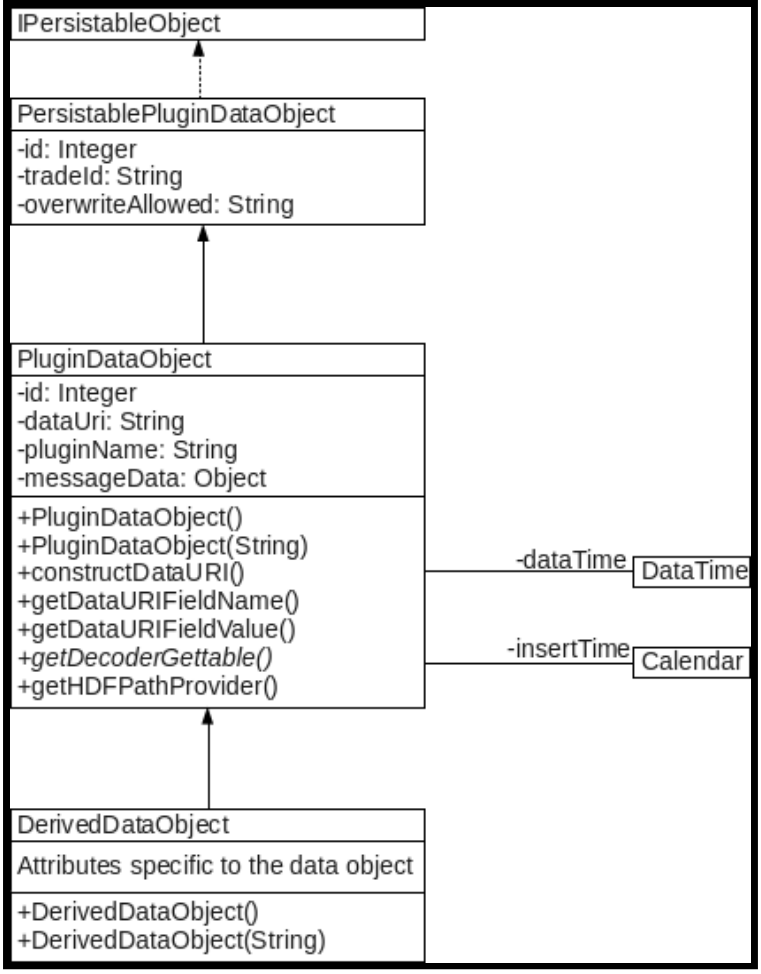

Figure 2-5. PluginDataObject Class Hierarchy

Description of a Minimally Derived PluginDataObject

The PluginDataObject also exposes methods for constructing the dataUri, retrieving dataUri fields by column identifier, as well as getter and setter methods for the above attributes. The following assume a minimally derived PDO named FooRecord that exposes a single Integer attribute, **reportType**.

```
 @Entity
@Table(name = "foo", uniqueConstraints = { @UniqueConstraint(columnNames =   { "dataURI"
}) })
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
@DynamicSerialize
public class FooRecord extends PluginDataObject {

    private static final long serialVersionUID = 1L;

    /**
     * A report type that identifies this instance.
     */
    @DataURI(position = 1)
    @Column
    @DynamicSerializeElement
    @XmlAttribute
    private Integer reportType;

    /**
     * Construct an empty instance of this class.
     */
    public FooRecord() {
    }

    /**
     * Constructor for DataURI construction through base class. This is used by
     * the notification service.
     * @param uri
     *          A dataURI applicable to this class.
     */
    public FooRecord(String uri) {
        super(uri);
    }

    /**
     * Get the report type that identifies this instance.
     * @return The reportType identifying the instance.
     */
    public Integer getReportType() {
        return reportType;
    }

    /**
     * Set the report type to identify this instance.
     * @param reportType
     *            The reportType to set. Any not-null Integer value.
     */
    public void setReportType(Integer reportType) {
        this.reportType = reportType;
    }

    /**
     * Get the IDecoderGettable reference for this record. This class returns
     * a null reference indicating that the class does not implement the
     * IdecoderGettable interface.
     * @return The IDecoderGettable reference for this record.
     */
    @Override
    public IDecoderGettable getDecoderGettable() {
        return null;
```

```
    }
}
```

Lines 1 through 5 are preamble that guide the persistence mechanism.

- **@Entity.** This annotation indicates that this class should be persisted to a database. Any class marked with this annotation must expose an empty (no argument) constructor, may not be final, and must define a primary key. Note that the primary key **id**, for a class is implicit by extending the PluginDataObject base class.
- **@Table.** Indicates that this is the primary table for this Entity. This annotation is required so that the uniqueConstraint on dataURI may be declared.
- **@XmlRootElement.** Defines this class as the root element for an XML tree containing this class' data.
- **@XmlAccessorType.** This annotation provides control over the default serialization of properties and fields in this class. The use of XmlAccessType.NONE indicates that no fields or properties will be bound unless they are specifically annotated.
- **@DynamicSerialize.** Indicates that this class should be serialized.

Line 6 tells the compiler that we are extending the PluginDataObject base class. All PDOs must extend this base class. Other functionality may be exposed by implementing one or more of the following interfaces.

- **IPersistable.** PDOs implementing this interface indicate to the persistence layer that the data in this class will be stored to an HDF repository. In addition this interface exposes methods that allow clients to retrieve the persistence time for this object as well as utility methods that aid in storing to the HDF repository.
- **IDecoderGettable.** Exposes methods that allow class attribute values and associated units to be read using parameter names.
- **ISpatialEnabled.** Exposes the "**getSpatialObject**" method which indicates that this object is locatable in some reference frame.
- **IPointData.** Expose methods allow instance data to be stored using the PointData functionality.

Lines 13 through 17 declare an attribute that will be used to store a "report type" as an Integer value. Several Java annotations are used in this declaration as follows.

- **@DataURI**. This annotation flags an attribute so that it is used as part of the datauri construction. This annotation takes two arguments;
- "**position**" is mandatory and gives the attribute's physical position within the constructed datauri. Note that positions start at 1. Position 0 is reserved for the DataTime contained in the PluginDataObject base class.
- "**embedded**" is optional and defaults to "**false**". A value of "true" indicates that this attribute is a composite object that will contribute its attributes to the datauri being constructed.
- **@Column**. This annotation is used to specify that this attribute will be mapped to a field within the table declared in line 2. All arguments are optional with "length," "nullable," and "unique" being common.
- **@DynamicSerializedElement.** Flag that indicates that this attribute will be serialized. An assumption is made that proper setter and getter methods exist for the attribute. The annotation takes no arguments.
- **@XmlAttribute**
- **@XmlElement.** These annotations control the mapping of class attributes to/from XML marshalling using the JAXB API.

The default no-argument constructor required by the @**Entity** annotation is defined on line 22 and line 31 defines a constructor which takes a dataURI as an argument. The setter and getter methods for the class attribute are defined in lines 35 to 50 and finally the **getDecoderGettable** method is defined for this class in lines 58 to 61.

Common Usage

The following code shows a common idiom for creating and using a PDO. Using the derived PDO above:

```
  public PluginDataObject [] decode(String data) {
    PluginDataObject [] retData = null;

    // Create a new instance of the PluginDataObject
    FooRecord record = new FooRecord();

    // Assign data to the fields
    record.setPluginName("foo");
    record.setReportType(10);
    // Note that the DataTime constructor requires a java.util.Date
    record.setDataTime(new DataTime(new java.util.Date());

    // Now construct the dataURI
    try {
        record.constructDataURI();
    } catch(PluginException pe) {
        // do something with the invalid record, set it to null here
        record = null;
    }
    if(record != null) {
        // create an array containing the record.
        retData = new PluginDataObject[] { record, };
    } else {
        // create an empty array
        retData = new PluginDataObject[0];
    }
    // And pass the return data to the caller.
    return retData;
}
```

Also common is using a parser class to decode the data and pass back a populated array of data objects, similar to the following

```
  Parser p = new Parser(data);
PluginDataObject [] retData = p.parseData();
return retData;
```

There are of course many variations on the above themes. In the second case we would expect that the PluginDataObject has been fully populated by the Parser class.

Creating Derived Class

There are currently two methods for creating a derived PDO class. The first is to copy an existing class and then modify the class definition. This can be useful however the entire process can be error prone. The second method is to use the "mkPlugin.sh" shell script. This script will create a skeleton for all parts of a data plugin using the current coding standards and conventions.

```
  mkPlugin.sh pluginName PluginName
```

For example:

```
  mkPlugin.sh foo Foo
```

will generate the following:

```
 base decoder project directory
    com.raytheon.uf.edex.plugin.foo
created files
    com.raytheon.uf.edex.plugin.foo/.project
    com.raytheon.uf.edex.plugin.foo/.classpath
    com.raytheon.uf.edex.plugin.foo/.settings
    com.raytheon.uf.edex.plugin.foo/build.properties
    com.raytheon.uf.edex.plugin.foo/component-deploy.xml
    com.raytheon.uf.edex.plugin.foo/bin
    com.raytheon.uf.edex.plugin.foo/META-INF/MANIFEST.MF

    com.raytheon.uf.edex.plugin.foo/src/com/raytheon/uf/edex/plugin/foo/FooDecoder.java
    com.raytheon.uf.edex.plugin.foo/utility/edex_static/base/distribution/foo.xml

    com.raytheon.uf.edex.plugin.foo/res/spring/foo-ingest.xml
    com.raytheon.uf.edex.plugin.foo/res/spring/foo-common.xml

base dataplugin project directory
    com.raytheon.uf.common.dataplugin.foo
created files
    com.raytheon.uf.common.dataplugin.foo/.project
    com.raytheon.uf.common.dataplugin.foo/.classpath
    com.raytheon.uf.common.dataplugin.foo/.settings
    com.raytheon.uf.common.dataplugin.foo/build.properties
    com.raytheon.uf.common.dataplugin.foo/component-deploy.xml
    com.raytheon.uf.common.dataplugin.foo/bin
    com.raytheon.uf.common.dataplugin.foo/META-INF/MANIFEST.MF

    com.raytheon.uf.common.dataplugin.foo/src/com/raytheon/uf/common/dataplugin/foo/FooR
ecord.java
    com.raytheon.uf.common.dataplugin.foo/META-INF/services/com.raytheon.uf.common.seria
lization.ISerializableObject
```

These generated project directories can be copied into the Eclipse workspace and imported using the menu selection "File | Import" to display the Import dialog (see **Figure 2-6**).

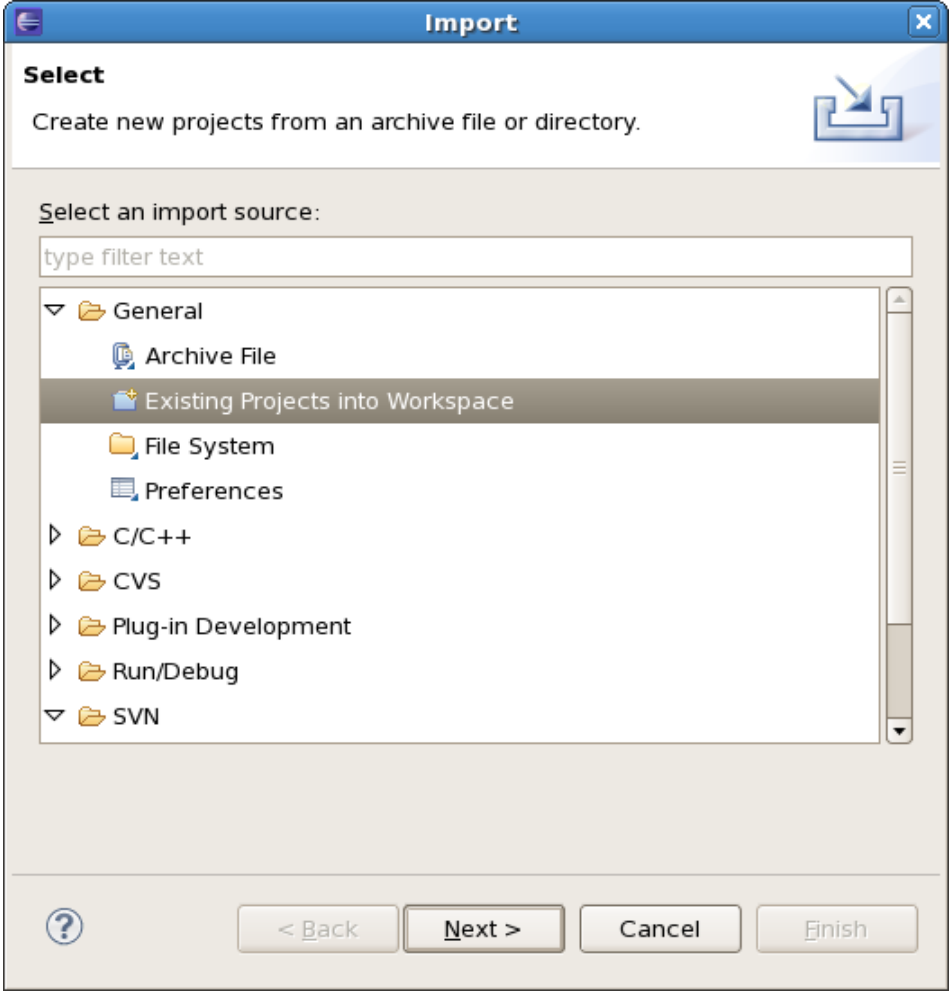

Figure 2-6. Import Dialog: Select

Selecting "Existing Projects into Workspace" will bring up the Import Project dialog, which will require the user to select the Eclipse Workspace to import from. See **Figure 2-7**.

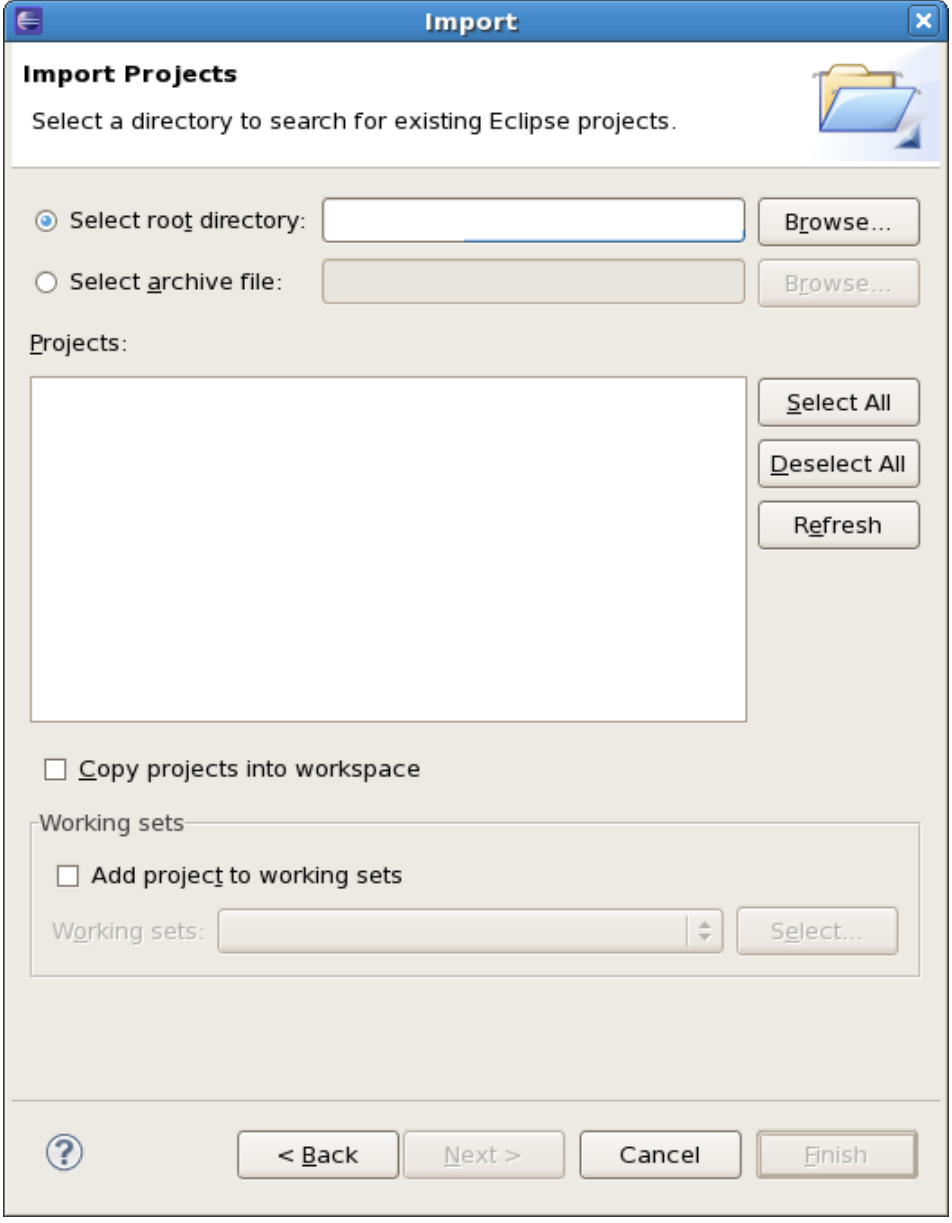

Figure 2-7. Import Dialog: Import Projects

After these projects have been imported into the Eclipse workspace, an entry for each project needs to be entered into the file "feature.xml" located in the "**com.raytheon.edex.feature.uframe**" project. Note that the "...edex.plugin..." project should only need to be mentioned in this feature.xml. The "...common.dataplugin..." project will also need to be added anywhere else it is referenced.

TopoAccess

In EDEX: Import **com.raytheon.uf.edex.topo.TopoQuery**.

In CAVE: import **com.raytheon.uf.viz.core.topo.TopoQuery**.

Get a TopoQuery instance by calling **TopoQuery.getInstance().**

To retrieve topo data for a single point: Call **TopoQuery.getHeight(coordinate)**.

To retrieve topo data for a list of points (or along a path): Call **TopoQuery.getHeight(coordinateList).**

To retrieve topo a grid of topo data: Call **TopoQuery.getHeight(gridGeometry).**

JAXB Serialization

Java Architecture for XML Binding (JAXB) is a set of Java classes in the javax.xml.* packages. They are used to serialize data in xml format, maintain configuration data for GUI displays, and send data to/from CAVE and the EDEX server. The following is an example of how it is used.

The xml file to parse and place in a data class is the localized file **AvnFPS->Configuration->scripts->ClimateDataBase.xml**:

```
 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <!-- comment removed
     -->

<ClimateDataFTPArgs>
    <Site>ftp3.ncdc.noaa.gov</Site>
    <DataDir>pub///data/noaa</DataDir>
    <IshDir>/pub/data/noaa</IshDir>
    <User>anonymous</User>
    <Password>daniel.gilmore@noaa.gov</Password>
</ClimateDataFTPArgs>
```

In the package **com.raytheon.viz.aviation.xml** is the class ClimateDataFTPArgs, which can be used to marshal / unmarshal the xml file as shown in **Figure 2-3**.

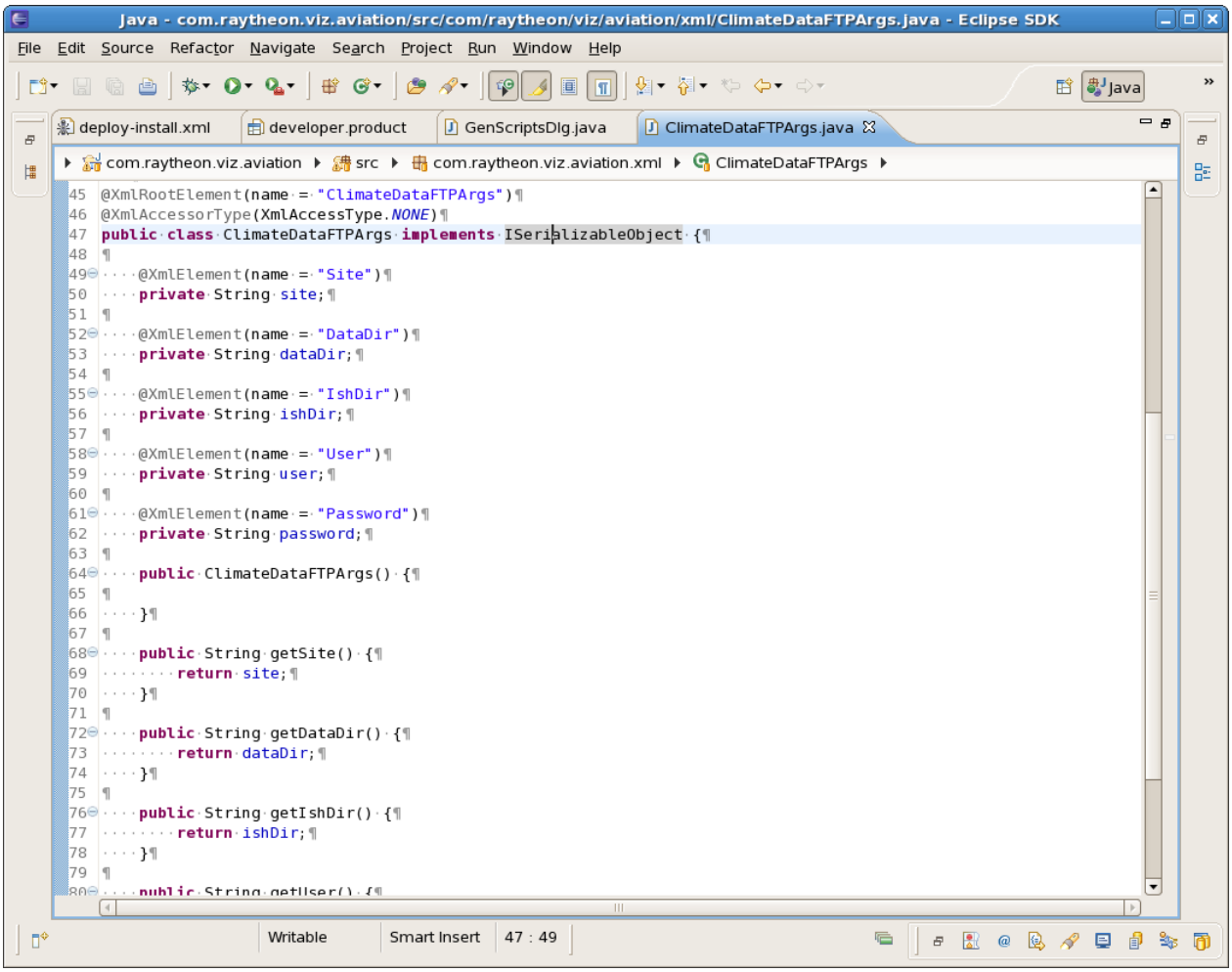

Figure 2-3. ClimateDataFTPArgs.java Tab

The class on line 47 in the figure implements ISeralizableObject. Using this interface indicates the class uses the JAXB annotations and in conjunction with adding the class to the **com.raytheon.uf.common.serialization.ISerializableObject** file in the META-INF/services directory will ensure the localized file is detected at run time.

Line 45 gives the name of the tag for the root element of the xml file. All other tags for this data element need to be embedded in this tag, i.e., <ClimateDataFTPArgs>. Lines 49 and 50 associate the string element **site** with tag <Site>. The other data elements are associated with an element tags in a similar manner. The rest of a data class is normally getter and setter methods.

In the package **com.raytheon.viz.aviation** is the class GenScriptsDlg. Its initFtpArgs method gets the localized xml file and uses JAXB to unmarshal the file into an instance of the ClimateDataFTPArgs class as shown in **Figure 2-4**.

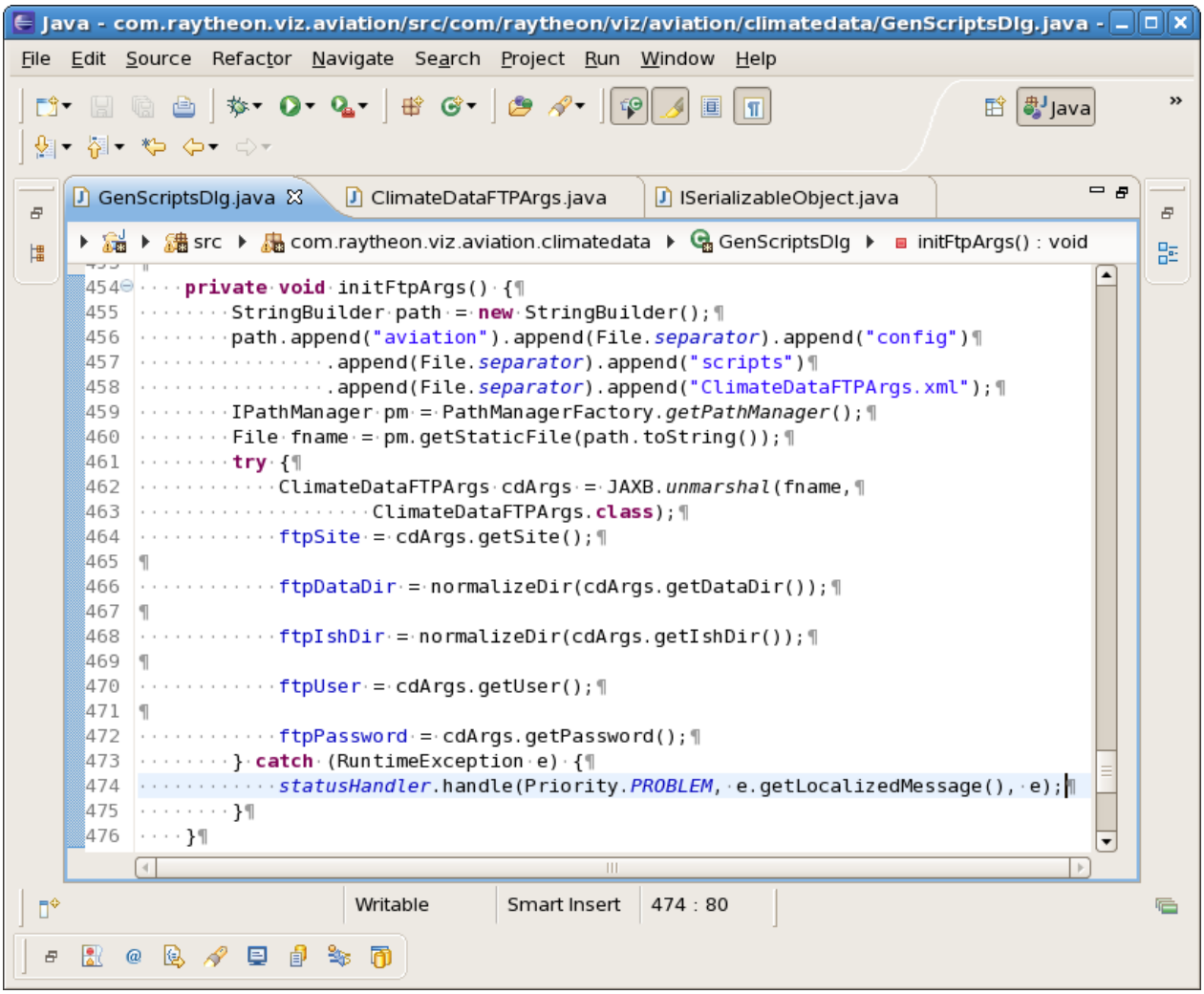

Figure 2-4. GenScriptsDig.java Tab

For a more complex example, see the class **com.raytheon.uf.common.menus.AbstractMenuUtil's toXml** and **fromXml** methods.

Dynamic Serialization

Dynamic Serialize is a software layer built on top of Thrift (see http://wiki.apache.org/thrift/ (http://wiki.apache.org/thrift/) for more information). It is extremely fast, and it is the recommended method of communication between client applications (such as CAVE) and EDEX.

Dynamic Serialize uses getters and setters on a Java class and cglib (see http://cglib.sourceforge.net/ (http://cglib.sourceforge.net/)) to perform serialization and deserialization. To use Dynamic Serialize, follow these steps:

1. Add the @DynamicSerialize annotation to the Java class.
2. Add the @DynamicSerializeElement annotation to the fields on the class you wish to serialize.
3. Ensure each field annotated in the previous step has getters and setters that follow the standard Java naming conventions. Eclipse can generate these for you; right-click in the file, select *Source -> Generate Getters and Setters*.
4. Use SerializationUtil.transformToThrift() to serialize data and SerializationUtil.transformFromThrift() to deserialize data.

Serialization Adapters

To serialize complex or third-party classes, you will need to provide a serialization adapter. Implement the ISerializationTypeAdapter interface (see **com.raytheon.uf.common.serialization.ISerializationTypeAdapter**) and register the adapter with the DynamicSerializationManager (see **com.raytheon.uf.common.serialization.DynamicSerializationManager**).

Implementing a serialization adapter requires implementing a serialize() and deserialize() method. The order in which you serialize data should be the opposite of the order in which you deserialize data. See the BuiltInTypeSupport class (**com.raytheon.uf.common.serialization.BuiltInTypeSupport**) for a number of serialization adapter examples for classes from the Java standard library.

Registration can be done in either of two ways:

1. For in-house developed code: add a @DynamicSerializeTypeAdapter annotation to the class.

```
 // ExampleClass.java
@DynamicSerialize
@DynamicSerializeTypeAdapter(factory = ExampleClassAdapter.class)
public class ExampleClass {
    // ... rest of class code omitted...
}
```

2. For third-party or Java standard library classes, modify the DynamicSerializationManager's static block to directly register the adapter. The following code snippet registers the DateSerializer class to handle serialization and deserialization of the standard Date class with the DynamicSerializationManager:

```
 // DynamicSerializationManager.java
public class DynamicSerializationManager {

    // ...skipping declaration of private fields, internal classes
    static {
        SerializationMetadata md = new SerializationMetadata();
        md.serializationFactory = new DateSerializer();
        md.adapterStructName = Date.class.getName();
        serializedAttributes.put(Date.class.getName(), md);
        // ...skipping registration of other classes ...
    }
}
```

Using Dynamic Serialize with Python

Dynamic Serialize communicates using a Thrift byte stream, which allows cross language communication using the serialized data. In the AWIPS II baseline, we have provided the dynamicserialize Python package to enable Python to communicate with EDEX using Dynamic Serialize. The following snippet of Python code explains how to serialize and deserialize data using the dynamicserialize package:

```
 # DynamicSerializeSample.py

import dynamicserialize
bytes = dynamicserialize.serialize(obj)
# send bytes somewhere, either a service or a file
response = dynamicserialize.deserialize(bytesResponse)
```

For more information on how to use the EDEX Request/Handler API to allow Python code to communicate with EDEX, see the documentation on the Request/Handler API.

Caveats on the Python Interface

Because there are Java language features not supported in Python (and vice versa), there a number of complications and caveats you must be aware of:

1. When serializing Python objects to send to Java, the Python types must match the Java types. For example if a Java field with @DynamicSerializeElement on it is a primitive long, the Python type must be long. Using a Python int in that field will cause an exception in Java. Python has no way of knowing the Java field's type, so it assumes you set the value to the matching type.
2. As Java enums have no Python equivalent, they are serialized as strings. In Python enum fields should be assigned the name of the enum value. Take as an example the GFE class GridParmInfo (see **com.raytheon.uf.common.dataplugin.gfe.db.objects.GridParmInfo**). In Java, the values for the gridType field would be GridType.SCALAR or GridType.NONE. In Python, you would assign these fields "SCALAR" or "NONE."
3. The Python dynamicserialize.dstypes package must have knowledge of the classes you wish to serialize or deserialize. These Python classes are equivalents to the Java classes. Luckily you can auto-generate these Python modules if one does not exist for your class yet. See the next section for more information.
4. If the Java objects are serialized with adapters, you must have equivalent adapters in the dynamicserialize.adapters package.

Converting Java Classes to Python

The PythonFileGenerator is provided as a tool for generating Python modules and classes for use with the Python dynamicserialize package. It is located in the plugin **com.raytheon.uf.common.serialization**. It has a Java main() that analyzes Java @DynamicSerialize classes and produces an equivalent Python module and class. We recommend you run it from Eclipse, and you may need to adjust the classpath on the Run Configuration screen to include the Java plugin project that you wish to serialize.

The tool takes two arguments:

1. '-f filename' where filename is the absolute path to a META-INF/services/com.raytheon.uf.common.serialization.ISerializableObject file or any file that lists the objects you want to be serialized by fully qualified class name, one per line.
2. '-d outputDir' where outputDir is your dynamicserialize.dstypes directory.

The tool will generate a Python class for each Java class it finds in the filename argument, and it will generate the necessary directory structures and __init__.py files for each subpackage so that the classes can be imported using 'from dynamicserialize.dstypes.packageName import *' syntax.

Python Serialization Adapters

If the Java class utilized a serialization adapter, you must also create a serialization adapter for the Python class. To create a serialization adapter in Python you must do the following:

1. Create a Python module within the dynamicserialize.adapters package that implements serialize and deserialize methods and has a global ClassAdapter that calls out which class it handles.
2. Register the serialization adapter by altering the dynamicserialize.adapters package's __init__.py module.

The following is source code for a serialization adapter that handles the Java standard library class Point (see **java.awt.Point and com.raytheon.uf.common.serialization.adapters.PointAdapter** for Java implementations):

```
 # PointAdapter.py:
# Adapter for java.awt.Point
from dynamicserialize.dstypes.java.awt import Point

ClassAdapter = 'java.awt.Point'

def serialize(context, point):
    context.writeI32(point.getX())
    context.writeI32(point.getY())

def deserialize(context):
    x = context.readI32()
    y = context.readI32()
    point = Point()
    point.setX(x)
    point.setY(y)
    return point
```

As you can see, this module implements its serialize and deserialize methods in ways very similar to its Java counterpart. The ClassAdapter global calls out the fully-qualified name of the Java class that is handled by this serialization adapter. If you have multiple classes that, for some reason, could be handled by the same adapter, ClassAdapter could also be a list of class names.

Now to register this adapter so it can be used, we alter the dynamicserialize.adapters package's __init__.py file to add an entry for PointAdapter to the global __all__. See the following:

```
 # __init__.py for Dynamic Serialize adapters.

__all__ = [
        'PointAdapter',
        'StackTraceElementAdapter',
        #  ... rest of list omitted for brevity
        ]

# rest of module's code follows...
```

Localization

Localization serves two primary purposes:

1. To allow site and user-specific customizations of the software.
2. To back up and share these customizations to different workstations.

All changes to localization files are dynamically read in when made via the Localization Perspective in CAVE. This is accomplished via Observer classes in the **com.raytheon.uf.common.localization** package. The Localization Perspective is the preferred method of editing localization files. This lets the AWIPS software manage the files and ensures the files do not get out of sync.

Localization Levels

There are currently five defined localization levels (as defined in **com.raytheon.uf.common.localization.LocalizationContext.LocalizationLevel**):

1. **Base.** Files that should never be changed by users and are applicable to all sites.
2. **Configured.** Files that should never be changed by users but are site-specific files generated from configuration files that are shared by all users configured for that site.
3. **Site.** Files that contain site-specific information and are shared by all users configured for that site.
4. **Workstation.** Files specific to a workstation (based on hostname).
5. **User.** Files specific to the user.

These levels are hierarchical, so there is a set order of precedence that should be used. If a user version of a file exists, it takes precedence over the site, configured and base versions. If a site version of the file exists, it takes precedence over the configured and base versions. If a configured version of the file exists, it takes precedence over the base version. Hence, the order is **user > workstation > site > configured > base**. Typically the base version of the file is either a file that should never be changed or provides defaults if there are no site and user preferences.

Localization Types

Localization Types (defined in **com.raytheon.uf.common.localization.LocalizationContext.LocalizationType)** provide a basic way to categorize localization files based on which AWIPS2 component (the CAVE client or the EDEX server) will be primarily using the file. There are currently four localization types:

1. **CAVE_CONFIG.** config.xml CAVE files tied to preference stores.
2. **CAVE_STATIC.** Files only used by CAVE.
3. **COMMON_STATIC.** Files that both CAVE and EDEX use.
4. **EDEX_STATIC.** Files only used by EDEX.

Localization Context

The localization context is an object that consists of a LocalizationType, LocalizationLevel, and a context name. The LocalizationContext is used in some methods of retrieving localization files.

Localization Code

The most commonly used java code for localization are these files:

- **com.raytheon.uf.common.localization.LocalizationContext.LocalizationLevel**
- **com.raytheon.uf.common.localization.LocalizationContext.LocalizationType**
- **com.raytheon.uf.common.localization.LocalizationContext**
- **com.raytheon.uf.common.localization.PathManagerFactory**

The following are methods of retrieving localization files in all localization contexts.

- List all xml files in a directory.

```
 String[] extensions = new String[] { ".xml" };
String fileNamePath = "/path/to/file/";
LocalizationFile[] locFiles = PathManagerFactory.getPathManager().listStaticFiles(
        fileNamePath, extensions, false, true);
```

- Get a HashMap of all localization versions of a particular file.

```
 IPathManager pm = PathManagerFactory.getPathManager();
Map<LocalizationLevel, LocalizationFile> shefIssueMap = pm.getTieredLocalizationFile(Loc
alizationType.COMMON_STATIC, "/path/to/file");
```

- Retrieve a Site level file by name.

```
 IPathManager pm = PathManagerFactory.getPathManager();
LocalizationContext lc = pm.getContext(LocalizationType.COMMON_STATIC,
        LocalizationLevel.SITE);
LocalizationFile xmlLocalizationFile = pm.getLocalizationFile(lc,
        "/path/to/file");
```

- Read/Write an XML file via Localization.

```
 // Get the file from localization
IPathManager pm = PathManagerFactory.getPathManager();
LocalizationContext lc = pm.getContext(LocalizationType.COMMON_STATIC,
        LocalizationLevel.SITE);
LocalizationFile xmlLocalizationFile = pm.getLocalizationFile(lc,
        "/path/to/file");

// Read the xml contents into the XML data object
XMLObject xml = null;
if (xmlFile != null) {
    xml = (XMLObject) SerializationUtil
            .jaxbUnmarshalFromXmlFile(file.getFile().getAbsolutePath());
} else {
    xml = new XMLObject();
}

// Save the xml object as an xml file
// Localization software takes care of the localization updates
SerializationUtil.jaxbMarshalToXmlFile(xml, xmlLocalizationFile getFile().getAbsolutePat
h());
```

Adding New Directories to the Localization Perspective

For directories and files to show up in the Localization Perspective, an entry must be made in a plugin.xml file. These files are in the viz packages. The steps follow.

1. Open the **plugin.xml** file in Eclipse.
2. Select the Extensions tab.
3. Look in the list for the **com.raytheon.uf.viz.localization.localizationpath**.
4. If it does not exist, click the Add button.
5. In the Extension Point Filter dialog, type "*localization" as shown in **Figure 2-1.**

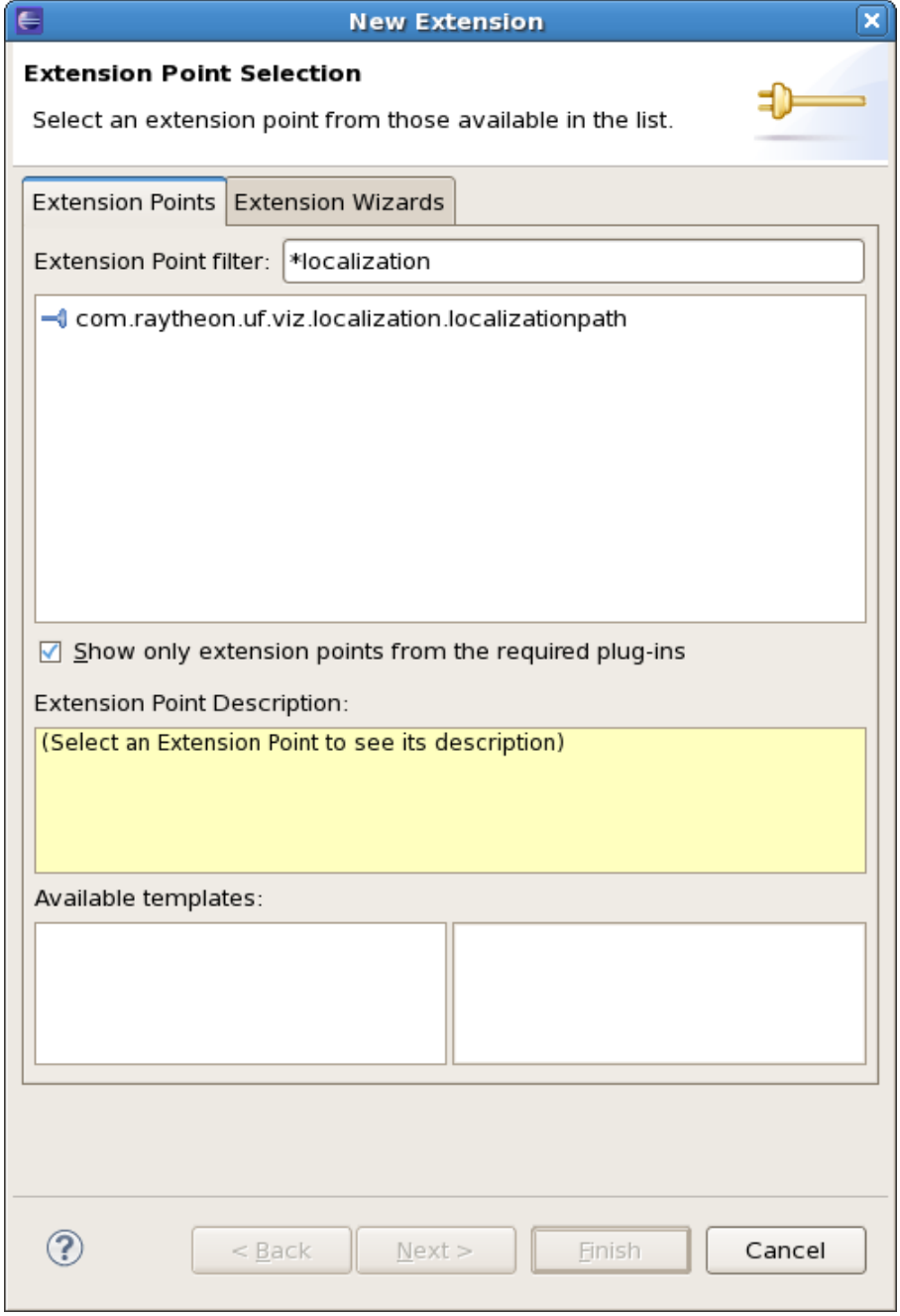

Figure 2-1. Extension Point Selection

6. Select the **com.raytheon.uf.viz.localization.localizationpath** item and click Finish. The item will now be in the list of extensions as shown in **Figure 2-2**.

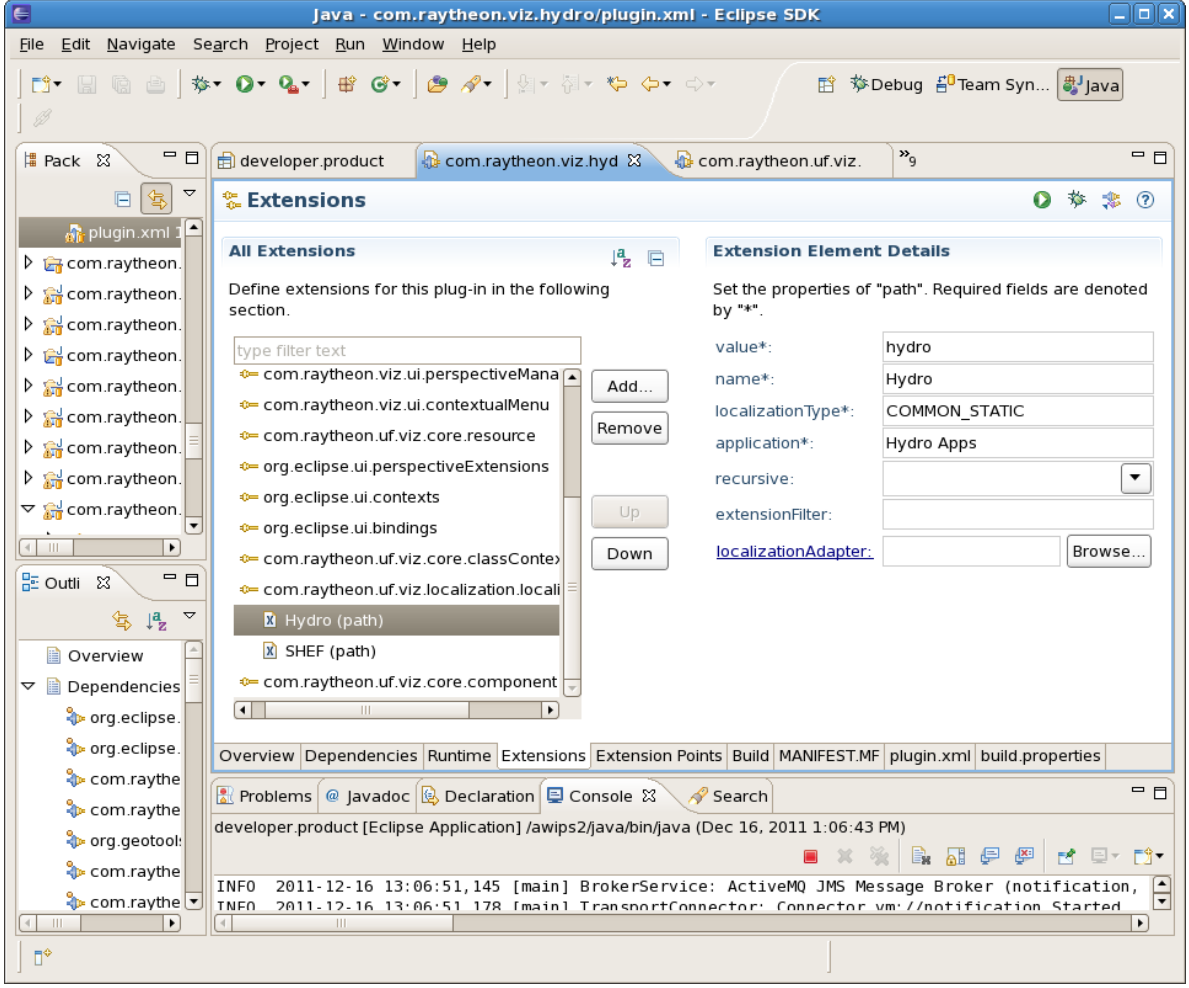

Figure 2-2. List of Extensions

7. Right-click on the localizationpath entry and select New->path menu item.
8. Enter a value: A unique identifier.
9. Enter a name: The name is the first-level subdirectory in the Localization Perspective.
10. Enter the localizationType (CAVE_STATIC, COMMON_STATIC, EDEX_STATIC).
11. Enter the Application where the entry will reside within the Localization Perspective (which main folder). Enter D2D to be in the D2D folder, Hydro Apps to be in the Hydro Apps folder, etc.
12. The recursive entry defaults to false. Set to true to recursively search for files within the localization folder structure.
13. Enter an extension to filter by extension.
14. Enter a localizationAdapter if needed.
15. Save the file and restart CAVE.

The resulting xml entry in the file:

```
<path
   application="Hydro Apps"
   localizationType="COMMON_STATIC"
   name="Hydro"
   value="hydro">
</path>
```

UFStatus

The UFStatus class allows for the logging of a message associating it with an AlertViz source, category, and priority. The following code snippets are from the Java class AvnConfigFileUtil. Perform the following steps in using UFStatus:

1. Import the following:

```
 import com.raytheon.uf.common.status.IUFStatusHandler;

import com.raytheon.uf.common.status.UFStatus;
import com.raytheon.uf.common.status.UFStatus.Priority;
```

   The last import simplifies the handle method's arguments.

2. Get a status handler:

```
 public class AvnConfigFileUtil {

private static final transient IUFStatusHandler statusHandler = UFStatus.getHandler(AvnC
onfigFileUtil.class);
```

   The same handler can be used for all instances of the class. Since there is overhead in getting the handler it is best to limit the calls to getHandler. There are other getHandler method calls that allow you to specify the category and source to use. Normally you will want it to default which is WORKSTATION and CAVE.

3. Example of a message:

```
public static LocalizationFile getStaticLocalizationFile(String configFile) {
   IPathManager pm = PathManagerFactory.getPathManager();
   LocalizationFile lFile = pm.getStaticLocalizationFile(configFile);
   if (lFile == null) {
       String site = LocalizationManager.getInstance().getCurrentSite();
       statusHandler.handle(Priority.CRITICAL, "Unable to find \""
               + configFile + "\" under the directory for site " + site + ".", null);
```

   The first argument is the Priority enum. It has six values that correspond to the AlertViz priorities 0-5: CRITICAL, SIGNIFICANT, PROBLEM, EVENTA, EVENTB, VERBOSE. The second is the message to log message, and the third (null) can be a caught exception that is the reason for the log. When not null it will generate a stacktrace.

The UFStatus can also be used inside Python scripts. This example is from SmartScript.py:

```
def statusBarMsg(self, message, status, category="GFE"):
   from com.raytheon.uf.common.status import UFStatus
   from com.raytheon.uf.common.status import UFStatus_Priority as Priority
   if "A" == status:
       importance = Priority.PROBLEM
   elif "R" == status:
       importance = Priority.EVENTA
   elif "U" == status:
       importance = Priority.CRITICAL
   else:
       importance = Priority.SIGNIFICANT
   if category not in self._handlers:
       self._handlers[category] = UFStatus.getHandler("GFE", category, 'GFE')
       self._handlers[category].handle(importance, message);
```

EDEX, Common, and Viz (Visualization) Plugins

EDEX Plugins

From a developer's standpoint, what you are doing here is creating "Common" objects (data) that will be serialized using thrift and written to HDF5. This is the job of the "EDEX" plugins. This process takes place in EDEX using one of the Camel server instances. This process can be a direct ingest path, monitoring a drop directory, or by using a Uniform Resource Identifier (URI) filter and extending the Composite Product Generator pattern. Any one will suffice. Because EDEX plugins work within the Camel Enterprise Service Bus (ESB), some knowledge of how Camel works is essential. To that end there are two eXtensible Markup Language (XML) files that describe the deployable options and the nature of the "Common" plugin to be produced.

1. A common XML file ~ $pluginname-common.xml
2. An ingest config XML file ~ $pluginname-ingest.xml

Example: We'll call our plugin "example"; this is the "common" file.

```
 <beans
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:amq="http://activemq.apache.org/schema/core"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://activemq.apache.org/schema/core
       http://activemq.apache.org/schema/core/activemq-core.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">
   <bean id="examplePluginName" class="java.lang.String">
       <constructor-arg type="java.lang.String" value="example" />
   </bean>

   <bean id="exampleProperties" class="com.raytheon.uf.common.dataplugin.PluginProperti
es">
       <property name="pluginName" ref="examplePluginName" />
       <property name="pluginFQN" value="com.raytheon.uf.common.dataplugin.example" />
       <property name="dao" value="com.raytheon.uf.common.dataplugin.example.dao.Exampl
eDao" />
       <property name="record" value="com.raytheon.uf.common.dataplugin.example.Example
Record" />
       <property name="dependencyFQNs">
           <list>
               <value>com.raytheon.uf.common.dataplugin.radar</value>
           </list>
       </property>
   </bean>

   <bean factory-bean="pluginRegistry" factory-method="register" depends-on="radarRegis
tered">
       <constructor-arg value="example"/>
       <constructor-arg ref="exampleProperties"/>
   </bean>
</beans>
```

The key things to note from this file example are the definitions for the "Common" plugin that this "EDEX" plugin will be creating. In the <bean> tag, the <property> attributes that describe the "pluginName," "dao," and the "record" classes are key. These are pointers to the JAVA classes that describe the Data Access Object (DAO) and the "Record" class of this plugin.

Example: This is the "ingest" file. This example shows an EDEX plugin implementing the "Composite Product Generator" pattern.

```
 <beans
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:amq="http://activemq.apache.org/schema/core"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://activemq.apache.org/schema/core
       http://activemq.apache.org/schema/core/activemq-core.xsd
       http://camel.apache.org/schema/spring
       http://camel.apache.org/schema/spring/camel-spring.xsd">
   <bean factory-bean="cpgSrvDispatcher" factory-method="register">
       <constructor-arg ref="exampleGenerator"/>
   </bean>

   <camelContext id="example-camel" xmlns="http://camel.apache.org/schema/spring" error
HandlerRef="errorHandler">
       <route id="ExampleGenerate">
           <from uri="jms-generic:queue:exampleGenerate?destinationResolver=#qpidDurabl
eResolver" />
           <doTry>
               <bean ref="serializationUtil" method="transformFromThrift" />
               <bean ref="exampleGenerator" method="generate" />
               <doCatch>
                   <exception>java.lang.Throwable</exception>
                   <to uri="log:ffmp?level=ERROR&amp;showBody=false&amp;showCaughtExcep
tion=true&amp;showStackTrace=true"/>
               </doCatch>
           </doTry>
       </route>
   </camelContext>
</beans>
```

The key things to note from this are: 1) the use of the "SerializationUtil," which deserializes URI messages that are placed on the product generation queue described in the <from> tag; and 2) that the name and method to be used by your generator are described in the <bean> tag named after your "ExampleGenerator" reference. The key thing to understand about the "EDEX" plugins is that they function for the purpose of creating, analyzing, and distributing the data objects (Common plugins) that are needed for display in CAVE by the Visualization (Viz) plugins.

Common Plugins

The "Common" data plugins in AWIPS II are the heart of the AWIPS II system. They are the data transport layer of the triad. They function for one purpose and one purpose alone. To thrift serialize data to HDF5 when created/ingested on EDEX. Then, deserialize and make that data available on the CAVE side. The majority of this work is done for you as a developer simply by extending one class and implementing the interface from another. The PersistablePluginDataObject is the Abstract class you will extend and the IPersistable interface is the one you will implement.

Example "Common" plugin class: ExampleRecord.java

```
 @Entity
@Table(name = "example", uniqueConstraints = { @UniqueConstraint(columnNames = { "dataUR
I" }) })
@XmlRootElement
@XmlAccessorType(XmlAccessType.NONE)
@DynamicSerialize
public class ExampleRecord extends PersistablePluginDataObject implements
        IPersistable {

    private static final long serialVersionUID = 76774564365671L;

    @Column(length = 7)
    @DataURI(position = 1)
    @DynamicSerializeElement
    @XmlElement(nillable = false)
    private String wfo;

    @Column(length = 32)
    @DataURI(position = 2)
    @DynamicSerializeElement
    @XmlElement(nillable = false)
    private String sourceName;

    @Column(length = 32)
    @DataURI(position = 3)
    @DynamicSerializeElement
    @XmlElement(nillable = false)
    private String dataKey;

    @Column(length = 32)
    @DataURI(position = 4)
    @DynamicSerializeElement
    @XmlElement(nillable = false)
    private String siteKey;

    /**
     * Default Constructor
     */
    public ExampleRecord() {
    }

    /**
     * Constructs a record from a dataURI
     *
     * @param uri
     *            The dataURI
     */
    public ExampleRecord(String uri) {
        super(uri);
    }
```

The preceding example shows the basic serialization annotations that are used to identify a particular Plugin Data Object (PDO) and its URI. Notice that they describe the length and the position of each field in the URI. The position is measured starting from position 2 in actuality. So, something described in your class as position 4 is in actuality at position 6 in the real URI. The reason for this is that positions 0 and 1 are designated for the "pluginName" and "dataTime." The manipulation of these is handled by the super class. In our example that would yield this.... Example: /example/11:23:2011 15:23:04:123/wfo/sourceName/dataKey/siteKey

This URI is essential to both writing the record to HDF5 and reading it back out. It is used as a multi-dimensional unique key, uniquely describing an AWIPS II data record.

The writing of the data to HDF5 is handled by another class that is mentioned in the "EDEX" common XML file in the DAO (Data Access Object). In the writing and reading of data in AWIPS II, this is where the rubber meets the road. The DAO is responsible for writing and populating PDO records from HDF5.

Example: common "DAO" class. ExampleDAO.java

```
 public class ExampleDao extends PluginDao {

    public ExampleDao(String pluginName) throws PluginException {
        super(pluginName);
    }

    @Override
    protected IDataStore populateDataStore(IDataStore dataStore, IPersistable obj) throw
s Exception {

        ExampleRecord record = (ExampleRecord) obj;

        // Do something to write the record into your HDF5

        // This method is intended to write the transient object that is used to hold th
e data and convert it into HDF5 Data Records. Note URI is the key

        IDataRecord rec = new FloatDataRecord(huc, record.getDataURI(), dataRec, 1, new
long[] { size });

        dataStore.addDataRecord(rec);

    }

    @Override

    public List<IDataRecord[]> getHDF5Data(List<PluginDataObject> objects, int tileSet)
throws PluginException {
        List<IDataRecord[]> retVal = new ArrayList<IDataRecord[]>();

        for (PluginDataObject obj : objects) {
            IDataRecord[] record = null;

            if (obj instanceof IPersistable) {
                /* connect to the data store and retrieve the data */
                try {
                    record = getDataStore((IPersistable) obj).retrieve(obj.getDataURI
());
                } catch (Exception e) {
                    throw new PluginException("Error retrieving Example HDF5 data", e);
                }
                retVal.add(record);
            }
        }
        return retVal;
    }

}
```

The majority of the work here is again done for you by the super class you are extending. In this case the PluginDao class. Since the "Common" plugins are the only ones that are accessed from both sides of the dependency triad. They are by far the most important link in the chain. Having a well written and swiftly executing "Common" plugins will aid in both construction and display of your data on the EDEX and CAVE (Viz) sides.

One last piece of instruction on the "Common" plugins regards the fact that they are frequently serialized. In AWIPS II most objects are serialized using Facebook's Thrift serialization. In the "Common" plugins META-INF/services directory, you must add a file called "**com.raytheon.uf.common.serialization.ISerializableObject**". The reason for this is that when Thrift seeks to serialize and de-serialize a class. It uses this entry in the META-INF as a lookup. So, any classes you wish to serialize must be listed using Fully Qualified Domain Name (FQDN) in the ISerializableObject file.

In our example here using ExampleRecord, we would have:
com.raytheon.uf.common.dataplugin.example.ExampleRecord

This class would have to be listed at a minimum in order for the Record to be recognized. If you have sub-objects that your record contains, they must also be listed in the file.

Viz Plugins

The last and most visible category of AWIPS II plugins is the "Viz" or CAVE plugin. CAVE plugins are generally used in the display of data created by the "EDEX" plugins and serialized and transported by the "Common" plugins. In General, they follow a pattern by which they have what is known as a "Resource" class and a "ResourceData" class. The purpose of the Resource class is to interact directly with the Graphical User Interface (GUI) changing the display parameters of the data. Here is an example Resource class.

Example: ExampleResource.java

```
 public class ExampleResource extends AbstractVizResource<ExampleResourceData, MapDescri
ptor> implements IResourceDataChanged {

    public String icao;

    public String fieldName;

    public String fieldUnitString;

    public ExampleRecord record;

    private HashMap<DataTime, GriddedImageDisplay2> griddedDisplayMap;

    public DataTime displayedDataTime;

    public DataTime previousDataTime;

    private String colormapfile = null;

    /* The font used */
    public IFont font = null;

    public ExampleResource(ExampleResourceData data, LoadProperties props) {
        super(data, props);

        data.addChangeListener(this);
        this.dataTimes = new ArrayList<DataTime>();
        griddedDisplayMap = new HashMap<DataTime, GriddedImageDisplay2>();
    }

    /*
     * (non-Javadoc)
     *
     * @see com.raytheon.viz.core.rsc.IVizResource#getName()
     */
    @Override
    public String getName() {
        ExampleRecord record = null;
        for (ExampleRecord rec : resourceData.dataObjectMap.values()) {
            record = rec;
            break;
        }

        if (record == null) {
            return "";
        }

        StringBuilder prefix = new StringBuilder();
        prefix.append(record.getIcao());
        prefix.append(" ");
        prefix.append(record.getParameterName());

        return prefix.toString();
    }

    @Override
    public void resourceChanged(ChangeType type, Object object) {
        if (type.equals(ChangeType.DATA_UPDATE)) {
            PluginDataObject[] pdos = (PluginDataObject[]) object;
            for (PluginDataObject pdo : pdos) {
                try {
                    ExampleRecord example = (ExampleRecord) pdo;
```

```
                resourceData.dataObjectMap.put(example.getDataTime(), example);
                record = example;
            } catch (Exception e) {
                statusHandler.handle(Priority.PROBLEM,
                        "Error updating Example resource", e);
            }
        }

        issueRefresh();
    }
}

@Override
protected void disposeInternal() {

    for (DataTime key : griddedDisplayMap.keySet()) {
        GriddedImageDisplay2 gDisplay = griddedDisplayMap.get(key);
        if (gDisplay != null) {
            gDisplay.dispose();
        }
    }

    griddedDisplayMap.clear();

    if (font != null) {
        font.dispose();
    }
}

@Override
protected void initInternal(IGraphicsTarget target) throws VizException {
    if (this.font == null) {
        this.font = target.initializeFont("Dialog", 11, null);
    }
    init = true;
}

@Override
protected void paintInternal(IGraphicsTarget target,
        PaintProperties paintProps) throws VizException {

    this.displayedDataTime = paintProps.getDataTime();

    // Pull the record out
    this.record = resourceData.dataObjectMap.get(this.displayedDataTime);

    if (record == null) {
        // Don't have data for this frame
        return;
    }

    GriddedImageDisplay2 gridDisplay = griddedDisplayMap.get(displayedDataTime);

    if (record.getDataArray() == null) {
        record = resourceData.populateRecord(record);
    }

    if (gridDisplay == null) {
        gridDisplay = new GriddedImageDisplay2(ShortBuffer.wrap(record
                .getDataArray()), record.getGridGeometry(), this,
                target.getViewType());
        gridDisplay.init(target);
        this.previousDataTime = displayedDataTime;
```

```
            griddedDisplayMap.put(displayedDataTime, gridDisplay);
        }

        ColorMapParameters colorMapParameters = getCapability(
                ColorMapCapability.class).getColorMapParameters();

        if (record != null && init) {
            StyleRule sr = StyleManager.getInstance().getStyleRule(
                    StyleManager.StyleType.IMAGERY, getMatchCriteria());
            this.colormapfile = ((ImagePreferences) sr.getPreferences())
                    .getDefaultColormap();

            IColorMap cxml = ColorMapLoader.loadColorMap(colormapfile);
            ColorMap colorMap = new ColorMap(colormapfile, (ColorMap) cxml);
            colorMapParameters.setColorMap(colorMap);

            colorMapParameters.setDataMapping(((ImagePreferences) sr
                    .getPreferences()).getDataMapping());

            cwatmax = colorMapParameters
                    .getDataMapping()
                    .getEntries()
                    .get(colorMapParameters.getDataMapping().getEntries()
                            .size() - 1).getDisplayValue().floatValue();
            cwatmin = colorMapParameters.getDataMapping().getEntries().get(0)
                    .getDisplayValue().floatValue();
            colorMapParameters.setDataMax(Short.MAX_VALUE);
            colorMapParameters.setDataMin(Short.MIN_VALUE);
            colorMapParameters.setColorMapMax(cwatmax);
            colorMapParameters.setColorMapMin(cwatmin);

            init = false;
        }

        gridDisplay.paint(target, paintProps);

    }

    /*
     * (non-Javadoc)
     *
     * @see
     * com.raytheon.viz.core.rsc.capabilities.IInspectableResource#inspect(com
     * .vividsolutions.jts.geom.Coordinate)
     */
    @Override
    public String inspect(ReferencedCoordinate latLon) throws VizException {
        String inspect = "NO DATA";
        if (record != null) {

            if (record.getDataArray() == null) {
                record = resourceData.populateRecord(record);
            }

            Coordinate coor = null;
            try {
                if (record.getDataArray() != null) {
                    coor = latLon.asGridCell(record.getGridGeometry(),
                            PixelInCell.CELL_CENTER);
                    int index = (int) ((record.getNx() * Math.round(coor.y)) + Math.roun
d(coor.x));
                    int value = 0;
                    if (index < record.getDataArray().length && index > -1) {
```

```
                        value = record.getDataArray()[index];

                        if (value >= 10) {
                            inspect = value + ":  "
                                    + SCTI.getSCTImessage(value);
                        }
                    }
                }
            } catch (TransformException e) {
                e.printStackTrace();
            } catch (FactoryException e) {
                e.printStackTrace();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        return inspect;
    }

    @Override
    public void project(CoordinateReferenceSystem crs) throws VizException {
        for (DataTime dTime : griddedDisplayMap.keySet()) {
            GriddedImageDisplay2 gDisplay = griddedDisplayMap.get(dTime);
            if (gDisplay != null) {
                gDisplay.reproject();
            }
        }
    }

    @Override
    public void remove(DataTime dataTime) {
        this.dataTimes.remove(dataTime);
        GriddedImageDisplay2 display = this.griddedDisplayMap.remove(dataTime);
        if (display != null) {
            display.dispose();
        }
    }
}
```

Note that the Resource class implements and extends specific classes. Resources will in most cases extend the AbstractVizResource class. Note also that the Resource's ResourceData class is a parameter passed to the public constructor for the Resource class. This is not by accident. It is the ResourceData that gives the Resource class the PDOs ("Common") plugins it will display. All of the methods listed in this ExampleResource are methods required to implement or otherwise override. The IResourceDataChanged interface allows the Resource to receive updates of new PDOs when new ones are created by EDEX. There are many other interfaces you can implement for specific needs if your Resource requires them.

The other class that most CAVE ("Viz") plugins have is the ResourceData class. The ResourceData class serves to gather and store "Common" data plugins that the Resource will display.

```
 @XmlAccessorType(XmlAccessType.NONE)
@XmlType(name = "exampleResourceData")
public class ExampleResourceData extends AbstractRequestableResourceData {

    @XmlAttribute
    public String sourceName;

    @XmlAttribute
    public String huc;

    @XmlAttribute
    public String dataKey;

    @XmlAttribute
    public String siteKey;

    public String wfo;

    public ExampleRecord[] records;

    public Map<DataTime, ExampleRecord> dataObjectMap;


    public ExampleResourceData() {

    super();
    this.nameGenerator = new AbstractNameGenerator() {

        @Override
        public String getName(AbstractVizResource<?, ?> resource) {
        return mapName;
        }

    };

}

    @Override
    protected AbstractVizResource<?, ?> constructResource(
            LoadProperties loadProperties, PluginDataObject[] objects) {
        records = new CWATRecord[objects.length];
        dataObjectMap = new HashMap<DataTime, ExampleRecord>();

        for (int i = 0; i < objects.length; i++) {
            records[i] = (ExampleRecord) objects[i];
            dataObjectMap.put(records[i].getDataTime(), records[i]);
        }

        return new Exampleesource(this, loadProperties);
    }

    /**
     * @return the records
     */
    public ExampleRecord[] getRecords() {
        return records;
    }
}
```

There are a couple of things to note in this example of ResourceData class. Notice the @XmlAttribute annotations that are placed on some of the public variables. The reason for this is that these are used to filter which URI's of a plugin type you will see in your Resource. This is the

reason that the ResourceData class itself is annotated for serialization. This magic is all done in the Bundle/Procedure XML files that reference the ResourceData objects. That discussion however is outside the scope of this one. The basic reason for the ResourceData class is to package and deliver the PDO's ("Common") plugins that the Resource will then display. Again like other examples, most of the work here is actually done in the super class or through implemented interfaces. In this and most cases, our ExampleResourceData class extends the *AbstractRequestableResourceData super class*.

Plugins

AWIPS II is designed as a plugin-based architecture modeled after the Eclipse RCP project. The basic concept of this architecture is that a core set of plugins provides useful APIs and then more plugins are added to provide additional functionality.

Pluginization

Pluginization is the act of making a component pluggable. Pluginization can also be understood as making components modular, not dependent on unrelated plugins, and able to adapt to future requirements. To understand pluginization, it is helpful to consider some examples:

- Within Eclipse:
    - If you wish to develop Java code, you do not need the C++, Ruby, Python, etc plugins installed.
    - If you wish to use Git for version control, you do not need the Subversion plugins installed, and vice versa.
- Within AWIPS II:
    - If you wish to display satellite data, you should not need the radar plugin installed, and vice versa.
    - If you wish to use the D2D perspective, you should not need the GFE perspective installed.

Some guidelines for achieving good pluginization:

- Limit/reduce the number of dependencies a plugin needs to work
- Modularize plugins, i.e. don't have a huge plugin that does a variety of things, instead separate and target plugins for specific functionality
- Contemplate if the plugin and its code is likely to be reused and built upon in the future
    - If so, strive for plugin agnostic design

Plugin agnostic components are designed in such a way that functionality can be injected, extended, or contributed to by other plugins. With plugin agnostic design, the code of the component does not know what plugins will be installed and contributing functionality, instead it provides hooks for other plugins to identify themselves and their contributions. Injection through the Spring Framework as well as the Eclipse RCP Framework help achieve this design. Plugin writers should be aware of this type of design and strive to mimic it for their own applications to allow for maximum extendability.

Unfortunately, good pluginization is easier to spot in hindsight, when you want to hook in a new capability but find you cannot. This is why the plugin layout evolves sometimes, such as when a plugin splits into multiple plugins or classes are moved from one plugin to another.

AWIPS II Plugins

Common

The plugins that contain code that is common to both CAVE and EDEX are considered to be the common. These plugins generally contain the name 'common' in their fully-qualified name. These plugins provide various utilities and core capabilities for both applications including serialization, file access, geospatial operations, metadata/data access, and http services to name a few. Common plugins should strive to limit their dependencies and should only depend on other common plugins and FOSS plugins.

EDEX

EDEX plugins are those that provide the functionality of the EDEX server application. These plugins generally contain the name 'edex' in their fully-qualified name.

These plugins include Apache Camel configuration, sbn/file endpoint data distribution, data decoding/storage, data type registration, and many others. EDEX plugins should only depend on common plugins and other EDEX plugins.

CAVE

CAVE plugins are those that provide the basic functionality of a CAVE/Visualization-based client application. These plugins generally contain the name 'viz' in their fully-qualified name. The CAVE plugins include the Eclipse RCP Framework, graphics rendering APIs, request service APIs for data access, dialogs for user interaction, and many user interface utilities. CAVE plugins should only depend on common plugins and other viz plugins.

Note: Because CAVE is built from the Eclipse RCP Framework, many tutorials on Eclipse RCP can be applied within CAVE.

Plugin Naming

Plugin naming conventions generally follow Java package naming conventions (https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html). Plugin names are always in lowercase and are based on three parts:

1. The organization that developed the plugin.
    - **com.raytheon, gov.noaa.nws.ncep, gov.nasa, org.apache, edu.wisc.** The organization name is generally a reverse of a domain name.
2. The part of the system the plugin is developed for.
    - **(common|edex|viz)**. The organization name is followed by the part of the system the plugin was developed for: common, edex, or viz (CAVE). Raytheon plugins usually have a 'uf' that stands for uFrame and applies to any code developed for AWIPS II. Raytheon plugins that were originally created before 2009 may not have the 'uf' in the plugin name.
3. The functionality the plugin provides.
    - This is up to the developer to be descriptive. Examples include **d2d.ui, gfe, radar, satellite,** and **geospatial**. For more modular plugin separation, extra descriptors can be added just like with package naming, such as **d2d.core**, **d2d.ui**, **d2d.ui.obs**, **d2d.ui.popupskewt**.

Raytheon examples:

- **com.raytheon.uf.common.dataplugin.radar.** A plugin that defines a metadata object structure for radar data and common utilities for radar data.
- **com.raytheon.uf.edex.plugin.radar.** A plugin that is used for decoding radar data into the metadata structure defined in the common dataplugin.
- **com.raytheon.uf.viz.radar.** A plugin that contains code for displaying and interacting with radar data in CAVE.
- **com.raytheon.rcm.server.** A plugin that contains code for use in the radar server.

NWS examples:

- **gov.noaa.nws.ncep.common.dataplugin**
- **gov.noaa.nws.ncep.common.log**
- **gov.noaa.nws.ncep.edex.plugin**
- **gov.noaa.nws.ncep.viz**

Other examples:

- **org.postgres**
- **javax.media.opengl**

AWIPS II Environment, Development Approach, Driving Requirements

System Operational Environment

- Large, dispersed organization, Conterminous/Contiguous/Commercial United States (CONUS) and Outside Conterminous/Contiguous/Commercial United States (OCONUS).
- 24/7 operations.
- Continuously changing.
- Meteorologist and Hydrologist users.
- Common and varied functions across sites.
- Uniform products with unique details.
- Unique products at some sites.
- Limited resources.

Approach to Development

- The AWIPS II architecture was a clean-sheet design.
- Development was done off-line while Legacy AWIPS continued to support the mission.
- Legacy AWIPS applications were reengineered for the new environment while maintaining the existing user interface (e.g., black box).

Driving Requirements of AWIPS II

- Maximize adaptability.
- Maximize affordability (e.g., provide best value).
- While meeting "ility" requirements.

Affordability is defined herein by the organization's multi-year budget forecast that is available for the system's development, maintenance, and support costs (e.g., Total Cost of Ownership). Although the forecast will vary from year to year and *might* increase, or decrease, it must be - and was considered as - an immutable system design constraint to meet or beat, management reserve notwithstanding. Of course, the system must still meet all functional and operational requirements, or otherwise design in the facility to be tailored to the local environment.

Adaptability is the ability to change to fit current circumstances. The *degree* of adaptability is the ease with which this is done. The design goal is to achieve the highest degree of adaptability possible within the affordability constraint. Affordability and adaptability are affected by decisions made in the design and maintenance of the system.

SSDD Motivation, Assumptions, Contents

Motivation

- Maintain the integrity of the AWIPS II Architecture.
- Leverage inherent capability of AWIPS II Architecture.
- Minimize cost while meeting need.

Assumptions

- Software Developer is the intended reader.
- Software Developer knows how to program using Java, understands Object Oriented concepts, and has at least an intermediate understanding of Eclipse IDE/Plug-in Framework, but may be new to AWIPS II.
- Knowledge in the following areas are helpful:
  SWT, Python, ESB concepts, JMS, XML, SQL, Hibernate, and Spring.

SSDD Contents

- References are used in lieu of replicating information that exists elsewhere to minimize maintenance. Information on AWIPS hardware design, communications networks, configuration management, etc., is readily available elsewhere (i.e., *System Manager's Manual*).
- Many examples provided throughout the document are readily available in the Eclipse/AWIPS Development Environment (ADE).

DataAccessLayer Class

The DataAccessLayer class is part of the Data Access Framework (DAF). It contains only static methods which return other Python objects to be used as part of a data request.

Methods

getAvailableTimes

Get the times of available data to request

```
getAvailableTimes(request, refTimeOnly=False)
```

- Arguments:
  - request: an IDataRequest defining the data to request
  - refTimeOnly: Boolean - If True only unique refTimes (i.e. cycle times) are returned
- Returns: a list of DataTime objects

getGridData

Gets the grid data that matches the request at the specified times. Each combination of parameter, level, and data time will be returned as a separate PyGridData.

```
getGridData(request, times=[])
```

- Arguments:
  - request: an IDataRequest defining the data to request
  - times: a list of DataTime objects, a TimeRange object, or None if the data is time agnostic
- Returns: a list of PyGridData objects

getGeometryData

Gets the geometry data that matches the request at the specified times. Each combination of geometry, level, and data time will be returned as a separate PyGeometryData.

```
getGeometryData(request, times=[])
```

- Arguments:
  - request: an IDataRequest defining the data to request
  - times: a list of DataTime objects, a TimeRange object, or None if the data is time agnostic
- Returns: a list of PyGeometryData objects

getAvailableLocationNames

Gets the available location names that match the request without actually requesting the data.

```
getAvailableLocationNames(request)
```

- Arguments:
  - request: an IDataRequest defining the data to request
- Returns: a list of strings of available location names

getAvailableParameters

Gets the available parameters names that match the request without actually requesting the data.

```
getAvailableParameters(request)
```

- Argument:
  - request: the request to find matching parameter names for
- Returns: a list of strings of available parameter names.

getAvailableLevels

Gets the available levels that match the request without actually requesting the data.

```
getAvailableLevels(request)
```

- Arguments:
  - request: the request to find matching levels for
- Returns: a list of strings of available levels.

getRequiredIdentifiers

Gets the required identifiers for this datatype. These identifiers must be set on a request for the request of this datatype to succeed.

- Arguments:
  - datatype: the datatype to find required identifiers for
- Returns: a list of strings of required identifiers

getOptionalIdentifiers

Gets the optional identifiers for this datatype.

- Arguments:
  - datatype: the datatype to find optional identifiers for
- Returns: a list of strings of optional identifiers

newDataRequest

Creates a new instance of IDataRequest suitable for the runtime environment

```
newDataRequest()
```

- Returns: a new IDataRequest (e.g. a DefaultDataRequest object)

getSupportedDatatypes

Gets the datatypes that are supported by the framework

- Returns: a list of strings of supported datatypes

changeEDEXHost

Changes the EDEX host that the DAF is communicating with.

```
changeEDEXHost(newhost)
```

- Arguments:
  - newhost: a string for the new hostname to connect to (e.g. ec, localhost)

Data Available via the DAF

The various data plugins in EDEX get "plugged in" to the DAF (typically) by the contractor's developers in the baseline Java code. As of OB 16.1.1, the following plugins are enabled in the DAF:

- acars
- airep
- binlightning
- bufrmosavn
- bufrmoseta
- bufrmosgfs
- bufrmoshpc
- bufrmoslamp
- bufrmosmrf
- bufrmosngm
- bufrua
- climate
- common_obs_spatial
- ffmp
- gfe
- grid
- hazards
- hydro
- ldadmesonet
- maps
- modelsounding
- obs
- pirep
- practicewarning
- profiler
- radar
- radar_spatial
- satellite
- sfcobs
- warning