The OpenMap Developer's Guide

1 The OpenMap Developer's Guide 4
2 What is OpenMap, and what can you do with it? 4
  2.1 Configurations - Application, Applet, Image Server, Components 4
  2.2 The OpenMap Application 4
  2.3 The OpenMap Applet 5
  2.4 Using OpenMap as an Image Server. 6
  2.5 Using OpenMap Components in Other Applications 6
3 The General Use of Projections 7
4 Event Handling 8
5 Architecture Components 9
  5.1 The MapBean, MapHandler (BeanContext) and MapPanel 9
  5.2 The LayerHandler 12
  5.3 Using MouseEvents with the MouseDelegator and MapMouseModes 12
  5.4 Using Properties and the PropertyHandler 13
  5.5 Various GUI Controls, the ToolPanel and Menus 15
6 Displaying Data with OMGraphics 17
  6.1 OMGraphic Classes, Render types, Line types 17
  6.2 Basic OMGraphic Types 18
  6.3 OMArc and OMCircle 18
  6.4 OMGrid 18
  6.5 OMLine and OMArrowHead 18
  6.6 OMPoly, OMDistance, OMSpline and OMDecoratedSpline 19
  6.7 OMRect 19
  6.8 OMRasterObject family, including OMBitmap, OMRaster, OMScalingRaster, and OMScalingIcon 19
  6.9 OMText 20
  6.10 OMGeometryList and OMAreaList 20
  6.11 OMGraphics and Rendering Attributes 20
  6.12 OMGraphic Management using the OMGraphicList 21
  6.13 The Generate and Render Paradigm 22
  6.14 Using MouseEvents with OMGraphics 22
  6.15 The OMGraphicHandler Interface and FilterSupport 24
  6.16 Compound OMGraphics, Extensions and Customization 24
7 Displaying Data on the Map 25
  7.1 Layer Basics 26
  7.2 Projection Changes and Painting 27
  7.3 Layer User Interfaces (Palettes) 27
  7.4 The OMGraphicHandlerLayer 28
    7.4.1 Behavior Policies in the OMGraphicHandlerLayer 28
    7.4.2 The OMGraphicHandlerLayer and MouseEvents 29
    7.4.3 MapMouseInterpreters 29
    7.4.4 GestureResponsePolicies 30
    7.4.5 The OMGraphicHandlerLayer as a GestureResponsePolicy 32
  7.5 PlugIns 32
  7.6 GraphicLoaders 33

7.7 Modifying OMGraphics with the Drawing Tool 33
7.8 Animation Techniques 35
8 Displaying Supported Data Formats 35
8.1 Location Data via CSV files or a Database 35
8.2 ESRI Shape Data 36
8.3 National Geospatial-Intelligence Agency (NGA) Databases 38
8.4 Raster Product Format (RPF), including CADRG and CIB 38
8.5 Vector Product Format (VPF), including VMAP and DCW 39
8.6 Digital Terrain Elevation Data (DTED) 39
8.7 ETOPO 40
8.8 ArcInfo Export Files (e00) and MapInfo files (mif) 40
9 Conclusion 40
10 Appendix A: Configuring the Applet 41
11 Appendix B: Examples of using OpenMap Image Creation Applications 42
12 Appendix C: Configuring the com.bbn.openmap.image.ImageServer 42
13 Appendix D: An `openmap.properties` File 44

1 The OpenMap Developer's Guide

This developer's guide is intended to assist developers with configuring and developing OpenMap software. It provides an overview of the architecture and addresses concepts that are important for using the package, and also inspects the contents of the OpenMap toolkit and describes how different components can be used and modified. It also assumes that the reader is somewhat familiar with the Java programming language and the component naming conventions used in Java. More information about Java, including tutorials, can be found at Sun's Java website at http://java.sun.com.

2 What is OpenMap, and what can you do with it?

OpenMap is a geospatial visualization toolkit written in Java, provided as open source software from BBN Technologies. It can provide a way to visualize and understand your data by displaying new relationships and trends that may be otherwise hidden by sheer volume of information, or by creating new information by combining your data with other data sources and showing those relationships.

The package includes an application framework and software components that display information gathered from different map data sources. OpenMap components can display map data from local files and remote servers, in a variety of formats.

The application framework provides several ways for the map data to be queried and changed, and for all the components in the application to communicate and work together. The framework is based on Sun's Java Beans technology and can be configured with text property files so new components can be added and removed from the application without recompiling application code.

OpenMap can be run as an application, an applet, as components within another application or within a servlet environment to provide map images for a web server.

2.1 Configurations - Application, Applet, Image Server, Components

The OpenMap components can be used in a variety of ways, in a number of different architectures.

2.2 The OpenMap Application

OpenMap comes with an application component (`com.bbn.openmap.app.OpenMap`) that creates the basic framework for configuring and running a mapping application. The application framework is a shell, a container for components added to an application.

This shell provides the mechanism for these components to find and connect to other components they need, and can easily form the backbone of applications with differing interfaces and functionalities.

The OpenMap application is configured with an openmap.properties file. The contents of this file specify which components are created and added to the application framework, including the layers. New applications can be configured without recompilation, simply by modifying the openmap.properties file with a text editor. Components that have be written with an understanding of the framework can be added to the application simply by making additions to the properties file. Also, components written to use properties will be given their settings in order to initialize themselves properly. Layers that rely on the location of data files or servers, for instance, usually have properties that let those locations be set at runtime. Configuring the openmap.properties file will be discussed in more detail later.

Layers in an OpenMap application can use data in many different ways when running in an application environment. They can create map features:

- By computation.
- From reading data files directly off a local hard drive.
- From reading data files from a URL.
- From reading data files contained in a jar file.
- Using information retrieved from a database (JDBC).
- Using information received from a server (images or map objects).

2.3 The OpenMap Applet

OpenMap also comes with an applet component that creates the application framework within an applet framework (`com.bbn.openmap.app.OpenMapApplet`). The important distinction to make for OpenMap running as an applet is that the components are restricted by the Java environment (sandbox) from communicating with any other computer other than the one that the applet was served from. This may affect how layers access their data sources, for instance. Layers can't access data on the local hard drives of the host computer. They can create map features:

- By computation.
- From reading data files from a URL, as long as the URL references the server from where the applet was downloaded.
- From reading data files contained in a jar file.
- Using information retrieved from a database (JDBC), if the database server and the applet server are the same machine.

- From reading data files contained in a jar file that is made available to the applet.
- Using information received from a server (images or map objects. The server must be running on the applet server, or have a redirecting proxy server running on the applet server that acts as a go-between for the server and applet.

There are other restrictions with regard to certain types of application windows that are allowed, and how much access the applet has to the host computer system information.

See Appendix A for an example html page that will launch the OpenMap applet.

2.4 Using OpenMap as an Image Server.

OpenMap comes with components that can be used in a servlet environment to create map images for a website.

The `com.bbn.openmap.image.ImageServer` is a class that can use an openmap.properties file to configure layers to use for an image, and then create an image from those layers. This class can be run as a stand-alone application to create images.

The `com.bbn.openmap.image.MapRequestHandler` is an extension of the ImageServer, and while this class also parses an openmap.properties file to configure its layers, it is also capable of parsing request strings to create images dynamically, adjusting the projection and layer content of the images as requested.

The `com.bbn.openmap.image.SimpleHttpImageServer` is a example of using the MapRequestHandler. It is a lightweight (simple!) http server that directs map image requests to the MapRequestHandler, returning an image based on the parameters of the request. For a robust version of an image server, the MapRequestHandler can be coupled with an established servlet environment like Apache Tomcat.

The OpenMap toolkit also includes a set of .NET pages that use the MapRequestHandler to create a Map service.

See Appendix B for examples of how to modify an openmap.properties file to run an ImageServer application.

2.5 Using OpenMap Components in Other Applications

OpenMap user interface components and Layers are based on Java Swing components, and can be used in other applications as regular swing components outside of the OpenMap application framework. Other components, used for cache mechanisms, projection transformations and data reading and writing can also be used in other applications.

3 The General Use of Projections

One of the main ideas behind OpenMap is that locating your components on the map shouldn't be something you worry about - that's something OpenMap takes care of.

In geospatial terms, a projection is the translation between a spherical model of the earth to a flat, two-dimensional surface. There are different advantages to different projections, depending on the task at hand. In OpenMap, projections are components that handle the translations between latitude and longitude coordinates and screen locations.

Cartesian coordinate systems are not easily handled in OpenMap. OpenMap projections only handle translation from decimal degree latitude/longitude data to map view x/y coordinates.

A datum is a definition of how coordinates on the sphere are defined. OpenMap projections assume that latitude and longitude coordinates are defined in terms of the WGS 84 datum. The API will be modified at some point in the future to allow other datum to be specified for coordinates. If your data coordinates are defined in a different datum, there will be some error introduced into the map when different data sets are matched.

It's important to make sure your data is compatible with other data being displayed over the same map. OpenMap currently does not have a mechanism for translating coordinates from other datum into WGS 84, but there are packages available on the internet for performing this function.

NOTE: OpenMap projections are based on a spherical model of the Earth. Ellipsoidal flattening (eccentricity) is not taken into account.

OpenMap projections do more than providing forward and inverse translation operations on coordinates. OpenMap components let you define map shapes that should be drawn on the map. Map shapes defined as pairs of coordinate points are known as vector features.

The OpenMap projections are capable of determining how vector features, defined in terms of latitude and longitude coordinates, are rendered on themselves. The geometry of these shapes depend on the line type specified for a shape, and OpenMap projections handle three different line types:

- Great Circle lines. The line between two coordinates is the shortest distance between these points on the surface of the Earth.

- Rhumb Lines. The line between two coordinates has constant bearing. As you move along a line from one point to another, you are always moving in the same direction.
- Straight. The lines between coordinates drawn in pixel space.

Shapes created from vector feature geometries created from different data sources will line up with each other if all the coordinates are defined in the same datum.

OpenMap can display raster data (images) in a map, but it has to be noted that images have their own projection. For a raster background image to be used accurately in OpenMap, the raster's projection parameters must match the OpenMap projection parameters. When raster images are placed in OpenMap, the corners of the image are forward projected (located on the map), and the image is drawn on the map. If the two projections don't match, the features in the raster image won't be located in the correct location on the map. If the raster image has a projection that doesn't match an OpenMap projection, there are two things that need to be done to resolve the mismatch. A new OpenMap projection component can be written to duplicate the projection of the raster image, or the image can be re-projected to match a current OpenMap projection.

4 Event Handling

OpenMap components communicate with each other using the Java event model. Information is passed from one component to another using event objects. A component registers itself as an event listener to the other component that generates events. Whenever the source component generates a new event, it sends that event to all of its listeners. This model is used throughout OpenMap communications.

There are different types of events stored in the `com.bbn.openmap.event` package. As an example, the NavigatePanel (`com.bbn.openmap.gui.NavigatePanel`) generates `com.bbn.openmap.event.PanEvents` and `com.bbn.openmap.event.CenterEvents` when its buttons are pressed. The arrow buttons on the NavigatePanel generate PanEvents. PanEvents contain information that specify a direction and a distance. The middle button of the NavigatePanel generates a CenterEvent, which simply a latitude/longitude location.

The MapBean (`com.bbn.openmap.MapBean`), which is the main map window component in OpenMap, implements the PanListener interface:

```
public interface PanListener extends java.util.EventListener {
    public void pan(PanEvent evt);
}
```

and the CenterListener interface:

```
public interface CenterListener extends java.util.EventListener {
        public void center(CenterEvent evt);
    }
```

and can be connected to the NavigatePanel as a PanListener and a CenterListener:

```
    NavigatePanel navPanel = new NavigatePanel();
    MapBean mapBean = new MapBean();

    navPanel.addPanListener(mapBean);
    navPanel.addCenterListener(mapBean);
```

The `MapBean.pan()` method will be called by the NavigatePanel with a direction and distance which it interprets as a indication of where to pan over its map. The `MapBean.center()` method will also be called by the NavigatePanel with an event specifying a latitude and longitude to use to center itself.

This model is generalized and used throughout OpenMap, so it is important to understand the concept of listeners and events and how they are used to pass information between components.

5 Architecture Components

5.1 The MapBean, MapHandler (BeanContext) and MapPanel

As mentioned previously, the `com.bbn.openmap.MapBean` component is the main map window component in the OpenMap toolkit.

The **MapBean** derives from the `java.awt.Container` class. Because the MapBean is a Swing component, it can be added to a Java window hierarchy like any other user interface component. A MapBean contains a `com.bbn.openmap.proj.Projection` object that defines the geographic positioning of the map represented by the MapBean. The Projection is defined by a combination of the projection type, the latitude/longitude of the center of the window, a scale factor for the projection, and the pixel height and width of the window. Whenever a Projection object changes on a MapBean, it sends out an event (`com.bbn.openmap.event.ProjectionEvent`) to its listeners (`com.bbn.openmap.event.ProjectionListeners`).

To create the map in the MapBean, **Layers** (`com.bbn.openmap.Layer`) are added to the MapBean. Layers derive from the java.awt.Component class and they are the only type of object that can be added to a MapBean. Because the Layers are Components contained within a MapBean Container, the rendering of Layers onto the map is controlled by the Java component rendering mechanism. This mechanism controls how layered components are painted on top of each other. In addition to making sure that each component gets painted into the window in the proper order, the Component class also includes a method that allows it to tell the rendering mechanism that it would like to be painted. It is this feature that allows Layers to work independently from each other, and lets the MapBean avoid knowing what is happening on the Layers.

Layers are also ProjectionListeners, so when they are added to the MapBean they receive a ProjectionEvent whenever the map is panned, zoomed, or resized. The Layer is free to respond to the ProjectionEvent however it needs to. Its main responsibility is to be ready to render its contribution to the map whenever its `paint()` method is called, and if the projection dictates what it draws, it must only render to the map in its `paint()` method when it is ready. Other than holding Layers as Components in itself and passing along projection changes, the MapBean does not concern itself with what is being rendered on the map.

The `com.bbn.openmap.BufferedMapBean` extends the MapBean by forcing its layers to paint their map features into a buffered image. This drawing buffer is then rendered whenever the Java AWT thread is called to paint the Layers and that call is not at the request of a Layer. This dramatically increases performance for window exposes since it avoids the (potentially expensive) Layer painting process. If a layer requests to be painted, then the drawing buffer is regenerated by the Layers and painted into the map window.

The `com.bbn.openmap.BufferedLayerMapBean` extends the BufferedMapBean with a special internal image buffer that holds all layers that have been designated as 'background' layers. This buffer is especially useful when some layers are animating moving map features and the map is getting repainted often. Using a separate buffered image for background layers greatly reduces the amount of time and work needed to render the map, increasing the rate at which the map can be updated. By default, the OpenMap application uses the BufferedLayerMapBean instead of the MapBean precisely because of this increased performance.

While the MapBean is the main map window component in OpenMap, it is not the heart of the OpenMap architecture. The OpenMap architecture is based on the MapHandler (`com.bbn.openmap.MapHandler`) object. Understanding what the MapHandler does is one of the most important aspects of customizing an OpenMap application.

The **MapHandler** is a Java BeanContext (`java.beans.beancontext.BeanContext`), which can be thought of as a big bucket that can have objects added to or removed from it. The benefit of having a BeanContext object as the center of the architecture is that it will send events to listeners when its object membership changes. Any `java.beans.beancontext.BeanContextMembershipListener` added to a BeanContext will receive these events, and can use the events to set up or sever connections with the objects being added or removed.

The reason that the MapHandler exists, as opposed to simply using the BeanContext, is that it is an extended BeanContext that keeps track of SoloMapComponents. (`com.bbn.openmap.SoloMapComponents`). SoloMapComponent is an interface that can be used on an object to indicate that there is only supposed to be one instance of a component type in the BeanContext at a time. For instance, the MapBean is a SoloMapComponent, and there can only be one MapBean in a MapHandler at a time. A MapHandler has a `com.bbn.openmap.SoloMapComponentPolicy` that tells it what to do if it gets into a situation where duplicate instances of SoloMapComponents are added. Depending on the policy, the MapHandler will reject the second instance of the SoloMapComponent (`com.bbn.openmap.SoloMapComponentRejectPolicy`) or replace the previous component (`com.bbn.openmap.SoloMapComponentReplacePolicy`).

The MapHandler can be thought of as a map, complete with the MapBean, Layers, and other management components that are contained within it. It can be used by those components that need to get a handle to other objects and services. It can be used to add or remove components to the application, at runtime, and all the other objects added to the MapHandler get notified of the addition/removal automatically.

If you want your component to be told of the BeanContext, make it a BeanContextChild. It will get added to the MapHandler so that other components can find it, if it is on the `openmap.components` property list. If you are creating your own components programmatically, simply add the BeanContextChild component to the MapHandler yourself.

The `com.bbn.openmap.MapHandlerChild` is an abstract class that contains all the methods and fields necessary for an object to be a BeanContextChild and a BeanContextMembershipListener. If your object extends this class, you simply have to implement these methods:

```
public void findAndInit(Object obj);
```
Called whenever an object is added to the MapHandler.

```
public void findAndUndo(Object obj);
```
Called whenever an object is removed from the MapHandler.

You can perform tests on the object that is delivered to this method so you can and adjust your component accordingly. Make sure your component is stable if it doesn't find what it needs. You shouldn't assume that the other objects are going to be added in any particular order, or even added at all. You should also check that when objects are removed that the instance of the object is the same that is being used by your component before you disconnect from it (not just the same class type). As a MapHandlerChild, your component can be added to the OpenMap application without recompiling any OpenMap source code.

The **MapPanel** Interface (`com.bbn.openmap.gui.MapPanel`) describes a component that contains a MapBean, MapHandler, and a set of Menus that can be retrieved as a Swing JMenuBar or as a JMenu with sub-menus.

The **BasicMapPanel** (`com.bbn.openmap.gui.BasicMapPanel`) is one implementation of the MapPanel interface, and is used by the default OpenMap application. It is an extension of JPanel with a BorderLayout, and can be placed in any Java 2 application. The BasicMapPanel uses the MapHandler to locate MapPanelChild classes, which are able to tell the BasicMapPanel where they would like to be located within the BorderLayout.

In the OpenMap application, the openmap.properties file has an `openmap.components` property that lists all the components that make up the application. To change the components in the application, edit this list.

5.2 The LayerHandler

The **LayerHandler** object is used in the OpenMap application to manage layers - both those visible on the map, and those available for the map. The LayerHandler uses the `Layer.isVisible()` attribute to decide which layers are active on the map. It has methods to change the visibility of layers, add layers, remove layers, and change their order, and sends out events notifying listeners when the list of available layers has changed. It does not have a user interface, so it can be used with any application.

In the OpenMap application, the LayerHandler creates the layers available for the map based on the `openmap.layers` property in the `openmap.properties` file. Modifying the openmap.layers property lets you add and remove layers from the application. OpenMap layers have their unique properties that can be set to initialize them listed in the layer's JavaDocs. See Appendix C for an example of an openmap.properties file

5.3 Using MouseEvents with the MouseDelegator and MapMouseModes

MouseEvents are event objects that are generated when mouse actions are made over a Java component visible in a window. The MouseEvents describe what kind of action took place (mouse moved, clicked, released, dragged, and entered/exited window) and where in the window it happened. The OpenMap architecture supports the management of the distribution of these events, directing them to components in the application. Layers and other tool components can use these events to interpret and respond to user gestures over the map, displaying more information about map features, modifying the location of the features, or configuring tools for analysis queries.

MapMouseModes describe how MouseEvents and MouseMotionEvents are interpreted and consumed. Layers can use MapMouseListeners to subscribe to receive events from particular MapMouseModes, so it is possible to add some control over the conditions for which layers respond to MouseEvents at any given time.

The MouseDelegator is responsible for controlling which MapMouseMode is the MouseListener (java.awt.event.MouseListener) and MouseMotionListener (java.awt.event.MouseMotionListner) to the MapBean. The MouseDelegator manages a list of MouseModes, and knows which one is 'active' at any given time. The MouseDelegator listens for events from the MapBean, which tell it which layers have been added to the map. When it gets that list of layers, the MouseDelegator asks each layer for their MapMouseListener, and adds those MapMouseListeners to the MapMouseModes specified by the listener.

When a MouseEvent gets fired from the MapBean to the active MapMouseMode, the mode starts providing the MouseEvent to its MapMouseListeners. Each listener is given the chance to consume the event. A MapMouseListener is free to act on an event and not consume it, so that it can continue to be passed on to other listeners. The MapMouseListeners from the layers on top of the map are given a change to consume the MouseEvent before those on the bottom part of the map.

From a Layer's point of view, it has a method where it can be asked for its MapMouseListener. A layer can implement the MapMouseListener interface, or it can delegate that responsibility to another object, or can just return null if it's not interested in receiving events (the Layer default). The MapMouseListener provides a String array of all the MouseMode ID strings it is interested in receiving events from, and also has its own methods that the MouseEvents and MouseMotionEvents arrive in. The MapMouseListener can use these events to find out if events have occurred over any map features, and respond if necessary.

The MouseEvent delivered by a MapMouseMode to its listeners is actually a subclassed `com.bbn.openmap.event.MapMouseEvent`. The MapMouseEvent has these methods added for convenience:

```
public com.bbn.openmap.LatLonPoint getLatLon();
```
Provides the latitude and longitude for the MouseEvent's x/y position.

```
public com.bbn.openmap.event.MapMouseMode getMouseMode();
```
Provides the MapMouseMode that sent the MapMouseEvent.

5.4 Using Properties and the PropertyHandler

Properties (`java.util.Properties`) objects are used heavily by OpenMap components for configuration, and an interface was developed for OpenMap components use properties. The **PropertyConsumer** interface (`com.bbn.openmap.PropertyConsumer`) can be implemented by any component that wants to be able to configure itself with a java.awt.Properties object. It also has methods that let it provide information about the properties it can use, and what they mean.

In Java, Properties are a set of key-value pairs, each defined as Java Strings. The com.bbn.openmap.util.PropUtils class has methods that can be used to translate the value Java Strings into Java primitives and objects, like ints, floats, booleans, Color, etc.

Several PropertyConsumers may have their properties defined in a single properties file, which is what happens when the OpenMap application uses the `openmap.properties` file. In order for each PropertyConsumer to be able to figure out which properties are intended for it, the PropertyConsumer can be given a unique scoping property prefix string. In the `openmap.properties` instructions, this scoping string is referred to as a marker name. If the property prefix is set in a PropertyConsumer, it should prepend that string to each property key, separating them with a period. For example a layer may have a property key called `lineWidth`, which tells it how thick to draw its line graphics. If it is given a property prefix of layer1, it should check its properties for a `layer1.lineWidth` property. If the layer is given a `null` prefix (default), then it should look for a `lineWidth` property.

The methods for the PropertyConsumer are:

`public void setPropertyPrefix(String prefix);`
Set the scoping prefix.

`public void setProperties(Properties props);`
Provide the properties, with a null prefix.

`public void setProperties(String prefix, Properties props);`
Provide the properties with a specified prefix.

`public void getProperties(Properties props);`
Set the current values of the properties in the Properties object provided. If Properties is null, create one to fill. The keys in this Properties object should be scoped with a prefix if one is set.

`public void getPropertyInfo(Properties props);`
Set the metadata for the properties in the Properties object. Again, if Properties is null, create one and fill it. The keys in this Properties object should NOT be scoped, and the values for the keys should be a short explaination for what the property means. The PropertyConsumer may also provide a 'key.editor' property here with the value a fully qualified class name of the `com.bbn.openmap.util.propertyEditor.PropertyEditor` to use to modify the value in a GUI, if needed.

PropertyConsumers can use the `com.bbn.openmap.util.propertyEditor.Inspector` to provide an interface to the user to configure it at runtime. It also allows the

PropertyConsumer to provide its current state for properties files being saved for later use.

The **PropertyHandler** (`com.bbn.openmap.PropertyHandler`) is a component that uses an `openmap.properties` file to configure an application. It can be told which file to read properties from, or left to its own devices it will try to find an openmap.properties file in the Java ClassPath and in the application user's home directory. To do this, the `openmap.components` property contains a marker name list for objects. Each member of the list is then used to look for another property (`markername.class`) which specifies the classes to be instantiated.

If the PropertyHandler is given a MapHandler, it will load the components it creates into it after they are created. Intelligent components (MapHandlerChildren) are smart enough to wire themselves together. Order does matter for the `openmap.components` property, especially for components that get added to lists and menus. Place the components in the list in the order that you want components added to the MapHandler.

You'll notice that the application class (`com.bbn.openmap.app.OpenMap`) is pretty basic, using the PropertyHandler to instantiate all the components and add them to the MapHandler.

When the OpenMap application is creating objects from the `openmap.components` property, the marker name on that list becomes the property prefix for components. The ComponentFactory, which creates the components on behalf of the PropertyHandler, checks to see if the component is a PropertyConsumer, and if so it calls setProperties(prefix, properties) on it to let the component configure itself.

5.5 Various GUI Controls, the ToolPanel and Menus

The OpenMap toolkit contains many application widgets that can be used to control the map and the layers. These widgets can usually be found in the `com.bbn.openmap.gui` package.

The **ToolPanel** is a JToolBar used to place icons and simple controls in ready view of the user. The OpenMap application places a ToolPanel right above the map. The ToolPanel uses the MapHandler to locate objects that implement the Tool interface, simply adding them to itself as it find them. The **OMToolComponent** is a convenience class that implements the Tool, PropertyConsumer and LightMapHandlerChild interfaces, allowing any derived class the ready ability to show up in the ToolPanel, be configured with properties, and find other components in the MapHandler. Some OMToolComponents include:

- The **NavigatePanel** provides a set of arrows used to pan the map in 8 different directions. The NavigatePanel uses the MapHandler to find the MapBean, which listens for PanEvents from the NavigatePanel buttons.

- The **ZoomPanel** provides a pair of buttons for zooming in and out.
- The **ScaleTextPanel** provides a text field for directly setting the current scale.
- The **OverviewMapHandler** rovides a smaller, zoomed-out map to show what the main MapBean is displaying. The OverviewMapHandler can be directly added to any other component, but for the Tool interface, it provides a button which brings up the overview map in a separate window.
- The **ProjectionStackTool** provides two buttons which keep track of a ProjectionStack used to revert back to past projection settings and then forward to the most current projection settings. In order for the ProjectionStackTool to work with the MapBean projections, a `com.bbn.openmap.proj.ProjectionStack` object must also be added to the MapHandler.

The OpenMap application has traditionally used the **MenuBar** to manage its menus. The MenuBar is a `javax.swing.JMenuBar` that uses the MapHandler to find and add **MenuBarMenu** objects. This approach has been replaced in OpenMap 4.6 with the introduction of the `com.bbn.openmap.gui.menu.MenuList` object, which is responsible for creating Menus and providing them access to the MapHandler components they need to function within the application. The **MenuList** is capable of providing a JMenuBar or a JMenu containing its menus, and using it instead of the MenuBar has the great advantage over the MenuBar in that it maintains the defined order of the menus. The BasicMapPanel uses the MapHandler to locate a MenuList to use to manage its menus. There are some other useful class that can be used for menus:

The **AbstractOpenMapMenu** is a JMenu abstract class that has been enhanced with PropertyConsumer and LightMapHandlerChild methods.

The **OMBasicMenu** is an extension of the AbstractOpenMapMenu that has the ability to have its MenuItems and separaters defined in a properties file. If it has any MenuItems that extend the MapHandlerMenuItem class, it will automatically take care of giving them access to objects found in the MapHandler.

The **InformationDelegator** (`com.bbn.openmap.InformationDelegator`) is a central component used for communicating messages to the user. The InformationDelegator listens to the MapBean for message on when the active layers change, and connects itself to those layers in order to respond to their requests for tooltips over the map, for presentation of information text in different text areas, for dialog messages to be popped up, and for handling request to display information in a browser. As a MapPanelChild, it usually asks to be placed underneath the map. The InformationDelegator by default only has two text areas that it uses to place concise information around the map, one for coordinate information usually provided by the MapMouseModes, and one for information about data on the map that the user is gesturing over. More text areas can be programmatically added to the InformationDelegator, with requests directing new information to be displayed in those areas.

The **MiniBrowser** is a lightweight browser widget used to display html. It’s used by the InformationDelegator to display web pages and other HTML text requests.

The **OpenMapFrame** is a simple JFrame widget that knows how to use a MapPanel for displaying a map. It can interpret properties to position and size itself on the computer screen.

6 Displaying Data with OMGraphics

OMGraphics are the main classes used to represent data objects on the map. They are managed by Layers as objects, and know how to work with Projections to figure out where they should draw themselves in a map window. They can be dynamically repositioned, change their appearance under various conditions, and are able to respond to simple spatial queries about map pixel locations and their distance away from points.

In the latest version of OpenMap, OMGraphics are using java.awt.Shape objects to represent themselves on the map window. This has the benefit of allowing OMGraphics to provide Shapes that can be used in Java 2D API spatial queries and operations.

6.1 OMGraphic Classes, Render types, Line types

OMGraphics come in different flavors, depending on the shape you are trying to render on the map - OMArc, OMCircle, OMGrid, OMLine, OMPoint, OMPoly (including the OMDistance and OMSpline subclasses), OMRasterObject (including the OMBitmap, OMRaster, OMScalingRaster and OMScalingIcon subclasses), OMRect and OMText.

OMGraphics can be rendered in three ways, which are represented by their `renderType`. Choosing the render type property greatly affects how the object will react to projection changes.

- RENDERTYPE_LATLON means that the object should be placed on the map in its lat/lon location. You should expect the OMGraphic to scale the object as the map scale changes, and the object's location on the screen will change as the map location changes.
- RENDERTYPE_XY means the object should be placed at a screen pixel location on the map. The OMGraphic does not move or scale as the map projection changes.
- RENDERTYPE_OFFSET means the object should be placed at some screen pixel location offset from a lat/lon coordinate. The object will move with the map location changes, but will not scale if the map scale changes.

There are three different line types associated with OMGraphics that have lines rendered in lat/lon space. These setting do not affect OMGraphics with RENDERTYPE_XY or RENDERTYPE_OFFSET:

- LINETYPE_STRAIGHT means the lines will be straight on the screen between points of the OMGraphic, regardless of the projection type.
- LINETYPE_GREATCIRCLE means that the lines drawn between points of the OMGraphic will be the shortest geographical distance between those points.
- LINETYPE_RHUMB means that the line drawn between points of the OMGraphics will be of constant bearing, or going in the same direction.

6.2 Basic OMGraphic Types

Using the different OMGraphic types available, or combinations of these types, almost any information can be represented on a map.

6.3 OMArc and OMCircle

An OMCircle is a representation of a circle or ellipse on the map. The OMArc represents a slice of an OMCircle, and you can specify the starting angle and length of the arc in decimal degrees. The OMCircle is a subclass of the OMArc, where the starting angle is always zero and the length is always 360 degrees. OMCircles defined in x/y space can be rotated.

In X/Y pixel space, the OMCircle class can represent an ellipse. An ellipse cannot be represented in Lat/Lon space at this time.

An OMDistance object is a representation of a path in Lat/Lon space.

6.4 OMGrid

An OMGrid object is a representation of two-dimensional, equal-spaced data. The OMGrid relies on a `com.bbn.openmap.omGraphics.grid.OMGridGenerator` class to interpret the data for a particular projection and create an OMGraphic to use to represent the data.

6.5 OMLine and OMArrowHead

An OMLine object represents a path between two points over the map. OMLines can use OMArrowHead objects to turn themselves into arrows. Arrowheads can be placed at the front and back of the line.

6.6 OMPoly, OMDistance, OMSpline and OMDecoratedSpline

OMPoly objects represent a path between several points. An OMPoly can be closed (polygon) or open (polyline), and the fill paint is used to determine whether an OMPoly is open or closed.

An OMDistance object is an OMPoly rendered in lat/lon space that has labels on it displaying both the ground distance between points and the cumulative distance over the entire length.

The OMSpline and OMDecoratedSpline objects are new to OpenMap 4.6, and have an algorithm to add curves between the points on the polygon. The OMDecoratedSpline takes this one step further by allowing the curved polygon edge to be decorated with a symbolic edge or text.

6.7 OMRect

The OMRect object represents a box area inside a set of coordinate boundaries.

6.8 OMRasterObject family, including OMBitmap, OMRaster, OMScalingRaster, and OMScalingIcon

The OMRasterObject is a superclass for OMGraphics that use an image to represent themselves. The OMRasterObject knows what area its image covers. OMRasterObjects are located according to their upper left corner pixel, and painted accordingly upon the map. OMRasterObjects can be rotated.

OMBitmaps are two-color images created from data that species whether the pixels in the image are on or off. Pixels that are on are represented by the edge paint, and pixels that are off are represented by the fill paint. OMBitmaps can be used in conjunction with the `com.bbn.openmap.image.XBMFile` class, which can read X bitmap text files to create these images.

OMRaster is the main class for displaying images. OMRasters can be created from pixel data containing ARGB color information, pixel information to be derived from a color lookup table, or from a `java.awt.Image` object created from an image file. Any image file that the Java JVM can read (JPG, GIF, PNG, TIFF) can be given to an OMRaster object to be displayed on the map.

The OMScalingRaster is a subclass of OMRaster that, in addition to letting you specify the map location of the upper left corner, also lets you specify the location of the lower right corner of the image. When given a projection, the OMScalingRaster will automatically scale the image so the two corners are located appropriately. The OMScalingRaster does NOT handle warping of the image to make all of the pixels of the image appear in the proper location on the map, it simply does straight, simple scaling. Any projection mismatch between the OpenMap map projection and the projection of the image will have to be handled in another manner.

The OMScalingIcon is a subclass of OMScalingRaster that can be used to represent an object on the map. It centers its image over its location as well as scaling the icon appropriately for the given projection scale. It can be given a scale value that it uses to decide when to represent itself at full size, as well as a minimum and maximum scale where it will stop scaling itself, in order to prevent itself from becoming too small or too large.

6.9 OMText

The OMText object can be used to put text on the map, for labels and other information. OMText objects can hold multi-line text, and can be rotated. The font and justification of the text can be set as well.

6.10 OMGeometryList and OMAreaList

The OMGeometryList is a special form of OMGraphicList, one that contains OMGeometry objects. The com.bbn.openmap.omGraphics.OMGeometry interface describes the fundamental geometry of an object on the map, without any rendering attributes such as color. The OMGraphic class implements this interface, and adds attributes that control how they are painted on the map. An OMGeometryList is a collection of OMGeometry objects that are managed together and painted alike. When an OMGeometry list is generated, all of the OMGeometries on the list are combined to make a single java.awt.Shape object of separate parts. When the OMGeometryList is rendered, all of the drawing attributes set on the list, like line and file colors, are applied to its Shape object.

An OMAreaList is a class derived from the OMGeometryList with a special twist. An OMAreaList can accept different types of OMGeometries/OMGraphics, and connect them together to make a single area.

6.11 OMGraphics and Rendering Attributes

OMGraphics have attributes that allow you to control how they are rendered on the map:

- Line Paint. The `java.awt.Paint` used for the edge of the OMGraphic can be individually controlled.
- Select Paint. A secondary Paint for the edge when the OMGraphic is selected. The OMGraphic has `select()` and `deselect()` methods, which by default toggle the line paint and select paint when rendering the OMGraphic's edge. The `select()` and `deselect()` methods can be overridden to make any other desired modification.
- Fill Paint. A Paint for the interior of the area of the OMGraphic. The `com.bbn.openmap.omGraphics.OMColor.clear` object is the default fill Paint. TexturePaint and GradientPaint objects can be used, too, for adding patterns and effects.
- Texture Paint. A textured mask pattern for the OMGraphic. If not null, then it will be rendered on top of the fill paint. If the fill paint is clear, the texture mask will not be used. If you just want to render the texture mask as is, set the fill paint of the graphic instead. This is really to be used to have a texture added to the graphic, with the fill paint still influencing appearance.
- Matting and the Matting Paint. A setting (and a specific Paint) for adding additional color around the edge, to enhance the line Paint against a busy background.
- Stroke. Controls how the edge is drawn, specifying line width, dash pattern, end and corner accents, or customizable patterns.

The `com.bbn.openmap.omGraphics.DrawingAttributes` object is an object that knows how to manage these different types of attributes. The DrawingAttributes object can be used by a layer to read properties for attribute settings, and can be used to push attributes to and pull attributes from OMGraphics. It has a GUI interface that lets it provide editors for various attribute adjustments.

6.12 OMGraphic Management using the OMGraphicList

An OMGraphicList is a wrapper class that contains a `java.util.List,` and can be used to contain and manage OMGraphics. The OMGraphicList can control the order which its OMGraphics are rendered, find OMGraphics closest to a pixel location, find the OMGraphic that contains a pixel location and move OMGraphics to different places in the list.

OMGraphicLists are also OMGraphics, which allow them to be nested. OMGraphicLists also have the capability to set their behavior regarding how accessible their contained OMGraphics are. An OMGraphicList has the capability to be vague about its contents or completely transparent. If it's vague, the OMGraphicList behaves as if all of its parts contribute to one OMGraphic, and will return itself for any distance queries. A non-vague setting will cause the OMGraphicList to pass all queries onto its contained OMGraphics. Calling OMGraphic methods on the list (`setLinePaint()`, `setFillPaint()`, `setStroke()`, etc.) will cause all of the OMGraphics on the list to be configured with that value. To generate the OMGraphics on a list with the current projection, or to render them, `generate()` and `render()` can be called on the OMGraphicList.

6.13 The Generate and Render Paradigm

There is an important paradigm you have to work in with OMGraphics. Once an OMGraphic is created, it MUST be projected, which means that its position on the map needs to be calculated. This is done by taking the Projection object that arrives in the ProjectionEvent, and using it on the `OMGraphic.generate(Projection)` method. If the projection changes, the OMGraphics will need to be `generated()`. If the position of the OMGraphic has changed, or certain attributes of the OMGraphic are changed, the OMGraphics needs to be generated. The OMGraphics are smart enough to know when a attribute change requires a generation, so go ahead an call it, and the OMGraphic will decide to do the work or not. If you try to render an OMGraphic that has not been generated, it will not appear on the map. After the OMGraphic is generated, the `java.awt.Graphics` object that arrives in the `Layer.paint(Graphics)` method can be passed to the `OMGraphic.render(java.awt.Graphics)` method. See the layer section below for more information about managing the Projection object for use with your OMGraphics.

The `java.awt.Shape` object used internally on OMGraphics is created when the OMGraphic is projected. This Shape object only has meaning in the context of the projected map on the window. Once the projection changes, new Shape objects must be created inside the OMGraphic objects.

This is the most common problem encountered when the location of an OMGraphic is changed. If the OMGraphic is not appearing on the map, but is having render() called on it in the Layer's paint() method, it's likely that the OMGraphic thinks it needs to be regenerated with the current projection.

6.14 Using MouseEvents with OMGraphics

When viewed from the level of the OMGraphic, MouseEvents are simply pixel locations over the map window. OMGraphics have several query methods that layers can use to determine of what the user is doing with the mouse over the map. The OMGraphic must be generated in order for these methods to have any meaning:

```
public float distance(int x, int y);
```

Given a horizontal and vertical pixel coordinate, provide the pixel distance away from the OMGraphic. If the OMGraphic is solid (has a fill paint) and the coordinate is inside the area of the OMGraphic, the distance will be zero.

```
public float distanceToEdge(int x, int y);
```

Same as distance, but always returns a distance to the edge of the OMGraphic, even if the coordinate is inside the area of the OMGraphic.

```
public boolean contains(int x, int y);
```

Returns true if the coordinate is inside the OMGraphic.

The OMGraphicList also provides methods that allow querying of MouseEvent coordinates over its OMGraphics.

```
public OMGraphic findClosest(int x, int y);
public OMGraphic findClosest(int x, int y, int limit);
```

Ask the OMGraphicList for the OMGraphic closest to the coordinate. In the second method, the query only returns an OMGraphic if the distance is within the provided pixel distance limit. Both methods return null of an OMGraphic isn't found.

```
public int findIndexOfClosest(int x, int y);
public int findIndexOfClosest(int x, int y, int limit);
```

Ask for the index on the list of the OMGraphic closest to the coordinate. In the second method, the query only returns an index if the distance is within the provided pixel distance limit.

```
public OMGraphic selectClosest(int x, int y);
public OMGraphic selectClosest(int x, int y, int limit);
```

Ask the OMGraphicList for the OMGraphic closest to the coordinate, except also call the `select()` method on the OMGraphic. This method is convenient because it also calls `deselect()` on all of the other OMGraphics that are not returned. In the second method, the query only returns an OMGraphic if the distance is within the provided pixel distance limit. Both methods return null of an OMGraphic isn't found.

```
public OMGraphic getOMGraphicThanContains(int x, int y);
```

Ask for the OMGraphic that contains the coordinate.

Using these methods on the OMGraphicList and OMGraphics, layers can be built that respond to user mouse movements over the map. These movements can trigger changes in the appearance of the OMGraphics, or trigger more information to be displayed about what the OMGraphic represents.

The OpenMap Layer class has convenience methods that let it directly send information display requests to the InformationDelegator. The InformationDelegator is always listening to layers that have been added to the MapBean, and can display tooltips over the map, display information in it's label line along the bottom of the map, and provide popup message dialogs and browser window content to the user.

6.15 The OMGraphicHandler Interface and FilterSupport

The OMGraphicHandler (`com.bbn.openmap.omGraphics.OMGraphicHandler`) is an interface describing an object that manages OMGraphics and is able to send or receive an OMGraphicList. It provides a mechanism to filter what OMGraphics are displayed based on some criteria, and can also return graphics based on those criteria. It is assumed that when a filter is applied to an OMGraphicHandler, it stays in place until `resetFiltering()` is called. This assumption means that subsequent calls to `filter()` works on the result of the previous calls.

The `com.bbn.openmap.omGraphics.FilterSupport` object provides a basic implementation of the OMGraphicHandler interface, and can be used by layers to manage, share and filter OMGraphics. By responding to spatial filter calls, it can return OMGraphicLists containing OMGraphics that are inside or outside specified areas.

6.16 Compound OMGraphics, Extensions and Customization

If your data requires more complex representation on the map, you have several ways of creating compound OMGraphics.

You can create customized OMGraphics by combining standard OMGraphics into a new object that contains an OMGraphicList, setting the OMGraphicList to be vague. Beyond adding the OMGraphics you need to your OMGraphicList, you can also add customized methods to help manage the new attributes of the list.

You can also create a custom OMGraphic object that contains other OMGraphics. The OMGraphic class has three methods you need to think about:

`public boolean generate(Projection proj);`
    You have to make sure all the parts of the list are projected.

`public void render(java.awt.Graphics g);`
    You have to make sure all the parts are drawn.

`public float distance(int x, int y);`
    Given a pixel coordinate, you have to check with the parts to return the pixel distance of the closest part from that coordinate.

The `com.bbn.openmap.layer.location.Location` object is an example of creating an object to contain OMGraphics, combining a generic OMGraphic marking a location, and a OMText object for the location label. Each part is separately managed in the Location's generate and render methods.

OMGraphics contain an object reference that can be accessed with setAppObject() and getAppObject(). This can be really handy to use to store more information about the data you are representing with that OMGraphic (web page addresses, additional attributes, etc). You can always extend the standard OMGraphics to make them contain the particular features you want.

7 Displaying Data on the Map

Layers are generally responsible for displaying data on the map. Since they are the only component type allowed to be added to the MapBean, any component providing OMGraphics for display on the map needs to have a Layer do that work for it.

OpenMap doesn't place any limitations on how data is gathered for the transformation into OMGraphics. The component architecture chosen for this task depends on the availability of the data, the storage capabilities and network connection of the application host, and the limitations of the Java environment that OpenMap is running inside.

The easiest way to write components that are flexible in how they read data files is to use the `com.bbn.openmap.io.BinaryFile` and it's buffered subclass, the BinaryBufferedFile. These classes are used by OpenMap layers to read binary files, and are flexible enough to hide the differences between accessing files locally though the file system, as a URL, or contained in a jar file. Reading the file from the local file system is the fastest and most efficient way to read the file, but the file system isn't always accessible to all Java environments (like Applets). Reading the file from a URL is good for smaller files or for situations where the data will be held in memory and it can make distribution easier, but again, applet environments are limited to reading URL data files only from the applet server host. Storing data in a jar file can make application packaging easier, but some users may want to use other data files and not want to have to deal with repackaging the data in a jar file. Using the BinaryFile can make all of these configurations possible with the same component simply by changing how the file is referenced. If the component is written to use properties, that can be done by changing the contents of the properties file.

If the application is running on a system with network connectivity, layers and other components acting as clients to remote servers can create OMGraphics. The OpenMap toolkit contains several packages that support client-server connections for transferring OMGraphics.

- The `com.bbn.openmap.layer.specialist` package contains a definition and support for a CORBA client-server implementation. The package contains a generic client layer that supports gesturing over the client map objects, and a server framework for creating servers that handle data from different sources.
- The `com.bbn.openmap.layer.link` package contains a socket client-server protocol and support components. The package also contains a generic client layer supporting gesturing over the map objects and a server framework for creating server that handle data from different sources. There is also language support for creating servers written in the C language that can communicate with the client layer.
- The `com.bbn.openmap.image` package contains classes that support the creation of image servers, where remote OpenMap layers can be used to create a map image to be retrieved from a http server. The SimpleHttpImageServer is a lightweight example of an http server responding to requests for map images. The `com.bbn.openmap.plugin.shis.SHISPlugin` is a client that communicates with the SimpleHttpImageServer.
- The `com.bbn.openmap.plugin.wms.WMSPlugIn` can receive images from a server conforming to the OpenGIS Consortium's (OGC) Web Map Service specification.

Other layer implementations, supporting specific data formats, will be described later in this document.

7.1 Layer Basics

Layers can do anything they want to in order to render their data on the map. When several layers are added to a map, the Java AWT painting mechanism takes care of making sure that each layer is rendered in the correct order.

Layers are PropertyConsumers, which means they can be configured with Java Properties. When writing layers, it's a good idea to avoid hard-coding configuration settings into the layer. The PropertyConsumer methods can be used to initialize those settings. The `com.bbn.openmap.util.PropUtils` class has many convenience methods that can be used to easily check for and translate property values into Java primitive values.

Layers can also use the MapHandler to find other components they need in the application. Layers are not added to the MapHandler by default however. To add a layer to the MapHandler the `addToBeanContext` property needs to be set to true in the layer's properties, or set to true programmatically using the `setAddToBeanContext(boolean)` method. After that property is set, the layer code needs to simply override the `findAndInit(Object)` and `findAndUndo(Object)` methods to attach to and detach from other components in the application.

7.2 Projection Changes and Painting

When a Layer is added to the MapBean, it automatically gets added as a ProjectionListener to the MapBean. That means that when the map changes, the layer will receive a ProjectionChanged event, letting it know what the new map projection looks like. It's then up to the layer to decide what it wants to draw on the screen based on that projection, and then call `repaint()` on itself when it is ready to have its `paint()` method called.

The one thing most OpenMap layers have in common is the use of the SwingWorker (`com.bbn.openmap.util.SwingWorker`), which is a thread-launching object. It's used by layers to spawn a thread when a new projection is received, in order to let the Java AWT thread that the ProjectionChanged event arrived on be free to move to the next ProjectionListener. Any layer that does a significant amount of work when the projection changes should launch a new thread to do the work of gathering, creating and generating the OMGraphics for the new projection. Using the Java AWT thread to do calculations will make the application appear slower, because the user interface will be unresponsive if the Java AWT thread is doing that other kind of work.

It should be noted that the `Layer.paint()` method doesn't always get called just when the Layer calls repaint() on itself. The Java AWT thread can call it for map window repaints or when another layer asks to be repainted. The Layer needs to be prepared to always render what is appropriate for the latest projection it has received. If it isn't ready to render based on the last projection, it shouldn't render anything. Most importantly, the layer should not do more than simply render its contribution to what is currently on the map, in order to take up as little time as necessary with that Java AWT thread. Having the layer do work in the `paint(java.awt.Graphics)` method will make the overall application feel slower. OMGraphics are the prevalent objects used to represent map features on the map. As an alternative, layers can draw on the map by simply using the rendering methods on the java.awt.Graphics object. OMGraphics are a convenience, not mandatory mechanism.

If you want to save and reuse the projection that the layer receives in the ProjectionEvent, the `Layer.setProjection(ProjectionEvent)` method should be called from within the `Layer.projectionChanged(ProjectionEvent)` method. This method will return the Projection object extracted from the ProjectionEvent, and will also save it so it can be retrieved using the `Layer.getProjection()` method.

7.3 Layer User Interfaces (Palettes)

The Layer.getGUI() method provides a way for a layer to create its user interface which can control its attributes. The getGUI() method should just return a java.awt.Component, which means you can customize it any way you want. The parent Layer class returns null by default if you decide not to provide a user interface to control layer parameters.

The OpenMap application has GUI controls that provide a way for the user to bring up a layer's palette, on the com.bbn.openmap.gui.LayersPanel. Other controls can be added as well. A layer can also display its own palette by making programmatic calls on itself when needed.

7.4 The OMGraphicHandlerLayer

The OMGraphicHandlerLayer is a base layer class that has the most of the functionality built into it that you would need in a layer. Most of the other OpenMap layers extend OMGraphicHandlerLayer, which allow them to interact and share OMGraphics.

Use the OpenMap layers as examples of different ways to create and manage OMGraphics. The GraticuleLayer creates its OMGraphics internally, while the ShapeLayer reads data from a file. The DTEDLayer and RpfLayer have image caches. The CSVLocationLayer uses a quadtree to store OMGraphics. You can also access a spatial database to create OMGraphics. Any technique of managing graphics can be used within a layer.

As a base class, the OMGraphicHandlerLayer has built-in capabilities that make it easier to manage OMGraphics in a layer. When extending the OMGraphicHandlerLayer, the `prepare()` method is the most important layer to implement. The `prepare()` method returns an OMGraphicList containing the OMGraphics appropriate for the projection currently set in the layer. All the work gathering, preparing and generating the OMGraphics should be performed in the `prepare()` method. The OMGraphicHandlerLayer also has a built-in SwingWorker object that can be used to call `prepare()` in a separate thread. The SwingWorker thread can be started by calling the `doPrepare()` method. If the SwingWorker is already busy doing work when the `doPrepare()` method is called, a new thread will be launched to call `prepare()` when the original thread completes. In the default implementation of the `prepare()` method, the current list is simply generated with the current projection and returned.

7.4.1 Behavior Policies in the OMGraphicHandlerLayer

The OMGraphicHandlerLayer makes use of policy objects that allow certain behaviors to be modified depending on the desired behavior. The OMGraphicHandlerLayer uses a ProjectionChangePolicy to determine how it handles its OMGraphicList when a new projection is received:

- The StandardPCPolicy is the default ProjectionChangePolicy assigned to the OMGraphicHandlerLayer, and it assumes that the OMGraphicList contents are going to be reused for the new projection, and doesn't empty the list before `prepare()` is called.
- The ListResetPCPolicy is a policy that assumes that the `prepare()` method is going to gather new OMGraphics for the current projection, and it clears out the OMGraphicList so that if `repaint()` is called before the `prepare()` method is complete, nothing from the layer will be drawn.

The RenderPolicy determines how the OMGraphicHandlerLayer renders its OMGraphics objects into a `java.awt.Graphics` object, which is placing the layer contents on the map:

- The StandardRenderPolicy is the default RenderPolicy, and basically renders the OMGraphicList, handling null and empty lists gracefully.
- The BufferedImageRenderPolicy keeps track of how long it takes to render the OMGraphicList contents, and creates an image buffer if it feels the need. This greatly reduces the amount of time it takes to paint the contents of the layer when the projection isn't changing, i.e. if some other layer is animating the movement of its OMGraphics.
- The RenderingHintsRenderPolicy lets you set java.awt.RenderingHints on the java.awt.Graphics object that the OMGraphicList is being rendered into, allowing anti-aliasing and other effects to be set for the layer.

The policies for an OMGraphicHandlerLayer and layers that derive from it can be chosen and set both programmatically and by properties in the properties file.

7.4.2 The OMGraphicHandlerLayer and MouseEvents

The OMGraphicHandlerLayer is the first layer to implement the **GestureResponsePolicy** interface (`com.bbn.openmap.omGraphics.event.GestureResponsePolicy`) and use a **MapMouseInterpreter** to track mouse events for the GestureResponsePolicy. These interfaces provide a framework that interprets what the mouse is doing over the map as it applies to the OMGraphics held by a layer.

7.4.3 MapMouseInterpreters

The MapMouseInterpreter interface extends the MapMouseListener interface, and represents a MapMouseListener that has access to OMGraphics. The MapMouseListener receives MouseEvents through the standard MapMouseListener methods (mousePressed, mouseReleased, mouseClicked, mouseMoved, mouseDragged, mouseEntered and mouseExited), but then makes calls on itself describing how those events relate to its OMGraphics:

```
public boolean leftClick(OMGraphic omg, MouseEvent me);
```
The OMGraphic has been clicked on with the left mouse button.

```
public boolean leftClickOff(OMGraphic omg, MouseEvent me);
```
The user has clicked on something other than the OMGraphic that was previously left-clicked on.

```
public boolean rightClick(OMGraphic omg, MouseEvent me);
```
The OMGraphic has been clicked on with the right mouse button.

```
public boolean rightClickOff(OMGraphic omg, MouseEvent me);
```
The user has clicked on something other than the OMGraphic that was previously right-clicked on.

```
public boolean mouseOver(OMGraphic omg, MouseEvent me);
```
The mouse has been moved over the OMGraphic.

```
public boolean mouseNotOver(OMGraphic omg);
```
The mouse is no longer over the OMGraphic, which was previously reported as having the mouse over it.

The MapMouseInterpreter also handles events that pertain to the area not covered by OMGraphics:

```
public boolean leftClick(MouseEvent me);
```
The left mouse button has been clicked on a location not covered by an OMGraphic the MapMouseInterpeter is aware of.

```
public boolean rightClick(MouseEvent me);
```
The right mouse button has been clicked on a location not covered by an OMGraphic the MapMouseInterpeter is aware of.

```
public boolean mouseOver(MouseEvent me);
```
The mouse is over a location on the map not covered by OMGraphics the MapMouseInterpreter is aware of.

The MapMouseInterpreter can react to these notifications in different ways. These methods can be overridden to create dynamic responses, or a GestureResponsePolicy object can be set in the MapMouseInterpreter that can be used to decide what the actions mean and what actions need to be taken. The StandardMapMouseInterpreter is an implementation of the MapMouseInterpreter that uses GestureResponsePolicies in a particular way.

7.4.4 GestureResponsePolicies

The GestureResponsePolicy (GRP) interface describes queries, notifications and requests that a MapMouseInterpreter uses to react to its interpreted mouse events. There are query methods that let the interpreter ask if responses are appropriate for an OMGraphic:

`public boolean isHighlightable(OMGraphic omg);`

A query method where the MapMouseInterpreter asks if the GUI should change to let the user know that something will happen if the OMGraphic is clicked on, or if it should ask for more information to display about the OMGraphic.

`public boolean isSelectable(OMGraphic omg);`

A query method where the MapMouseListener asks if the OMGraphic can be moved or modified.

There are notification methods letting the GRP know that it should react to user gestures concerning OMGraphics:

`public void select(OMGraphic omg);`

A notification to the GRP to let it know that an OMGraphic has been selected. This method is called if the OMGraphic is selectable.

`public void deselect(OMGraphic omg);`

A notification to the GRP to let it know that an OMGraphic has been deselected.

`public void highlight(OMGraphic omg);`

A notification to the GRP to let it know that an OMGraphic should be highlighted in some way to let the user know that the current mouse position is targeting it.

`public void unhighlight(OMGraphic omg);`

A notification to the GRP to let it know that an OMGraphic should be changed back to its normal appearance.

There are request methods letting the GRP provide information about an OMGraphic to provide more information about it in the GUI:

`public String getToolTipTextFor(OMGraphic omg);`

A request to retrieve text to use for a tooltip over the OMGraphic.

`public String getInfoTextFor(OMGraphic omg);`

A request for more information to display in the GUI for an OMGraphic.

`public List getItemsForOMGraphicMenu(OMGraphic omg);`

A request for options to be provided for a popup menu displayed for an OMGraphic.

`public List getItemsForMapMenu();`

A request for options to be provided for a spot not over an OMGraphic, such as a menu that appears when the map is clicked on.

When and if these methods get called depends on the MapMouseInterpreter, and how it's been designed to react to different MouseEvents.

7.4.5 The OMGraphicHandlerLayer as a GestureResponsePolicy

The OMGraphicHandlerLayer implements the GestureResponsePolicy. Layers that extend OMGraphicHandlerLayer can override these methods to create consistent reactionary behavior across layers. By default, the OMGraphicHandlerLayer considers all OMGraphics highlightable but not selectable.

If the OMGraphicHandlerLayer responds to its `getMouseModeIDsForEvents`() method with something other that null, it will create a **StandardMapMouseInterpreter** to use as a MapMouseListener. The mouse mode IDs can be set in the properties file using the "`mouseModes`" property, where the values for the property are a space-separated list of mouse mode IDs that should providing events to the layer.

7.5 PlugIns

PlugIns are defined and managed with components in the `com.bbn.openmap.plugin` package. PlugIns are objects that simply access data sources to create and manage OMGraphics. There is a PlugInLayer (deriving from OMGraphicHandlerLayer) that uses PlugIns to get graphics on the map. A programmer would write a PlugIn so it responds to its `getRectangle(Projection)` method with an OMGraphicList appropriate for the projection. If the AbstractPlugIn is extended to create a customized PlugIn, the getRectangle() method is the only one that needs to be written. The idea behind a PlugIn is that data access and OMGraphic creation can be isolated into a single object, and then that object can be used by a PlugInLayer for local data access, or by a server for remote data access by another OpenMap layer.

The PlugIn can also provide a GUI through its getGUI() method. The AbstractPlugIn returns null by default. PlugIns can also provide and/or implement the MapMouseListener interface; the PlugInLayer passes the request through to the PlugIn.

As of OpenMap 4.4, PlugIns have been elevated to the same management level as Layers. For the OpenMap application, they can be defined the openmap.properties file the same way as layers, in the openmap.layers property. The LayerHandler, when parsing the list and creating a PlugIn, will automatically wrap a PlugInLayer around the PlugIn.

7.6 GraphicLoaders

The **GraphicLoader** interface is for components that control the position of OMGraphics on their own schedule. It is especially useful for creating connections to outside processes and applications that are using OpenMap to display data that is dynamic in nature and changing according to real-time/simulated events.

A GraphicLoader uses an OMGraphicHandler as something to receive an OMGraphicList when the GraphicLoader wants to update its OMGraphics on a map. It may be asked for a user interface component or a name for itself.

The OpenMap toolkit contains components that automatically create a layer for a GraphicLoader. If a **GraphicLoaderConnector** is added to the MapHandler it will monitor other objects added to the MapHandler, looking for GraphicLoaders that do not already have a receiver listening for updates. When it finds one, it will create a GraphicLoaderPlugIn to act as a receiver for the GraphicLoader, and attach it to a PlugInLayer.

The `com.bbn.openmap.graphicLoader` package has a couple of GraphicLoader classes defined to help with implementing the interface. The **AbstractGraphicLoader** is an OMComponent, implementing the PropertyConsumer interface and the MapHandlerChild methods. Extending this class makes it easy to create a GraphicLoader that can use the MapHandler to attach to other components, and to receive properties from the property file for configuration. It also contains a java.awt.Timer, so it can be configured to trigger itself at regular intervals to update its OMGraphicList. The MMLGraphicLoader is an extension of the AbstractGraphicLoader, implementing the MapMouseListener interface. This makes it possible to have a GraphicLoader respond to user gestures.

7.7 Modifying OMGraphics with the Drawing Tool

The OpenMap package contains a drawing tool component that lets the user to modify the location, shape and appearance of various OMGraphics. The **OMDrawingTool** (`com.bbn.openmap.tools.drawing.OMDrawingTool`) can be used to create or edit an OMGraphic, and then deliver that OMGraphic when the user is done. It can be added to the MapHandler so that any component can grab and use it.

In order to modify it, the OMDrawingTool wraps an OMGraphic in an `com.bbn.openmap.omGraphics.EditableOMGraphic` object. There are various EditableOMGraphics, each with the knowledge of how to manipulate specific types of OMGraphics. The EditableOMGraphic is responsible for placing visible grab points on top of an OMGraphic, and managing how an OMGraphic changes when those grab points are moved by the user.

The OMDrawingTool uses **EditToolLoaders** to figure out which EditableOMGraphic to use for an OMGraphic. An EditToolLoader provides information which OMGraphic classes are handled by an EditableOMGraphic. When an EditToolLoader is added to the MapHandler, an OMDrawingTool can find it and dynamically change what kinds of OMGraphics it can handle.

When a component calls `create()` or `edit()` on the OMDrawingTool to have it manage an OMGraphic, a handle to that OMGraphic is returned from those method calls. This presents an opportunity for the component to set specific parameters on the OMGraphic. While it might be tempting to save this handle and add the OMGraphic to an OMGraphicList, this handle should be released by the calling code. The `create()` and `edit()` methods take a **DrawingToolRequestor** as an argument which is notified when the OMDrawingTool is finished. This notification comes with an **OMAction** object (`com.bbn.openmap.omGraphics.OMAction`) that describes what the requestor should do with the OMGraphic. Handling the OMGraphic with the OMAction object allows the requestor to handle the situations where the OMGraphic has been deleted as well as being added.

The OMDrawingToolLauncher is a component that can direct the OMDrawingTool to create OMGraphics for different DrawingToolRequestors. The launcher uses the MapHandler to locate the requestors in the application, and listens to the OMDrawingTool for what OMGraphics it can create. There are a couple of points to note about this arrangement:

- The DrawingToolRequestor receiving the OMGraphic doesn't know anything about it until the OMDrawingTool provides it with the OMAction object.
- The DrawingToolRequestor doesn't have to be a Layer. It can be a proxy for a server where the OMGraphic is passed on and never seen again.

The DrawingToolLayer (`com.bbn.openmap.layer.DrawingToolLayer`) and the DemoLayer (`com.bbn.openmap.layer.DemoLayer`) are examples of how a layer can use the OMDrawingTool to edit OMGraphics. The DemoLayer also uses the drawing tool to create areas on the map which are used as filters to control which of its OMGraphics are visible.

The OMGraphicList.doAction() method automatically handles OMGraphics and OMActions. If a layer is implementing the DrawingToolRequestor interface, it simply needs to pass the OMGraphic and OMAction to its OMGraphicList and call repaint() on itself.

7.8 Animation Techniques

While the GraphicLoader seems like the automatic choice for animating OMGraphics over the map, any Layer or PlugIn can do the same thing. All that is required to create a moving OMGraphic is to change its location parameters, regenerate it with the current projection, and call repaint on itself. If the OMGraphic is not regenerated, it will disappear from the map.

In order to increase the refresh rate of the map while decreasing the amount of work that Java needs to do, the amount of work done in the paint method of the active layers needs to be reduced. This is especially important for layers that are simply drawing background information to the map.

The **BufferedLayerMapBean** is a MapBean that uses a BufferedLayer to create an image for some layers. It uses the Layer `background` flag to determine which layers should be added to this buffered image. Any layer that isn't responding to MouseEvents should be designated as a background layer to reduce its `paint()` method processing load on the application.

If the BufferedLayerMapBean isn't used, or the background flag of layers isn't being controlled, then OMGraphicHandlerLayers can use the BufferedImageRenderPolicy to buffer themselves and reduce their paint times. This is a setting that can be implemented in the openmap.properties file for the layer.

Another thing to consider when doing animation is control how often the animating layer calls for a map to be repainted. Instead of updating the map whenever an OMGraphic is changed, it's better to have a timer in place that calls `repaint()` in regular intervals, checking first to see if any change to the map is needed. Also, choosing larger time intervals may help making the application feel less burdened.

8 Displaying Supported Data Formats

The OpenMap package comes with several layer packages that are capable of displaying a variety of image types. The layers in these packages can also serve as an example for creating layers that may need similar caching or display functionality.

8.1 Location Data via CSV files or a Database

Often, data that needs to be displayed is made up of location data, or objects located at a specific place. OpenMap has a `com.bbn.openmap.layer.location.LocationLayer` that handles this data pretty easily. The LocationLayer manages composite OMGraphics called `com.bbn.openmap.layer.location.Location` objects. Location objects contain a marker OMGraphic that represents the actual object, and a OMText to display the name of that object.

How a LocationLayer loads Locations is abstracted to the LocationHandler interface, which is responsible for creating Locations from a particular data source. OpenMap comes with a two default implementations:

- The CSVLocationHandler can read comma-separated value (CSV) file. Since a CSV file is understood to be made up of columns of similar data, the CSVLocationHandler properties set which columns should be interpreted as latitudes, longitudes and display names. There is also the option of specifying a column as containing a URL for an image to use as an icon for the location
- The DBLocationHandler can be configured to issue SQL commands to a JDBC server to retrieve location information from a relational database. The DBLocationHandler expects that any query to the database returns a set of results in a particular order, and has a queryString property it will use for the SQL request that will make that happen. Knowledge of how the data is being stored in the database, including database and table names and configurations, are required in order to provide the property query string.

Other implementations of LocationHandler interface can be created and connected to the LocationLayer by setting the `locationHandlers` property for the LocationLayer. This property can be set for several LocationHandlers, which the LocationLayer can use at the same time.

8.2 ESRI Shape Data

There are several options available for displaying data files in the ESRI Shape file format. Shape files contain vector geometries (points, polygons and polyline) and are usually distributed in a set of three files: the geometry file with the *shp* name extension, the index file with the *shx* name extension, and the database attribute file with the *dbf* name extension. All three files share the same name, other than the extension attached.

The *shp* file contains all the geometry for objects that are going to be placed on the map. The Shape file specification doesn't specify or limit the coordinate system used for the geometries in the file, but OpenMap can ***only display shape files that have their geometries defined in decimal degree latitude/longitude data***. If the *shp* file contents are already projected and defined in terms of meters or other units, the OpenMap shape code will not be able to display them.

There are several things to consider when choosing which OpenMap Shape solution should be used, including the size of the file, where the file is located, whether you want to view the contents of the attributes and whether you want to render each feature differently.

The `com.bbn.openmap.layer.shape.ShapeLayer` displays all of the contents of the *shp* file with the same attributes. It's better suited for displaying large Shape files that are available on the local machine, when you aren't interested in the contents of the database file and simply want to display the data. All of the OMGraphics created for this layer are rendered in the same way. This layer creates a spatial index file that enables it to use the current map projection to display only the geometries on the map. Whenever the projection changes, this layer recalculates what geometries are needed, and re-reads those geometries from the data file. This layer makes the tradeoff of more I/O intensity against less memory consumption.

The `com.bbn.openmap.layer.shape.BufferedShapeLayer` extends the ShapeLayer, and simply reads the entire contents of the Shape file at once and holds on to the OMGraphics created for the geometries. These OMGraphics are simply regenerated when the projection changes. Again, all of the OMGraphics are rendered the same way. This layer makes a tradeoff opposite than the ShapeLayer, for less I/O intensity for more memory consumption.

The `com.bbn.openmap.layer.shape.MultiShapeLayer` lets you define several shape files that you want displayed at the same time in one layer. While all the contents of a particular shape file are rendered the same way, each shape file can have its own rendering attributes. The OMGraphics for this layer are managed in the standard ShapeLayer manner.

The `com.bbn.openmap.layer.shape.areas.AreaShapeLayer` lets you render geometry in a shape file differently depending on the contents of the dbf file. The properties for the file can be set to render different types of geometries in a specific way, where a particular attribute field in the database file is used as a key to determine how each geometry should be rendered. This layer can respond to mouse events to provide more information about individual geometries on the map, displaying name information of objects through the InformationDelegator. This layer holds all of the attribute information in memory while managing the OMGraphics for the geometries in the standard ShapeLayer manner.

The `com.bbn.openmap.plugin.esri.EsriPlugIn` uses classes in the `com.bbn.openmap.dataAccess.shape` package to read and write shape files. The EsriPlugIn reads the *shp* and *dbf* files, and can display the *dbf* file contents in a table available via the layer's palette. Clicking on the *dbf* table entry causes the map object to highlight, and clicking on a map object highlights the table entry. Individual geometries can be set to rendered differently at runtime, although those new rendering settings can't be saved between application sessions. This plugin makes the tradeoff for more memory consumption against less intensive I/O and the ability to query the *dbf* file contents.

8.3 National Geospatial-Intelligence Agency (NGA) Databases

NGA (formally the National Imagery and Mapping Agency, NIMA) is the US Government agency responsible for providing mapping data to the US Department of Defense, and it makes several databases available to the public. More information about these databases can be found on the NGA website (http://www.nga.mil), and data can be downloaded from the GeoEngine website http://geoengine.nima.mil. These databases include raster image databases, vector feature database and elevation database, and OpenMap includes layers that can display this data.

8.4 Raster Product Format (RPF), including CADRG and CIB

The `com.bbn.openmap.layer.rpf.RpfLayer` can be used to display RPF data, which is kept in a file and directory format used to create seamless, worldwide image databases.

RPF databases contain images created from satellite imagery, aeronautical charts, harbor charts, and other paper maps used by NIMA. All charts have a base scale, and the RpfLayer will scale the images to match the current map projection if the map scale is close enough to be appropriate.

In order to view RPF data with the RPFLayer:

- Displaying RPF data requires that the MapBean is using a CADRG projection, an equal arc projection with the pixel spacing specified in the RPF specification.
- The map has to be centered over the area where the RPF data covers. OpenMap is a seamless worldwide data viewer, and won't automatically zoom to where there is data. It shows what data it has for the given map location.
- Your map scale needs to be set to a value that fits with the RPF image type. The RpfLayer options provide different settings to modify aspects of this behavior, but in general, the RpfLayer will look at the data available and try to display the images that best match the current projection settings. For instance, if you are trying to view CADRG JNC charts, the best scale to view them is 1:2,000,000, which is the base scale of JNC maps. If you have no other chart types available, JNCs can be viewed within the scale range of 1:8,000,000 to 1:500,000. This range is controlled by the scale factor setting of the layer, and defaults to 4.

The palette for the RpfLayer has different controls that let you tell the layer that you want to limit what chart type to view, change the transparency of the images, and also enable a coverage tool that will show you where your data is located. The coverage tool works in any projection, not just CADRG.

The **RpfFrame** class is the object that reads and decodes the individual RPF image files, and can be used as a stand-alone application to analyze individual image files.

8.5 Vector Product Format (VPF), including VMAP and DCW

The Vector Product Format is another NGA database describing a set of vector (points, lines and polygons) features. The `com.bbn.openmap.layer.vpf.VPFLayer` can be used to view VPF data, which includes the VMAP Level 0 and 1, Digital Chart of the World (DCW) and Digital Nautical Chart (DNC) databases.

All of the different vector objects are organized into groups called features (roads, railroads, inland water, etc). Features are also identified by their type, as point, edge and area features, and are further organized into groups called coverages.

The **VPFLayer** (`com.bbn.openmap.layer.vpf.VPFLayer`) is the layer that displays VPF data. VPFLayers are configured to display features, and in order to help the layer locate the features the properties for the layer have to specify the coverage type, the feature type, and the feature name. For instance, to display roads from a VMAP database, the properties must specify a coverage type for `trans` (transportation), a feature type for lines, and the feature name `roadl`.

The `com.bbn.openmap.layer.vpf.VPFConfig` class can be run on a VPF database top-level directory to create properties for a VPFLayer. The VPFConfig class uses the database description files to present the features in a organized manner, and lets the user pick those features and how they should be displayed, and then prints out the properties to use to configure that layer. These properties should be pasted into the openmap.properties file.

8.6 Digital Terrain Elevation Data (DTED)

DTED data consists of elevations posts spaced out in a two dimensional array. The posts are spaced depending on the resolution of the data:

- Level 0: 30 arc second spacing (~300 meters)
- Level 1: 3 arc second spacing (~100 meters)
- Level 2: 1 arc second spacing (~30 meters)

The DTED data is organized into frame files that space 1 degree by 1 degree areas, with the files named and organized into directories according to their location and resolution.

The **DTEDFrameCache** (com.bbn.openmap.layer.dted.DTEDFrameCache) understands this directory and file structure, and can respond to elevation queries. These queries can be for a specific point, or for an array of elevations covering a specific area. If added to the MapHandler, the DTEDFrameCache can be accessed by any component that is interested in elevation information.

The **DTEDLayer** uses its own DTEDFrameCache to create terrain images from the data. These images can be shaded according to slope, shaded and colored according to elevation, or colored according to elevation. These images are generated on the fly, and the map must be using the CADRG projection in order to be viewed. The **DTEDFrame** class reads and decodes the frame files, and can be used as an application in individual frames files for analysis.

The **DTEDCoverageLayer** is a utility layer that simply draws boxes on the map where the frame files are located, to assist the user in zooming in over the data.

8.7 ETOPO

ETOPO data consist of ASCII data files that also provide elevations at certain intervals, but ETOPO data also contains depth readings under the oceans of the earth. The **ETOPOLayer** (`com.bbn.openmap.layer.etopo.ETOPOLayer`) is an extension of the DTEDLayer that creates a terrain image for the data. The ETOPOLayer doesn't rely on the CADRG projection, however, and can produce a terrain image for any projection.

8.8 ArcInfo Export Files (e00) and MapInfo files (mif)

ArcInfo export files, which can be recognized by the .e00 suffix on their file names, can be displayed using the `com.bbn.openmap.layer.e00.E00Layer`.

MapInfo files, recognized by their .mif suffix) can be displayed with the `com.bbn.openmap.layer.mif.MIFLayer`.

These layers contain all the information needed to render their contents, including colors and styles. The layer simply needs to know the file name, which can be set with a property.

9 Conclusion

This OpenMap document is intended to be a dynamic and changing one, keeping track of the changing capabilities of the OpenMap package as they occur. It will be updated as questions and comments are received. As usual, these can be sent to the development team at openmap@bbn.com.

10 Appendix A: Configuring the Applet

An applet can be launched with the following html file:

```
<HTML>
<HEAD><TITLE>OpenMap Applet</TITLE></HEAD>

<BODY>

<!--"CONVERTED_APPLET"-->
<!-- CONVERTER VERSION 1.0 -->

<OBJECT classid = "clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
      WIDTH = 500
      HEIGHT = 500
      codebase = "http://java.sun.com/products/plugin/1.2/jinstall-12-
      win32.cab#Version=1,2,0,0">

<PARAM NAME = CODE VALUE = com.bbn.openmap.app.OpenMapApplet >
<PARAM NAME = ARCHIVE VALUE = openmap.jar >
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2">

<COMMENT>
<EMBED type="application/x-java-applet;version=1.2"  java_CODE =
com.bbn.openmap.app.OpenMapApplet java_ARCHIVE = openmap.jar WIDTH =
500 HEIGHT = 500
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-
install.html"><NOEMBED>
</COMMENT>

</NOEMBED></EMBED>
</OBJECT>

<!--
<APPLET  CODE = com.bbn.openmap.app.OpenMapApplet  ARCHIVE = openmap.jar
WIDTH = 500 HEIGHT = 500 />
-->

<!--"END_CONVERTED_APPLET"-->
</BODY>
</HTML>
```

The openmap.jar file and openmap.properties file should be located in the same directory as this html file, which is located in the OpenMap package at openmap/share/omapplet.html

11 Appendix B: Examples of using OpenMap Image Creation Applications

12 Appendix C: Configuring the com.bbn.openmap.image.ImageServer

The ImageServer is configured with a Java Properties file. These properties are similar to the properties found in the standard openmap.properties file.

The properties file can contain properties scoped for a particular ImageServer. Scoping is performed using a property prefix, so the properties can be defined as:

prefix.property=value

The ImageServer needs to be told what the prefix is. If a prefix is not provided, that's OK, just the property name will be looked for. The properties for the ImageServer look like this, assuming a prefix `imageserver` is used:

```
# ImageServer layer definitions
imageServer.layers=<layer1 layer2 ...>
layer1.className=<classname>
layer1.prettyName=<pretty name of layer>

# Add other attributes as required by layer1...
layer2.className=<classname>
layer2.prettyName=<pretty name of layer>

# Add other attributes as required by layer2...
# First formatter listed is default.
imageServer.formatters=<formatter1 formatter2 ...>
formatter1.class=<classname of formatter 1>

# Add other formatter1 properties
formatter2.class=<classname of formatter 2>
```

As of OpenMap 4.5, if the ImageServer can't find the 'prefix.layers' or 'layers' property (no prefix used), the 'openmap.layers' property will be used instead. This lets you configure an openmap.properties file for the ImageServer while allowing you to test layer configurations with the OpenMap application.

The ImageServer class also lets you quickly create an image file from an openmap.properties file using the ImageServer properties. At the command line, type:

```
java com.bbn.openmap.image.ImageServer -file output_image_file_name
-properties <path to openmap.properties file>
```

The image will be created in the file you specify.

13 Appendix D: An `openmap.properties` File

```
# **********************************************************************
#
# <copyright>
#
#  BBN Technologies, a Verizon Company
#  10 Moulton Street
#  Cambridge, MA 02138
#  (617) 873-8000
#
#  Copyright (C) BBNT Solutions LLC. All rights reserved.
#
# </copyright>
# **********************************************************************
#
# $Source: /cvs/distapps/openmap/openmap.properties,v $
# $RCSfile: openmap.properties,v $
# $Revision: 1.34 $
# $Date: 2003/09/25 21:02:40 $
# $Author: dietrick $
#
# **********************************************************************
#
# WHAT IS THIS FILE?
#
# This is a generic OpenMap properties file, which controls how
# components are loaded into the OpenMap application.  It specifies
# the initial projection the map should have when OpenMap is started,
# the layers that should be available, which ones should be turned on,
# and lets you adjust all the layers' attributes for their startup
# configuration.  Most importantly, this file also lets you add and
# remove different components from the application itself.  You can
# modify it with any text editor you like.
#
# WHAT ARE PROPERTIES?
#
# Java properties are a set of key=value pairs.  The key is the name
# of the thing you are setting, and the value is what you are setting
# it to.  There are a couple of things to look for that we do with key
# values in this properties file.
#
# First, all the properties that can be set for a component are
# *hopefully* listed in the Javadocs (Java API documentation).  If a
# component is specified as a com.bbn.openmap.PropertyConsumer, it
# will be given a change to configure itself based on the settings
# within the properties file.  Components can be layers or any other
# part of the OpenMap application.
#
# Secondly, the keys are scoped to focus the value to a specific
# instance of a components.  If there are more that one layer of a
# specific type (say, two ShapeLayers, one for roads and one for
# rivers), the names of they keys will have a different prefix for the
# key.  For instance, ShapeLayers have a lineColor attribute you can
# set in this file, and the value used is a hexidecimal value for an
# ARGB color (transparency (A), red (R), green (G), blue (B)):
#
# For a red, non-transparent color for the line.
# lineColor=FFFF0000
#
# But there are two layers - to scope the property for different
# layers, a prefix must be added to the property, separated by a '.':
#
# roads.lineColor=FFFF0000
# rivers.lineColor=FF0000FF
#
# In this case, the roads layer has a red line color and the rivers
# layer has a blue line color.  The prefix 'roads' and 'rivers' is
# something picked to your liking and used only within this property
# file, and is referred to in OpenMap documentation as a marker name.
# Marker names are first used in a list - the openmap.layers property
# is a perfect example:
#
# openmap.layers=first second third
#
# In this example, I've chosen first, second and third to be marker
# names of three different layers.  Later in the properties file, I
# would use these marker names to define the layers and each layer's
# attributes.  For the 'first' layer, I'll define a ShapeLayer:
#
# # All layers require a class name that defines which one to use
# first.class=com.bbn.openmap.layer.shape.ShapeLayer
#
# # And a pretty name to use in the GUI
# first.prettyName=Roads
#
# # Now come properties that are particular to ShapeLayers:
#
# # These first two are mandatory for the ShapeLayer:
# first.shapeFile=<path to shape file (.shp)>
# first.spatialIndex=<path to spatial index file (.ssx)>
#
# # These are optional, and override the defaults set in the ShapeLayer:
# first.lineColor=FFFF0000
# first.lineWidth=2
#
# You do this for each layer listed.  To add a layer to the
# application, you make up a marker name, add it to the list, and then
# define a set of properties for it using the marker name as a prefix.
#

# This marker name list paradigm is used throughout the properties
# file.  It's an important concept to understand before modifying this
# file.
#
# HOW MANY PROPERTIES FILES ARE THERE?
#
# OpenMap looks for this file in several different places.  When it
# finds one, it loads all the properties, and moves on.  If two
# properties have the same key, the last version read wins.  The most
# important place to keep a version of the openmap.properties file is
# a personalized one in your home directory or profile directory.
# This is the last one read.  It also lets you personalize your
# application setup and not affect anyone else using the OpenMap
# installation.
#
# OK, lets define the map!
#
###########################################################
# These properties define the formatters the Simple Http
# Image Server uses. The default formatter is the first one
# in the list.
# The layers property can be used to override the
# openmap.startUpLayers property as the default layers.
###########################################################

formatters=gif jpeg
gif.class =com.bbn.openmap.image.AcmeGifFormatter
jpeg.class=com.bbn.openmap.image.SunJPEGFormatter
#layers=date drawing daynight graticule shapePolitical tz


# ###################################################
# These properties define the starting projection of the map.
# These properties are listed in com.bbn.openmap.Environment.java,
# and affect the initialization of the application.
# ###################################################

# Latitude and longitude in decimal degrees
openmap.Latitude=20f
openmap.Longitude=-20f
# Scale: zoom level (1:scale)
openmap.Scale=3000000000f

# Projection type to start the map with.  Try "cadrg", "orthographic",
# which are projection IDs.  Check a particular projection class for
# it's ID string.
openmap.Projection=mercator

# Width and Height of map, in pixels
openmap.Width=640
openmap.Height=480

# Change this for a different title in the main window.
openmap.Title=OpenMap(tm)

# pixel X/Y window position (if values < 0, then center window on screen)
openmap.x=-1
openmap.y=-1

# The background color of the map, in hex AARRGGBB values (AA is
# transparancy, RR, GG and BB are red, green and blue in hex values
# between 00-FF (0-255)).
openmap.BackgroundColor=FF89C5F9
```

```
# Here is a property that lets you add debug statements to the
# application, to get more informative printouts detailing what is
# going on in the application.  Each source file may Debug statements
# throughout it (Debug.debugging("keyword") or
# Debug.message("keyword", "statement")), and you can turn those
# statements on by adding those keywords to this list.  Follows the
# marker name paradigm, with space-separated names.
#openmap.Debug=basic

# Ironically, the "properties" property, which details where the
# PropertyHandler is looking for its properties, won't work here.

# ###################################################
# These are miscellaneous variables for the OpenMap application.
# ###################################################
# How to lauch a browser to display additional information.
# Windows example: openmap.WebBrowser=C:\\program files\\internet explorer\\iexplore.exe
openmap.WebBrowser=/usr/bin/netscape -install
# Used for creating web pages for the browser
openmap.TempDirectory=/tmp
# Help web pages
openmap.HelpURL=http://openmap.bbn.com/doc/user-guide.html
# Use internal frames as an application - used to be gui.UseInternalFrames
openmap.UseInternalFrames=false

# OpenMap has a Debug messaging mechanism that lets you set certain
# keywords as environment variables (-Ddebug.keyword) to enable
# printouts from different objects.  The code contains these keywords,
# and by looking at a classes' code, you can figure out what the
# keywords are (look for Debug.message("keyword", "message") and 'if
# (Debug.debugging("keyword")' statements).  You can enable these
# messages by putting those keywords in a space-separated list in this
# property.  If you don't want extra messages, you can ignore this
# property and leave it empty.

#openmap.Debug=

# ###################################################
# These properties define the general components to use in the
# application, OTHER than layers.  Notice the marker name list.  You
# can add and removed parts of the application here, simply by
# adjusting this marker name list and adding properties for that
# marker name.  Note the order in which menu objects are important,
# except helpMenu which is always adjusted to be the last menu item.
#
# If you want to remove components from the application, simply remove
# the marker name from the openmap.components list.  You don't have to
# delete the class definition property, too.  If you want to add a
# component to the list, add the marker name to the openmap.components
# list when you want it to be created and added relative to the other
# components, and then add a 'marker name'.class property for that
# component to this file.
# ###################################################
```

```
openmap.components=informationDelegator mouseDelegator projectionstack addlayer dropSupport dndCatcher glc menulist toolBar navpanel zoompanel projectionstacktool scalepanel layersPanel omdtl overviewMapHandler mouseModePanel deleteButton navMouseMode selectMouseMode distanceMouseMode nullMouseMode omdrawingtool omlineloader omcircleloader omrectloader ompointloader omsplineloader omdecsplineloader ompolyloader omdistloader layerHandler

# ###
# Applet components, also alternative component configuration
# with OMControlPanel on the left side of the applet.
# ###
#openmap.components=informationDelegator mouseDelegator projectionstack addlayer glc menulist toolBar omdtl mouseModePanel deleteButton navMouseMode selectMouseMode distanceMouseMode nullMouseMode omdrawingtool omlineloader omcircleloader omrectloader ompointloader omsplineloader omdecsplineloader ompolyloader omdistloader controlpanel layerHandler

### Menu configuration
menulist.class=com.bbn.openmap.gui.menu.MenuList
menulist.menus=fileMenu controlMenu navigateMenu layersMenu gotoMenu helpMenu
fileMenu.class=com.bbn.openmap.gui.FileMenu
controlMenu.class=com.bbn.openmap.gui.ControlMenu
### If you use the OMControlPanel, you can add these...
#controlMenu.items=controlPanelToggle
#controlPanelToggle.class=com.bbn.openmap.gui.menu.ControlPanelToggleMenuItem
###
navigateMenu.class=com.bbn.openmap.gui.NavigateMenu
layersMenu.class=com.bbn.openmap.gui.LayersMenu
gotoMenu.class=com.bbn.openmap.gui.GoToMenu
gotoMenu.addDefaults=true
### Add your own views to the GoToMenu
#gotoMenu.views=Argentina India United_States
#Argentina.latitude=-39.760445
#Argentina.longitude=-65.92294
#Argentina.name=Argentina
#Argentina.projection=Mercator
#Argentina.scale=5.0E7
#India.latitude=20.895763
#India.longitude=80.437485
#India.name=India
#India.projection=Mercator
#India.scale=3.86688E7
#United_States.latitude=38.82259
#United_States.longitude=-96.74999
#United_States.name=United States
#United_States.projection=Mercator
#United_States.scale=5.186114E7
###
helpMenu.class=com.bbn.openmap.gui.DefaultHelpMenu
helpMenu.items=helpUserMenuItem
helpUserMenuItem.class=com.bbn.openmap.gui.UserGuideMenuItems
helpUserMenuItem.class=com.bbn.openmap.gui.menu.WebSiteHelpMenuItem

### Other components
addlayer.class=com.bbn.openmap.gui.LayerAddPanel
deleteButton.class=com.bbn.openmap.gui.OMGraphicDeleteTool
distanceMouseMode.class=com.bbn.openmap.event.DistanceMouseMode
dndCatcher.class=com.bbn.openmap.tools.dnd.DefaultDnDCatcher
```

```
dropSupport.class=com.bbn.openmap.tools.dnd.DropListenerSupport
glc.class=com.bbn.openmap.plugin.graphicLoader.GraphicLoaderConnector
informationDelegator.class=com.bbn.openmap.InformationDelegator
layerHandler.class=com.bbn.openmap.LayerHandler
layersPanel.class=com.bbn.openmap.gui.LayersPanel
menuPanel.class=com.bbn.openmap.gui.MenuPanel
mouseDelegator.class=com.bbn.openmap.MouseDelegator
mouseModePanel.class=com.bbn.openmap.gui.MouseModeButtonPanel
navMouseMode.class=com.bbn.openmap.event.NavMouseMode2
navpanel.class=com.bbn.openmap.gui.NavigatePanel
nullMouseMode.class=com.bbn.openmap.event.NullMouseMode
omcircleloader.class=com.bbn.openmap.tools.drawing.OMCircleLoader
omdecsplineloader.class=com.bbn.openmap.tools.drawing.OMDecoratedSplineLoader
omdistloader.class=com.bbn.openmap.tools.drawing.OMDistanceLoader
omdrawingtool.class=com.bbn.openmap.tools.drawing.OMDrawingTool
omdtl.class=com.bbn.openmap.tools.drawing.OMDrawingToolLauncher
omlineloader.class=com.bbn.openmap.tools.drawing.OMLineLoader
ompointloader.class=com.bbn.openmap.tools.drawing.OMPointLoader
ompolyloader.class=com.bbn.openmap.tools.drawing.OMPolyLoader
omrectloader.class=com.bbn.openmap.tools.drawing.OMRectLoader
omsplineloader.class=com.bbn.openmap.tools.drawing.OMSplineLoader
projectionstack.class=com.bbn.openmap.proj.ProjectionStack
projectionstacktool.class=com.bbn.openmap.gui.ProjectionStackTool
scalepanel.class=com.bbn.openmap.gui.ScaleTextPanel
selectMouseMode.class=com.bbn.openmap.event.SelectMouseMode
toolBar.class=com.bbn.openmap.gui.ToolPanel
zoompanel.class=com.bbn.openmap.gui.ZoomPanel

controlpanel.class=com.bbn.openmap.gui.OMControlPanel
controlpanel.overviewLayers=overviewLayer
controlpanel.isTool=false
controlpanel.overviewScaleFactor=30f
controlpanel.overviewMinScale=50000000f
controlpanel.overviewStatusLayer=com.bbn.openmap.layer.OverviewMapAreaLayer
controlpanel.lineColor=FFFF0000
controlpanel.fillColor=33FF0000
controlpanel.controls=com.bbn.openmap.gui.LayerControlButtonPanel
controlpanel.controls.orientation=horizontal
controlpanel.controls.configuration=NORTH

# ###
# BeanPanel properties
# ###
#beanpanel.class=com.bbn.openmap.tools.beanbox.BeanPanel
#beanpanel.beans.path=path to directory containing jar files
#beanpanel.tabs=tab1 tab2 tab3
#beanpanel.tab1.name=Generic
#beanpanel.tab1.beans=com.bbn.openmap.examples.beanbox.SimpleBeanObject
#beanpanel.tab2.name=Container
#beanpanel.tab2.beans=com.bbn.openmap.examples.beanbox.SimpleBeanContainer
#beanpanel.tab3.name=Military
#beanpanel.tab3.beans=com.bbn.openmap.examples.beanbox.Fighter

# ###################################################
# Properties defined for the overview map handler.
# ###################################################
```

```
overviewMapHandler.class=com.bbn.openmap.gui.OverviewMapHandler
# marker name list, layer defined later
overviewMapHandler.overviewLayers=overviewLayer
# how zoomed out to keep the overview map versus the main map
overviewMapHandler.overviewScaleFactor=10f
# when to stop zooming in
overviewMapHandler.overviewMinScale=10000000f
# the layer to use to render on top, showing where the main map covers.
overviewMapHandler.overviewStatusLayer=com.bbn.openmap.layer.OverviewMapAreaLayer

# ###################################################
# You can refer to other properties files and have their properties
# loaded as well.  This is good for defining a set of layers that work
# with a particular type of data, for instance, and then override some
# of those properties defined in those files to localize them for your
# setup.  This works with a marker name list.  By default, nothing defined.
# ###################################################

#openmap.include=include1 include2
#include1.URL=<http://whatever>
#include2.URL=<http://whatever>

# ###################################################
# Here is the list of layers to add to the map.  The properties for
# each marker name are defined later.
#
# If you want to remove a layer from the application, remove its
# marker name from the openmap.layers property list.  You do not have
# to delete all of its properties as well.
#
# If you want to add a layer to the application, add it's marker name
# to the openmap.layers property list, and then add its properties to
# this file.  As a minimum, Layers all need a 'marker name'.class
# property, and a 'marker name'.prettyName property (for the GUI
# components).  Consult the JavaDocs for a layer to see what other
# properties can be set for that layer.
# ###################################################

# Layers listed here appear on the Map in the order of their names.
openmap.layers=date dtlayer distlayer quake daynight cities test graticule demo shapePolitical
politicalCombo

# If you get more data, you can add other layers defined in this file.
# You'll want to look at the properties for each layer and modify them
# as needed.  Check the javadocs for definitions of the available
# properties for each layer.

#openmap.layers=date quake daynight test graticule terrain demo dtedcov jdted jrpf shapePolitical
ScaledPolitical ScaledFillPolitical

# These layers are turned on when the map is first started.  Order
# does not matter here...
openmap.startUpLayers=graticule drawing shapePolitical

# In OpenMap 4.4, the LayerAddPanel was added.  This panel allows
# certain layers/plugins to be added to the application dynamically.
# Only certain layers/plugin have been updated to be able to work with
# the Inspector to set their inital parameters. To add a layer to this
# list, create a unique marker name for a generic instance of the
# layer, and then supply the <layername>.class field and
# <layername>.prettyName (see below) for that object.  The class name
# will be the type of layer/plugin created by the LayerAddPanel, and
# this prettyName will be the generic description of the layer
# presented to the user.  The user will have an opportunity to name
# the specific layer that gets created.
openmap.addableLayers=earth shape grat utmgrid rpf shispi eipi wmsp epi

utmgrid.class=com.bbn.openmap.plugin.UTMGridPlugIn
utmgrid.prettyName=UTM Grid Overlay

epi.class=com.bbn.openmap.plugin.esri.EsriPlugIn
epi.prettyName=ESRI Layer

grat.class=com.bbn.openmap.layer.GraticuleLayer
grat.prettyName=Graticule Layer

earth.class=com.bbn.openmap.layer.EarthquakeLayer
earth.prettyName=Earthquake Layer

shape.class=com.bbn.openmap.layer.shape.ShapeLayer
shape.prettyName=Shape Layer

rpf.class=com.bbn.openmap.layer.rpf.RpfLayer
rpf.prettyName=RPF Layer

shispi.class=com.bbn.openmap.plugin.shis.SHISPlugIn
shispi.prettyName=Simple Http Image Server (SHIS) Plugin

eipi.class=com.bbn.openmap.plugin.earthImage.EarthImagePlugIn
eipi.prettyName=Earth Image Plugin

wmsp.class=com.bbn.openmap.plugin.wms.WMSPlugIn
wmsp.prettyName=WMS Layer

# ###################################################
# These are the properties for individual layers.  Consult the
# javadocs (Java API pages) for the individual layers for options.
# ###################################################

### Drawing version of the EditorLayer.  An EditorLayer, when active,
### places tools in the ToolPanel that creates/manipulates the
### OMGraphics on this particular layer.
dtlayer.class=com.bbn.openmap.layer.editor.EditorLayer
dtlayer.prettyName=Drawing Layer
dtlayer.editor=com.bbn.openmap.layer.editor.DrawingEditorTool
#dtlayer.showAttributes=false
dtlayer.loaders=lines polys rects circles points
dtlayer.mouseModes=Gestures
dtlayer.distance.class=com.bbn.openmap.tools.drawing.OMDistanceLoader
dtlayer.lines.class=com.bbn.openmap.tools.drawing.OMLineLoader
dtlayer.polys.class=com.bbn.openmap.tools.drawing.OMPolyLoader
dtlayer.rects.class=com.bbn.openmap.tools.drawing.OMRectLoader
dtlayer.circles.class=com.bbn.openmap.tools.drawing.OMCircleLoader
dtlayer.points.class=com.bbn.openmap.tools.drawing.OMPointLoader

### Another Drawing version of the EditorLayer, with the editor set up
### to only create OMDistance objects.
distlayer.class=com.bbn.openmap.layer.editor.EditorLayer
distlayer.prettyName=Distance Layer
distlayer.showAttributes=false
distlayer.editor=com.bbn.openmap.layer.editor.DrawingEditorTool
distlayer.loaders=distance
distlayer.distance.class=com.bbn.openmap.tools.drawing.OMDistanceLoader
distlayer.distance.attributesClass=com.bbn.openmap.omGraphics.DrawingAttributes
distlayer.distance.lineColor=FFAA0000
distlayer.distance.mattingColor=66333333
distlayer.distance.matted=true

### Layer to catch Drag and Drop events from the DefaultDnDCatcher and
### DropListenerSupport
simpleBeanLayer.class=com.bbn.openmap.examples.beanbox.SimpleBeanLayer
simpleBeanLayer.prettyName=Simple Bean Layer

### Layer used by the overview handler
overviewLayer.class=com.bbn.openmap.layer.shape.ShapeLayer
overviewLayer.prettyName=Overview
overviewLayer.shapeFile=data/shape/dcwpo-browse.shp
overviewLayer.spatialIndex=data/shape/dcwpo-browse.ssx
overviewLayer.lineColor=ff000000
overviewLayer.fillColor=ffbdde83

###
# Demo layer - the layer's palette has a bunch of buttons to call
# the Drawing Tool.
demo.class=com.bbn.openmap.layer.DemoLayer
demo.prettyName=Demo

###
# DrawingToolLayer - no palette, just a generic layer to catch
# graphics from the OMDrawingToolLauncher, and to call the OMDrawingTool
# to edit graphics already part of the layer.
drawing.class=com.bbn.openmap.layer.DrawingToolLayer
drawing.prettyName=Drawing Tool Layer
drawing.addToBeanContext=true

### ShapeFile layers
shapePolitical.class=com.bbn.openmap.layer.shape.ShapeLayer
shapePolitical.prettyName=Political Boundaries
# Specify shapefile and spatial-index file as a filename or pathname.
# If the former, you must reference the directory where this file
# lives in your CLASSPATH
shapePolitical.shapeFile=data/shape/dcwpo-browse.shp
shapePolitical.spatialIndex=data/shape/dcwpo-browse.ssx
# Colors (32bit ARGB)
shapePolitical.lineColor=ff000000
shapePolitical.fillColor=ffbdde83
```

```
# ScaleFilterLayer switches between the layers at the transisition scales.
ScaledPolitical.class=com.bbn.openmap.layer.ScaleFilterLayer
ScaledPolitical.prettyName=Political Boundaries
# List 2 or more layers
ScaledPolitical.layers=ThinShapePolitical ShapePolitical
# List the transition scales to switch between layers
ScaledPolitical.transitionScales=5000000

# ScaleFilterLayer switches between the layers at the transisition scales.
ScaledFillPolitical.class=com.bbn.openmap.layer.ScaleFilterLayer
ScaledFillPolitical.prettyName=Political Areas
# List 2 or more layers
ScaledFillPolitical.layers=ThinShapeFillPolitical ShapeFillPolitical
# List the transition scales to switch between layers
ScaledFillPolitical.transitionScales=5000000

### ShapeFile layer, full resolution converted vmap political boundaries
ShapePolitical.class=com.bbn.openmap.layer.shape.ShapeLayer
ShapePolitical.prettyName=Political Edge
ShapePolitical.shapeFile=data/shape/vmap_edge.shp
ShapePolitical.spatialIndex=data/shape/vmap_edge.ssx

### ShapeFile layer, thinned converted vmap political boundaries
ThinShapePolitical.class=com.bbn.openmap.layer.shape.ShapeLayer
ThinShapePolitical.prettyName=Political Edge
ThinShapePolitical.shapeFile=data/shape/vmap_edge_thin.shp
ThinShapePolitical.spatialIndex=data/shape/vmap_edge_thin.ssx

### ShapeFile layer, full resolution converted vmap political areas
ShapeFillPolitical.class=com.bbn.openmap.layer.shape.ShapeLayer
ShapeFillPolitical.prettyName=Political Solid
ShapeFillPolitical.shapeFile=data/shape/vmap_area.shp
ShapeFillPolitical.spatialIndex=data/shape/vmap_area.ssx
ShapeFillPolitical.lineColor=BDDE83
ShapeFillPolitical.fillColor=BDDE83

### ShapeFile layer, thinned converted vmap political areas
ThinShapeFillPolitical.class=com.bbn.openmap.layer.shape.ShapeLayer
ThinShapeFillPolitical.prettyName=Political Solid
ThinShapeFillPolitical.shapeFile=data/shape/vmap_area_thin.shp
ThinShapeFillPolitical.spatialIndex=data/shape/vmap_area_thin.ssx
ThinShapeFillPolitical.lineColor=BDDE83
ThinShapeFillPolitical.fillColor=BDDE83

### ShapeFile layer, thinned converted vmap political boundaries,
# muiltiple shape files on one layer...
politicalCombo.class=com.bbn.openmap.layer.shape.MultiShapeLayer
politicalCombo.prettyName=Political Boundaries 2
politicalCombo.shapeFileList=pol_edges pol_fill

politicalCombo.pol_edges.shapeFile=data/shape/vmap_edge_thin.shp
politicalCombo.pol_edges.prettyName=Borders
politicalCombo.pol_edges.buffered=false

politicalCombo.pol_fill.shapeFile=data/shape/vmap_area_thin.shp
politicalCombo.pol_fill.prettyName=Areas
politicalCombo.pol_fill.lineColor=FFAAAA66
politicalCombo.pol_fill.fillColor=FFAAAA66
politicalCombo.pol_fill.buffered=false

### Graticule layer
graticule.class=com.bbn.openmap.layer.GraticuleLayer
graticule.prettyName=Graticule
# Show lat / lon spacing labels
graticule.showRuler=true
graticule.show1And5Lines=true
# Controls when the five degree lines and one degree lines kick in
#- when there is less than the threshold of ten degree lat or lon
#lines, five degree lines are drawn.  The same relationship is there
#for one to five degree lines.
graticule.threshold=2
# the color of 10 degree spaing lines (ARGB)
graticule.10DegreeColor=99000000
# the color of 5 degree spaing lines (ARGB)
graticule.5DegreeColor=99009900
# the color of 1 degree spaing lines (ARGB)
graticule.1DegreeColor=99003300
# the color of the equator (ARGB)
graticule.equatorColor=99FF0000
# the color of the international dateline (ARGB)
graticule.dateLineColor=99000000
# the color of the special lines (ARGB)
graticule.specialLineColor=99000000
# the color of the labels (ARGB)
graticule.textColor=99000000


### Date & Time layer
date.class=com.bbn.openmap.layer.DateLayer
date.prettyName=Date & Time
# display font as a Java font string
date.font=SansSerif-Bold
# like XWindows geometry: [+-]X[+-]Y, `+' indicates relative to
# left edge or top edges, `-' indicates relative to right or bottom
# edges, XX is x coordinate, YY is y coordinate
date.geometry=+20+30
# background rectangle color ARGB
date.color.bg=ff808080
# foreground text color ARGB
date.color.fg=ff000000
# date format (using java.text.SimpleDateFormat patterns)
date.date.format=EEE, d MMM yyyy HH:mm:ss z


### Day/Night shading layer properties
daynight.class=com.bbn.openmap.layer.daynight.DayNightLayer
daynight.prettyName=Day/Night Shading
# draw terminator as poly (faster calculation than image, defaults to
# true).
daynight.doPolyTerminator=true
# number of vertices for polygon terminator line.  this is only valid
# if doPolyTerminator is true...
```

```
daynight.terminatorVerts=512
# termFade - the distance of the transition of fade, as a percentage of PI.
daynight.termFade=.1
# currentTime - true to display the shading at the computer's current time.
daynight.currentTime=true
# updateInterval - time in milliseconds between updates.  currentTime has to be
# true for this to be used.  1000*60*5 = 300000 = 5min updates
daynight.updateInterval=300000
# Shading Colors (32bit ARGB)
daynight.nighttimeColor=64000000
daynight.daytimeColor=00FFFFFF


### Earthquake layer
quake.class=com.bbn.openmap.layer.EarthquakeLayer
quake.prettyName=Recent Earthquakes


### Test layer
test.prettyName=Test
test.class=com.bbn.openmap.layer.test.TestLayer
test.line.visible=true
test.circ.visible=true
test.rect.visible=true
test.text.visible=true
test.poly.visible=true
#test.poly.vertices=80 -180 80 -90 80 0 80 90 80 180 70 180 70 90 70 0 70 -90 70 -180

### VMAP Political layer - Run the com.bbn.openmap.layer.vpf.VPFConfig
### class to create properties for a set of features.
vmapPolitical.class=com.bbn.openmap.layer.vpf.VPFLayer
vmapPolitical.prettyName=VMAP Political
vmapPolitical.vpfPath=data/vmap/vmaplv0
vmapPolitical.coverageType=bnd
vmapPolitical.featureTypes=edge area text
#  just display coastlines and political boundaries
#vmapPolitical.edge= polbndl coastl depthl
vmapPolitical.edge= polbndl coastl
#  just display political areas and not oceans
#vmapPolitical.area= oceansea polbnda
vmapPolitical.area= polbnda

# Use this property for a better focus on feature types, especially
# for more fine-grained databases
#vmapPolitical.searchByFeature=true

### VMAP Coastline layer
vmapCoast.class=com.bbn.openmap.layer.vpf.VPFLayer
vmapCoast.prettyName=VMAP Coastline Layer
vmapCoast.vpfPath=data/vmap/vmaplv0
## a predefined layer from the VPF predefined layer set found in
## com/bbn/openmap/layer/vpf/defaultVPFLayers.properties
vmapCoast.defaultLayer=vmapCoastline

# Basic political boundaries with DCW
dcwPolitical.class=com.bbn.openmap.leyer.vpf.VPFLayer
dcwPolitical.prettyName=DCW Political Boundaries
dcwPolitical.vpfPath=path to DCW data
dcwPolitical.coverageType=po
dcwPolitical.featureTypes=edge area


### Java RPF properties
jrpf.class=com.bbn.openmap.layer.rpf.RpfLayer
jrpf.prettyName=CADRG
# This property should reflect the paths to the RPF directories
jrpf.paths=data/RPF
# Number between 0-255: 0 is transparent, 255 is opaque
jrpf.opaque=255
# Number of colors to use on the maps - 16, 32, 216
jrpf.number.colors=216
# Display maps on startup
jrpf.showmaps=true
# Display attribute information on startup
jrpf.showinfo=false
# Scale images to fit the map scale
jrpf.scaleImages=true
jrpf.coverage=true

### Another Java RPF Layer - usually keep CADRG and CIB separate,
# although you don't have to.
jcib.class=com.bbn.openmap.layer.rpf.RpfLayer
jcib.prettyName=CIB
# This property should reflect the paths to the RPF directories
jcib.paths=data/CIB/RPF
# Number between 0-255: 0 is transparent, 255 is opaque
jcib.opaque=255
# Number of colors to use on the maps - 16, 32, 216
jcib.number.colors=216
# Display maps on startup
jcib.showmaps=true
# Display attribute information on startup
jcib.showinfo=false
# Scale images to fit the map scale
jcib.scaleImages=true


### Java DTED Coverage properties
dtedcov.class=com.bbn.openmap.layer.dted.DTEDCoverageLayer
dtedcov.prettyName=DTED Coverage
# This property should reflect the paths to the DTED level 0 and 1
# directories.  These levels can be combined.
dtedcov.paths=data/dted
# DTED Level 2 data!
dtedcov.level2.paths=data/dted2
# Number between 0-255: 0 is transparent, 255 is opaque
dtedcov.opaque=255

# *NOTE* This property needs to be changed to specify a DTED coverage file,
# or the location where you want one created if the layer doesn't find
# it here.  If you add or remove coverage from your dted collection,
# you should delete this file so that an accurate one will be created.
```

```
dtedcov.coverageFile=<path to coverage file>

# This is an optional property. You can substitute a URL for the
# coverage file instead.  This is checked first before the coverage
# file, and if a valid file is found at the URL, then the coverage
# file is ignored.  The layer does not try to create a file at this
# URL!
#dtedcov.coverageURL=http://dstl.bbn.com/openmap/data/dted/coverage.dat

### Java DTED properties
jdted.class=com.bbn.openmap.layer.dted.DTEDLayer
jdted.prettyName=DTED
# This property should reflect the paths to the DTED level 0 and 1
# directories.  These levels can be combined.
jdted.paths=data/dted
# DTED Level 2 data!
jdted.level2.paths=data/dted2
# Number between 0-255: 0 is transparent, 255 is opaque
jdted.opaque=255
# Number of colors to use on the maps - 16, 32, 216
jdted.number.colors=216
# Level of DTED data to use on startup (0, 1, 2)
jdted.level=0
# Type of display for the data on startup
# 0 = no shading at all
# 1 = greyscale slope shading
# 2 = band shading, in meters
# 3 = band shading, in feet
# 4 = subframe testing
# 5 = elevation, colored
jdted.view.type=5
# Contrast setting on startup, 1-5
jdted.contrast=3
# height (meters or feet) between color changes in band shading on startup
jdted.band.height=25
# Minumum scale to display images. Larger numbers mean smaller scale,
# and are more zoomed out.
jdted.min.scale=20000000

###
# Terrain layer.
terrain.class=com.bbn.openmap.layer.terrain.TerrainLayer
terrain.prettyName=Terrain Tools
# This property should reflect the paths to the DTED directories
terrain.dted.paths=data/dted
# The default tool to use for the terrain layer.  Can be PROFILE or LOS.
terrain.default.mode=PROFILE

###
# PlugIn layer with a samplePlugIn marker name - comments are commented out!
#samplePlugIn.class=com.bbn.openmap.plugin.PlugInLayer
#samplePlugIn.prettyName=Earth At Night
#samplePlugIn.plugin=<path to PlugIn class>
# Then, plugin properties as needed, with layer marker name.plugin
# prefix.  These depend on the PlugIn, and you should consult the
# JavaDocs for that PlugIn class for specifics.
```

```
#samplePlugIn.plugin.property1=value
#samplePlugIn.plugin.property2=value

###
# LocationLayer that holds cities.  The palette for this layer lets
# you turn on the names and declutter matrix, if you want.  The
# declutter matrix can get expensive at small scales.
cities.class=com.bbn.openmap.layer.location.LocationLayer
cities.prettyName=World Cities
cities.locationHandlers=csvcities
cities.useDeclutter=false
cities.declutterMatrix=com.bbn.openmap.layer.DeclutterMatrix

csvcities.class=com.bbn.openmap.layer.location.csv.CSVLocationHandler
csvcities.prettyName=World Cities
csvcities.locationFile=data/cities.csv
csvcities.csvFileHasHeader=true
csvcities.locationColor=FF0000
csvcities.nameColor=008C54
csvcities.showNames=false
csvcities.showLocations=true
csvcities.nameIndex=0
csvcities.latIndex=5
csvcities.lonIndex=4
csvcities.csvFileHasHeader=true

#####################################
# CORBA Layers
#####################################

vpfspec.prettyName=Political Solid
vpfspec.class=com.bbn.openmap.layer.specialist.CSpecLayer
vpfspec.name=OPENMAP/VPF/VMAPLV0
#vpfspec.ior=file:///home/dietrick/htdocs/ior/vpf.ior
vpfspec.staticArgs=bnd polbnd


corbaScaledPolitical.class=com.bbn.openmap.layer.ScaleFilterLayer
corbaScaledPolitical.prettyName=Political Boundaries
# List 2 or more layers
corbaScaledPolitical.layers=corbaThinShapePolitical corbaShapePolitical
# List the transition scales to switch between layers
corbaScaledPolitical.transitionScales=10000000

corbaScaledFillPolitical.class=com.bbn.openmap.layer.ScaleFilterLayer
corbaScaledFillPolitical.prettyName=Political Areas
# List 2 or more layers
corbaScaledFillPolitical.layers=corbaThinShapeFillPolitical corbaShapeFillPolitical
# List the transition scales to switch between layers
corbaScaledFillPolitical.transitionScales=10000000

### ShapeFile layer
corbaShapePolitical.class=com.bbn.openmap.layer.specialist.CSpecLayer
corbaShapePolitical.prettyName=Political Edge
#corbaShapePolitical.ior=file:///home/dietrick/htdocs/ior/shapeedge.ior
corbaShapePolitical.name=OPENMAP/SHAPE/VMAPEDGE
```

```
### ShapeFile layer
corbaThinShapePolitical.class=com.bbn.openmap.layer.specialist.CSpecLayer
corbaThinShapePolitical.prettyName=Political Edge
#corbaThinShapePolitical.ior=file:///home/dietrick/htdocs/ior/shapethinedge.ior
corbaThinShapePolitical.name=OPENMAP/SHAPE/THINEDGE

### ShapeFile layer
corbaShapeFillPolitical.class=com.bbn.openmap.layer.specialist.CSpecLayer
corbaShapeFillPolitical.prettyName=Political Solid
#corbaShapeFillPolitical.ior=file:///home/dietrick/htdocs/ior/shapearea.ior
corbaShapeFillPolitical.name=OPENMAP/SHAPE/VMAPAREA

### ShapeFile layer
corbaThinShapeFillPolitical.class=com.bbn.openmap.layer.specialist.CSpecLayer
corbaThinShapeFillPolitical.prettyName=Political Solid
#corbaThinShapeFillPolitical.ior=file:///home/dietrick/htdocs/ior/shapethinarea.ior
corbaThinShapeFillPolitical.name=OPENMAP/SHAPE/THINAREA

#Java RPF properties
corbarpf.class=com.bbn.openmap.layer.rpf.corba.CorbaRpfLayer
corbarpf.prettyName=CADRG Server
# Number between 0-255: 0 is transparent, 255 is opaque
corbarpf.opaque=255
# Number of colors to use on the maps - 16, 32, 216
corbarpf.numberColors=216
# Display maps on startup
corbarpf.showMaps=true
# Display attribute information on startup
#corbarpf.colormodel=indexed
corbarpf.showInfo=false
corbarpf.scaleImages=true
corbarpf.killCache=false
corbarpf.chartSeries=ANY
corbarpf.jpegQuality=.6
#corbarpf.ior=http://blatz.bbn.com/users/dietrick/ior/crpf.ior
corbarpf.subframeCacheSize=50
corbarpf.auxSubframeCacheSize=10
corbarpf.colormodel=indexed
corbarpf.name=OPENMAP/RPF/CRFPSERVER
corbarpf.coverage=false
# Number between 0-255: 0 is transparent, 255 is opaque
corbarpf.coverageOpaque=150
# Fill the rectangles or not
corbarpf.CoverageFill=true
corbarpf.ONC.showcov=false
corbarpf.JNC.showcov=false
corbarpf.GNC.showcov=false

# End CORBA Layers
#############################################

# LinkLayer properties
#############################################
link.class=com.bbn.openmap.layer.link.LinkLayer
link.prettyName=Link Test
link.host=localhost
link.port=20525
link.propertiesURL=http://somehost/properties/linkshape.properties

# E00 Layer
##############################################
e00.class=com.bbn.openmap.layer.e00.E00Layer
e00.prettyName=E00 Political Layer
e00.FileName=/data/e00/germany/pppoly.e00

# Buffered Layer
##############################################
buf.class=com.bbn.openmap.layer.BufferedLayer
buf.prettyName=Buffered Layer
buf.layers=graticule dtedcov
buf.visibleLayers=graticule
```