Software Design of the GPSTool Package

(Christof Dallermassl (cdaller@iicm.edu))
Revision : 1.3

October 2, 2003

Abstract

This document describes the rough design of the modules in the org.dinopolis.gpstool package and the modules of the GPSMap application.

Table of Contents

1 Architectural Design

This section describes the modules contained in the org.dinopolis.gpstool package and explains the structure of the GPSMap application.

1.1 GPS Data Sources

One of the major modules in the org.dinopolis.gpstool package is the one that reads and interprets data from a gps device. This module is named org.dinopolis.gpstool.gpsinput.

The module was designed to be independent of the format of the data and of the source of the data. An example for different formats of the data could be NMEA or the proprietary Garmin protocol, the source could either be the serial port, a file or a network server that provides any clients with gps data (like gpsd[1] does).

So to be able to get gps information (like position, altitude, speed, etc.), the source in the form of a GPSDevice has to be chosen and a way to interpret the data coming from the device, in the form of a GPSDataProcessor.

These two classes are connected and from this moment on, gps information can be obtained. This information is delivered in the form of events, anyone can register for. The listener can register for all gps events or just for a specific one. In table 1 are the events and its value types listed.

Event Type	Value of Event
Location	GPSPosition
Heading	Float
Speed	Float
Number of Satellites	Integer
Altitude (in meters)	Float
Satellite Info	SatelliteInfo
Depth	Float
Estimated Pos Error (EPE)	GPSPositionError

Table 1: Events fired from the GPSDataProcessor

A short code snipped shows how to read NMEA data from a serial device:

```
// create processor for NMEA data:
  GPSDataProcessor gps_data_processor = new GPSNmeaDataProcessor();

// create gps device for serial port:
  Hashtable environment = new Hashtable();
  environment.put(GPSSerialDevice.PORT_NAME_KEY,"/dev/ttyS1");
  environment.put(GPSSerialDevice.PORT_SPEED_KEY,new Integer(4800));
  GPSDevice gps_device = new GPSSerialDevice();
  gps_device.init(environment);

// connect processor with device and open it:
  gps_data_processor.setGPSDevice(gps_device);
  gps_data_processor.open();


// create property change listener for gps events:
PropertyChangeListener listener = new ProperyChangeListener()
{
```

[1] http://freshmeat.net/projects/gpsd/

```
    public void propertyChange(PropertyChangeEvent event)
    {
      Object value = event.getNewValue();
      String name = event.getPropertyName();
      if(name.equals(GPSDataProcessor.LOCATION))
      {
        System.out.println("The new location is"
         +(GPSPosition)value.getLatitude() + "/"
         +(GPSPosition)value.getLongitude());
      }
    }
};

// register as listener for location events:
  gps_data_processor.addGPSDataChangeListener(GPSDataProcessor.LOCATION, listener);
```

A little example that demonstrates the features of this module is the java application org.dinopolis.gpstool.gpsinput.GPSTool. It shows how to read from a file or from the serial interface and how to register for gps events. As a matter of fact, this application was the beginning of the whole module.

1.2 GPSMap Application

GPSMap is the main application of the org.dinopolis.gpstool package. It is a moving map application that is able to show the current position on maps that may be downloaded from the internet, a track of the positions in the past, location markers for points of interest, etc.

GPSMap uses some parts of the open source openmap[2] framework. Although the openmap framework provides a lot of functionality, some was not reused but re-implemented to keep the dependencies to the library low.

Nevertheless, GPSMap uses openmap's MapBean class as its central component. A MapBean consists of layers that hold geographic information to be drawn for a specific area and scale.

The main class of the GPSMap application is org.dinopolis.gpstool.GPSMap.

1.2.1 Resources

GPSMap reads some command line parameters, but most if the configuration is read from a properties file (`GPSMap.properties`). This file must be in the classpath of the application and is read via the org.dinopolis.util.Resources class. Any changes of the configuration are saved into a file into the directory `.gpsmap` under the user' home directory. Not all resources can be edited via the "Preferences" dialog, so if you are missing some screws to turn, try the file itself.

The resources also hold the information for the resource editor (title, description, type).

1.2.2 User Interface

The user interface is widely configured in the resource files. The structure of the menu is completely defined in the resource file and the actions that are executed by selecting a menu entry are named in the resource file as well.

Localization can be done by creating a localized version of the resource file.

[2] `http://openmap.bbn.com`

1.2.3 Projection

This data is projected from the geoid coordinates (latitude, longitude) to screen coordinates. As the projections provided by openmap did not work for the maps of mapblast[3] or expedia[4], a new projection was developed. The maths was taken from the gpsdrive[5] project of Fritz Ganter.

This projection provides the calculation from latitude/longitude to screen (forward methods) and from screen coordinates to latitude/longitude (inverse methods).

The class that implements the projection is org.dinopolis.gpstool.projection.FlatProjection.

For a full understanding of this class it is necessary to read the documentation of the projections of the openmap framework.

1.2.4 Layers

GPSMap organizes its data in layers that are administered by a com.bbn.openmap.MapBean. Whenever the projection changes (scale or center is changed), the map bean informs all layers about this change (projectionChanged method). The layers have to recalculate (project) their data from latitude/longitude to the screen coordinates and paint them. As the calculation may take its time, this is usually done in a different task by a SwingWorker. As soon as the calculation is done, the data is painted on the screen (paintComponent method).

The usage of background tasks also explains the behavior of GPSMap, that after panning the map, other elements (in other layers) are drawn slightly later at their correct position.

If one wants to add geographic information (e.g. position of friends/cars, etc.) the best solution is to add a new layer that implements the projectionChanged and the paintComponent methods. That's all! Using the projection passed in the projectionChanged method, the conversion of geographical to screen coordinates is easy. Lengthy calculations should use a SwingWorker, so the user interface is not blocked.

In the following, some detailed information about different layers is given.

1.2.5 Map Layer

The map layer is probably the most important layer at the moment. It displays raster maps that were previously downloaded form expedia or mapblast and stored locally on the hard disk (directory `<home>/.gpsmap/maps`). The informations about the files is kept in the file `<home>/.gpsmap/maps.txt` (name of file, latitude/longitude of center of map, scale of map (in mapblast style), height/width of image). In this file, relative and absolute paths are accepted for maps.

One principle of the map painting algorithm is that if no maps for a given scale are available, maps of other scales are used as well and resized to fit the used scale (see figure 1 for an example).

The first attempt to draw the maps was the following: Find all maps that are visible and draw them in the order largest scale to smallest scale. So if there

[3] `http://www.mapblast.com`
[4] `http://www.expedia.com`
[5] `http://www.gpsdrive.de`

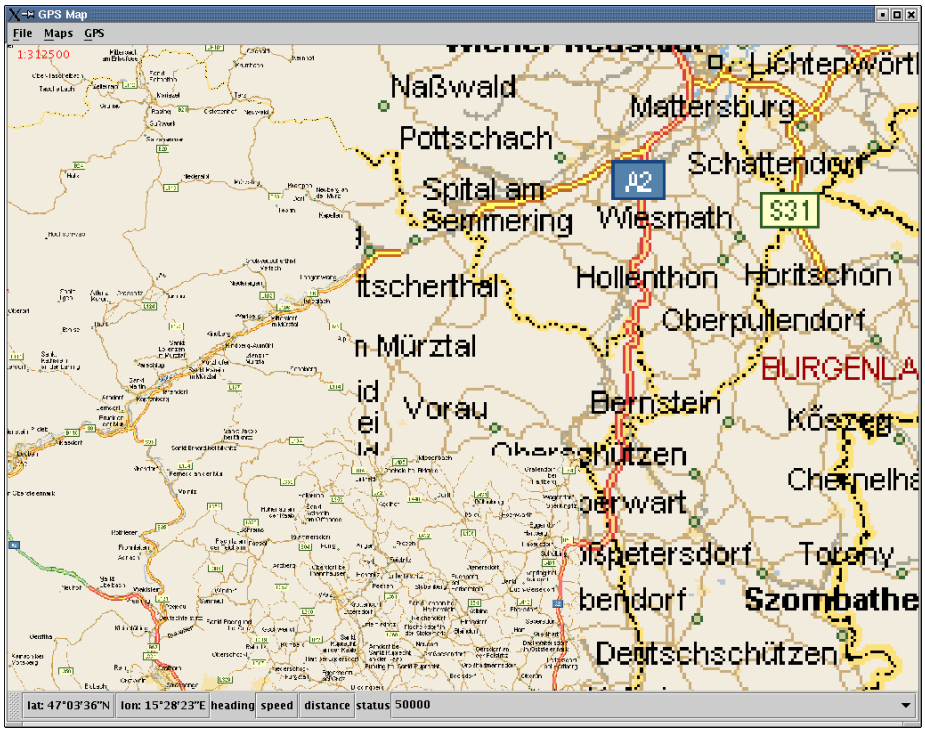

Figure 1: Maps of different scales may be displayed.

is a plan of the city Graz and a map of Europe, the city plan is painted over the map of Europe.

This algorithm scales very badly, as all maps are painted, even if the user does not see the maps because of another map lying over the first one.

So an algorithm was developed that searches the smallest map to show, paint it, and find the rectangles on the screen that are not covered by this map. For the remaining empty rectangles, the algorithm is repeated until the screen is filled, or no more maps are available. This algorithm is implemented and documented in the org.dinopolis.gpstool.gui.util.VisibleImage class.

Maps are only painted, if their scale is not completely different to the scale that is currently being used. This prevents the painting of the city plan, when the user wants to see western Europe, as the city plan would be so small anyway. So if the current scale is 1:200000, only maps up to (e.g.!) 1:100000 are used, other (more detailed) maps, are not even considered to be painted! This factor is configurable.

1.2.6 Location Marker Layer

The layer that displays location markers handles different sources[6] of markers. They may be read from a file or from a relation database and provide LocationMarker objects for a given area (limited by north, south, west, east

[6]interface org.dinopolis.gpstool.gui.layer.location.LocationMarkerSource

latitude/longitude).

Additionally the sources can be asked to apply a given filter, so only location markers for one or more given categories should be retrieved. This Filter was designed to be independent of the source, so in the case of a relation database source it is translated into the correct SQL statements.

1.3 Debug

The org.dinopolis packages use the org.dinopolis.util.Debug package for printing debug messages. This package is similar to the log4j package of the apache framework. It allows to define debug messages that are only printed if the attached debug level is activated.

The debug levels may be activated by using the appropriate API or by editing the debug properties file. For a detailed description of the Debug class, please see the design document of the debug utility.

2 Plugins

The GPSMap application supports different kinds of plugins. These plugins extend the functionality of the application. As there is a clearly and quite simple interface for plugins, the developement is quite simple and possible without the need to read and understand the source of the rest of the application. Another advantage of plugins is, that removing then makes the application smaller and probably faster. So big, fat computers may install and run all possible plugins, whereas the old notebook that is used in the car for location tracking just uses the plugins that are really needed.

In the following, a short explanation of the different aspects of plugin developement is given.

2.1 Types of Plugins

GPSMap supports different kinds of plugins for different purposes. Some are very special for a very small task (like saving the content of the map component to a file (screenshot)), others are very general and influence the functionaltiy in a wider way (like adding a layer to the map component, some entries in the menu, and may react on mouse clicks and keys from the user). In general, all plugin interfaces (and helper interfaces) are in the package org.dinopolis.gpstool.plugin.

- The ReadTrackPlugin is used to load tracks from files.
- The WriteImagePlugin is used to save a "screenshot" from the map component to a file.
- The MouseModePlugin: GPSMap supports different mouse modes that react on mouse activities from the user (click, drag, etc.). The user may switch a mouse mode on or off (and there is always only one mouse mode active). An example for a mouse mode is navigation (zoom in out, pan the map, etc.). As the mouse modes are switched on or off by the user, they have to provide some information for an entry in the menu or a toolbar: name, icon, shortcut key, ...

Additonally, a mouse mode may provide a layer that allows the mouse mode to draw something on the map component.

- The GuiPlugin is the most powerful plugin. It has the possibility to add one or more entries in the menu of the application. An example would be a window that provides information about memory usage or a tachometer that shows lots of informations from the gps device (speed, average, direction, etc.).

  Additionally, GuiPlugins may provide one or more mouse modes to be able to react on mouse clicks on the map component.

  If a GuiPlugin needs to draw anything on the map component, it may provide a Layer. GPSMap uses the MapBean class of the openmap library as the central map component. This class uses Layer objects as layers. Every layer is informed about changes of the projection (zoom in/out, move) in the projectionChanged(com.bbn.openmap.proj.Projection) method. The layer may paint some map details in the paint(java.awt.Graphics) or paintComponent(java.awt.Graphics) methods. Please, see section 1.2.3 for details about projections.

  The paint() method should return as fast as possible, so complicated calculations should not be done in this method but in a background thread. The class org.dinopolis.gpstool.gui.BasicLayer provides a framework that calls a calculation method in a background thread (SwingWorker). So the developer does not need to worry about this!

- The MapRetrievalPlugin: Used by the download mouse mode to retrieve raster maps from various sources. The application requests a raster map for a given location, size and scale. As different map sources may provide only specific scales, the MapRetrievalPlugin has a method that has to return the really used scale (may differ from the scale requested). All scale values are given in mapblast units (as this was the first server supported). E.g. 1000 is a very small scale (lots of details), whereas 1000000 (one million) shows most of Europe.

The interface Plugin is the base interface for all plugins that provide general information (like the name and version of a plugin). This information may be used by a plugin manager or by a plugin downloader to find the latest version of a plugin.

All plugins are instatiated and then initialized with a PluginSupport object that provides interfaces for all important modules and componentes of GPSMap.

Please see the javadoc documentation of the plugin interfaces and classes for further details.

2.2 Loading of Plugins

In this section the two mechanisms used to find and load plugins are explained:

- Find the implementation classes of a given interface or base class.
- Load classes from jar files that are not in the classpath.

The first problem when using plugins is to find one or more implementations of a special java interface or base class. The normal classloader does not provide this functionality, so the help of an external configuration is needed. Sun uses a file in the `META-INF/services` directory (in a jar file or elsewhere in the classpath) for this purpose. This file is named like the interface or base class and contains the names of classes that implement this interface or base class.

As an example, suppose there are two implementations of the interface org.dinopolis.gpstool.plugin.GuiPlugin, namely the classes foo.Bar and bar.Foo. So the file `META-INF/services/org.dinopolis.gpstool.plugin.GuiPlugin` contains two lines:

```
foo.Bar
bar.Foo
```

Sun uses an internal class (in the package sun.misc) to find the implementations (called *services*). As the sun class is internal and may change in the future, its functionality was newly implemented. org.dinopolis.util.servicediscovery.ServiceDiscovery retrieves the information in `META-INF/services` from one or more classloaders and returns the names or faster the instances of the classes of the given interface of base class.

The ServiceDiscovery uses the getResources(String) of the ClassLoader to find the files in the `META-INF/services` directory.

This leads to the second problem of plugins: it cannot be guaranteed that the classpath includes the plugin (jar). A good plugin architecture allows to find, load and use the jar files that contain the plugins in one or more directories. For this purpose, a special ClassLoader was written that loads classes from jar files located in one or more directories that may be given at runtime.

This class loader is org.dinopolis.util.servicediscovery.RepositoryClassLoader and in combination with the ServiceDiscovery described above, it provides the wanted functionality: Find and load implementations of a given class and use jar files in one or more directories.

end of document