Geomajas SLD Editor plug-in

Geomajas Developers and Geosparc

Geomajas SLD Editor plug-in

by Geomajas Developers and Geosparc

1.0.0-M1

Table of Contents

List of Figures

Chapter 1. Introduction

The Geomajas SLD Project is a stand-alone project under the Geomajas banner. It's goal is to provide a code for handling and manipulating SLD data and files. The initial version of the SLD editor provides a minimal editor and in-memory backend service that supports the following use cases:

- Open a list of SLD files and allow the user to select an SLD by name
- Show general information about the SLD: title and name
- Show the rules of the single FeatureTypeStyle of the SLD: name and title of the rule
- Support reordering of the rules
- Support deletion and addition of a rule
- Show the single symbolizer and (optional) filter of a rule
- Support management of fills, strokes and image point symbols
- Support management of a subset of filters (single binary operator combined with an optional NOT-operator)

Chapter 2. Design

1. MVP principle

The editor is designed around the Model-View-Presenter principle and makes use of the GWT Platform framework for its implementation. The general concepts can be found on the GWTP website but we will explain some of the design choices that we made to apply these concepts to the domain of SLD editing. Generally speaking, the problem of editing SLD's can be reduced to the general problem of editing a graph of entities, each entity representing a specific part of the SLD structure. The problem is further simplified by the fact that the SLD structure is a tree-like structure and all entity associations are therefore of the parent-child type.

The hierarchical nature of the SLD lends itself very nicely to a hierarchical tree of nested presenters. Although GWTP is fairly page-oriented, it is quite easy to map the concept of a nested presenter (which refers to a presenter whose layout is nested inside another presenter's layout) to the parent-child nature of SLD in the case of a desktop-like application such as the SLD editor.

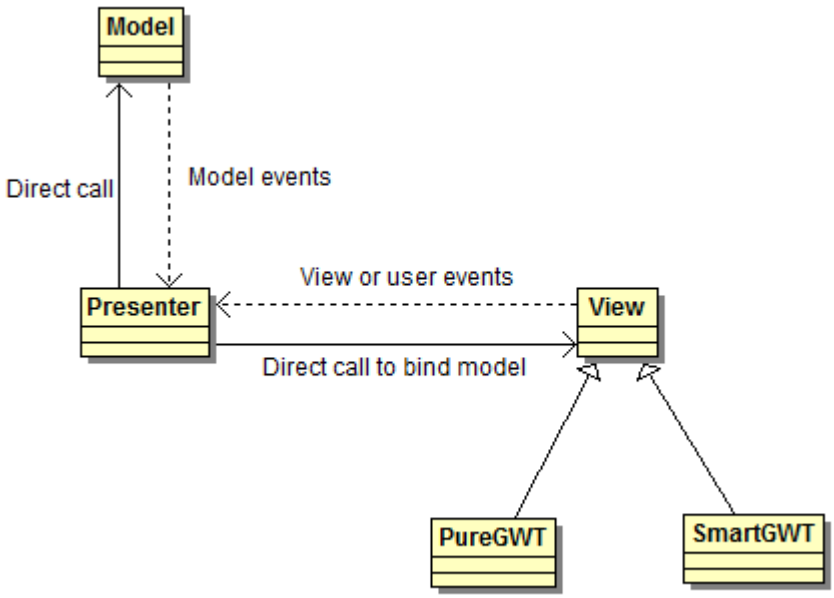
The question of how to deal with the model-presenter separation is also of some importance. In our view, the model should be unaware of the presenters and model-to-presenter coupling should be done through events in all cases. The presenter can of course hold a reference to the model and call its methods directly. The same holds for the view, which should be passive and unaware while the presenter holds a direct reference to it. We slightly divert from the common practice of a passive view in two ways:

1. When the user performs an action, this is usually forwarded to the presenter by directly exposing UI elements such as buttons in the View interface. We have chosen to convert such user actions to intermediate events which are independent of the underlying view technology and forward these events to the presenter through the event bus instead. This saves us from having to create a technology independent layer of abstract buttons, textfields and so on on top of the GWT/SmartGWT UI stacks.

2. Although the view is passive, it has some limited knowledge of the model in the sense that it knows how to bind certain model properties to UI elements and vice-versa. Most of our view therefore implement a `ModelToView(SubModel model)` method that takes a particular model subset (SubModel) as an argument.

The following figure describes the general communication flow in our MVP architecture:

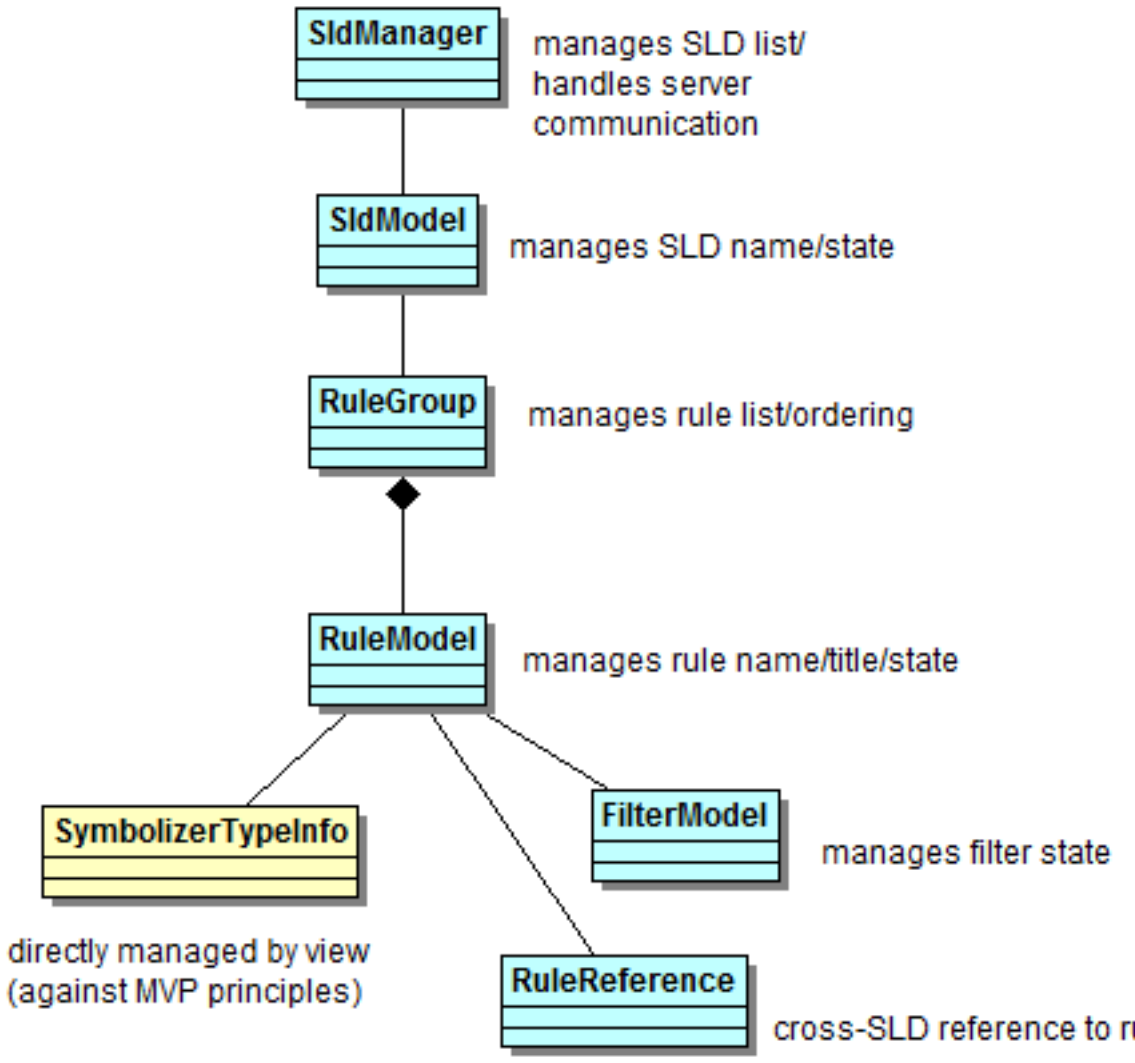
Figure 2.1. MVP architecture



2. Model

The model part of the architecture is responsible for communicating with the server and holding an editable representation of the SLDs. We have followed an approach of strictly separating interface and implementation by creating hierarchy of new interfaces to expose certain parts of the SLD tree to their respective views and presenters. The following picture shows the different model interfaces and their respective relationships:

Figure 2.2. Model part of the architecture

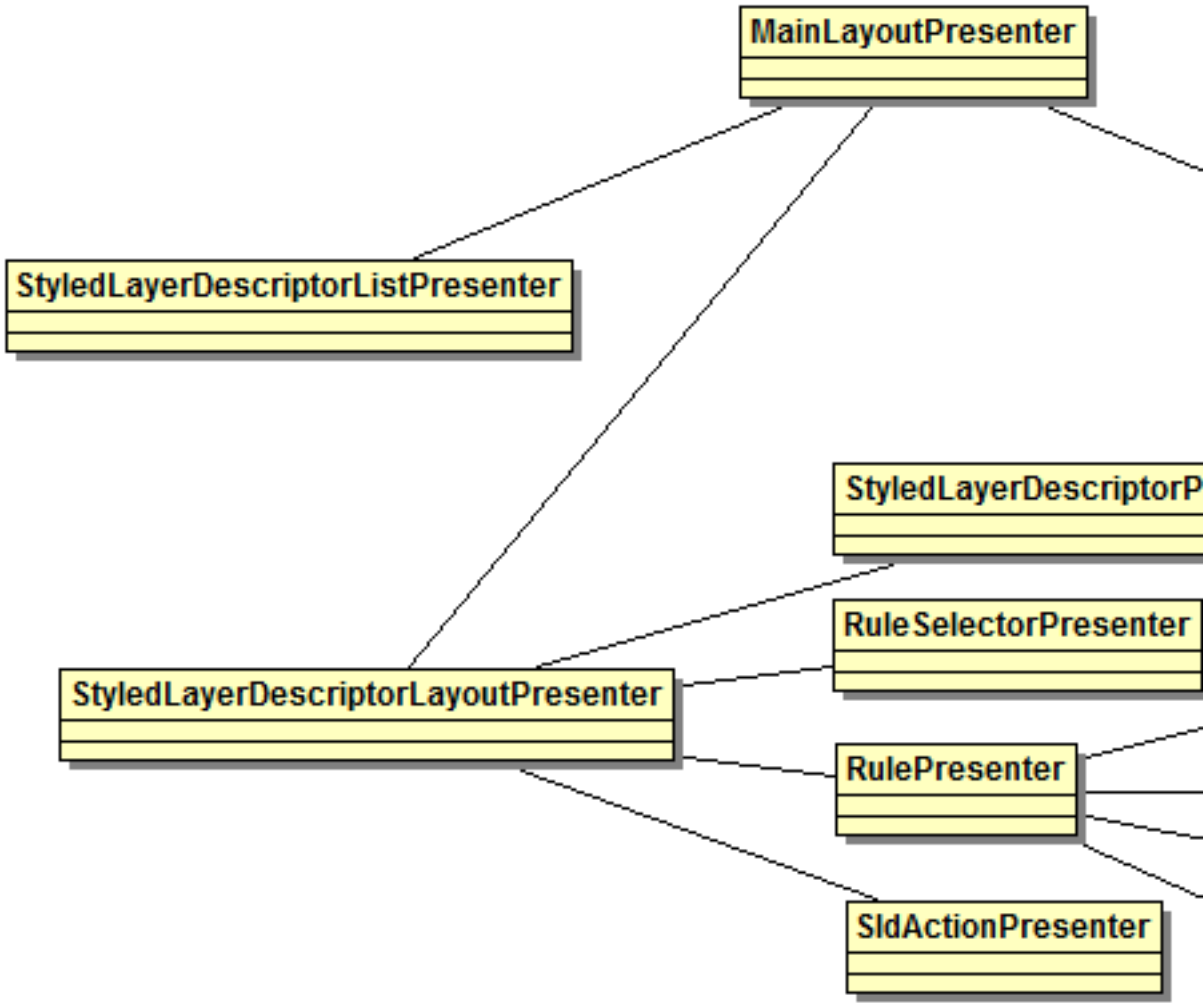


The interfaces (in blue) reflect the parts of the SLD that are currently manageable by the editor. The model is responsible for managing the SLD in terms of CRUD operations. The changes to the SLD model are applied locally and will only be forwarded to the server on request of the user. The model classes are responsible for maintaining the integrity if the model in terms of validation but know how to deal with an incomplete model where necessary. The eventual SLD configuration object is available only after synchronizing the complete model state (see `synchronize()` method at the various levels).

3. View/Presenter

The presenter and view hierarchy is responsible for visualizing the model and handling the user interaction with the model. Our presenter hierarchy closely mimics the model hierarchy and/or the view layout. Where there is a natural hierarchy between presenters, the nested presenter concept has been used to reflect this in the layout as well. In other cases, a separate layout presenter was needed to nest some presenters that have no logical common parent. The following figure shows the nesting of the various presenters:

Figure 2.3. Presenter part of the architecture



Chapter 3. Development guidelines

This chapter holds some general principles for extending the current code base. Most of it can be found in the GWTP documentation, some of it is added by ourselves.

1. Project layout

The project consists of 5 subprojects:

- geomajas-project-sld-editor-common-gwt: the common part of the code, containing the model, the presenters and the view interfaces
- geomajas-project-sld-editor-puregwt: the plain GWT implementation of the view classes
- geomajas-project-sld-editor-gwt: the SmartGWT implementation of the view classes
- geomajas-project-sld-editor-documentation: the documentation
- geomajas-project-sld-editor-webapp: the web application (war project)

2. How to add model classes

Changing the model should follow the following principles:

- start by adding the interfaces and/or events that you need in the common GWT model packages: `org.geomajas.sld.editor.common.client.model` and `org.geomajas.sld.editor.common.client.model.event`
- events should follow our practice of defining the event in a single class (copy/paste + rename an existing event), declare the model as HasHandlers for your event and fire the event through the event bus:

```
public class MyModelImpl implements MyModel, HasXxxHandlers {
 private final EventBus eventBus;

 @Inject
 public MyModelImpl(final EventBus eventBus){
  this.eventBus = eventBus;
 }

 public void fireEvent(GwtEvent<?> event) {
  eventBus.fireEvent(event);
 }

 public HandlerRegistration addXxxHandler(XxxHandler handler) {
  return eventBus.addHandler(XxxEvent.getType(), handler);
 }

}
```

- make the implementations of your interfaces
- bind your implementations to your interfaces in the GIN module class: `org.geomajas.sld.editor.common.client.gin.ClientModule`

3. How to bind/autowire model classes

As all model/presenter and view classes are created by GIN, they can be autowired to each other by declaring them in the constructor and using the `@inject` annotation. This is nice for singletons, but in some case you want to create multiple instances of a class and still use injection. The most flexible way to do this is to create a factory interface with a `create()` method. This factory should then be registered in the GIN module class `org.geomajas.sld.editor.common.client.gin.ClientModule` as follows:

```
install(new GinFactoryModuleBuilder().implement(MyModel.class, MyModelImpl.clas
```

The nice thing about this way of working is that you can add extra constructor arguments to your model constructor that are passed to the factory method at runtime instead of being injected. This is simply done by adding an `@assisted` annotation to these arguments (see `RuleModelImpl` for an example).

4. How to add presenter/view classes

A presenter/view pair should be created for each part of the model that can be managed as a single unit from a single UI class. There are two types of presenters in the current code base:

- singleton presenters: extensions of the `com.gwtplatform.mvp.client.Presenter` class that are initialised by the common GWTP lifecycle process. Normally, these presenters are connected with a specific place (think 'page') and revealed when the appropriate URL is called. For the current application we have chosen to force revelation of these presenters by tying them to an event of an ancestor presenter. This is typically done by implementing the handler of the event and annotating the handler method with a `@proxyEvent` annotation:

  ```
  @ProxyEvent
  public void onInitSldLayout(InitSldLayoutEvent event) {
   forceReveal();
  }
  ```

  In the above case, the InitSldLayout event is fired by the `StyleLayerDescriptorLayoutPresenter`, which is a parent presenter that is mainly responsible for drawing part of the layout. At the moment we have two such presenters, `MainLayoutPresenter` being the other one. Singleton presenters should also determine in which part of the layout their view will appear. This is done by firing a `RevealContentEvent` event in the `revealInParent()` method:

  ```
  @Override
  protected void revealInParent() {
   RevealContentEvent.fire(this, MainLayoutPresenter.TYPE_MAIN_CONTENT, this);
  }
  ```

  The second argument determines the view slot in which the view's widget should appear. See the implementation of the MainlayoutPresenter in case you need to implement your own layout structure.

- For separating logic and view with a simple graphical component whose lifecycle should be completely within your control, extending PresenterWidget is the answer. In this case the presenter can simply be injected in its parent presenter. Revealing the presenter can be done whenever you choose to do so. This is a useful strategy for dealing with popup windows, see `org.geomajas.sld.editor.common.client.presenter.CreateSldDialogPresenterWidget` for an example.

The view interfaces are defined in the presenter classes and should allow the presenter to pass the part of the model that is being edited to the view whenever this is required. We have typically added

a `modelToView()` method to the view interface to achieve this. Internally, view implementations should bind the model to their internal form elements, typically using the getter/setter pairs of the submodel. It could be stated that this way of working violates the classical MVP principle of having a passive view, but this is no problem as we only move the boiler plate code to modify the model via its accessors to the view.

The view implementations should extend `com.gwtplatform.mvp.client.ViewImpl` and return the widget class in their asWidget() method. At present all view classes are SmartGwt implementations. User events should be forwarded to the presenters by creating a custom event in the `org.geomajas.sld.editor.common.client.presenter.event` package. This avoids having to expose SmartGWT specific DOM events to the presenter. In most cases the view can suffice with a generic event that states that the SLD has been changed in some way. The `SldContentChanged` event was created exactly for this purpose. To fire events and register event handlers, the same practice as for models can be used.

5. Internationalization and utility classes

Classical GWT internationalization techniques are used to localize messages throughout the application. Because of GIN, GWT.create() statements can be avoided altogether, ensuring normal junit testability. The SldEditorMessages class can be injected into each class that needs localization.

Utility classes should also be defined as singletons, using a clear interface for separation. An example is `ViewUtil`, which abstracts some basic view functionality such as showing an alert message.