Cayenne Data View Design and Applications

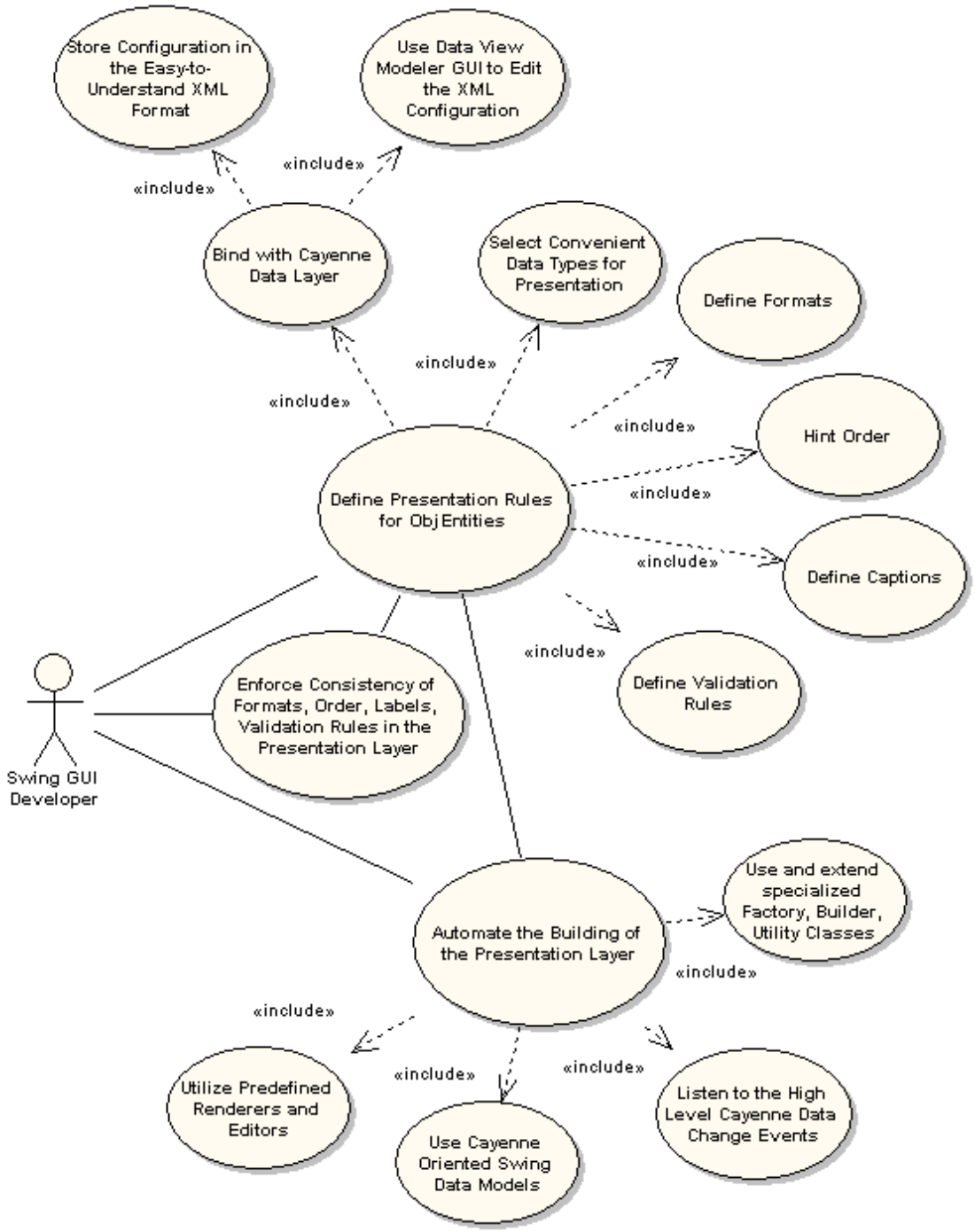
Andriy Shapochka, ObjectStyle Group

Table of Contents

Rationale

The Cayenne framework provides means to develop persistent layers of domain objects and wraps the JDBC specific logic so that the solid part of manipulations with the data can be performed uniformly and in terms of pure domain Java objects. While this greatly simplifies the design and the implementation of business tiers and the persistense related operations there usually remains a time consuming task. It comprises the building of the presentation layer and binding it with the business and data objects based on Cayenne. One who has the experience of the Swing GUI development knows how much time it takes to work out all the minutest details of formatting, in-place input validation, handling the interactive data modification, enforcing naming, order, and formatting consistensy, etc. Whenever the domain or requirements change, for example, new attributes are added to an ObjEntity, formats or captions for some kind of data are modified, or relationships acquire a meaning different from the original one, the developer is faced with the necessity to go through the number of Swing data models, panels, various helpers fishing out the bits of code to be corrected. Another important feature is easiness of the prototyping of data aware GUIs. Those who, at one time or another, worked with Borland VCL for C++ or Delphi or DataExpress/dbSwing for Java and similar frameworks could recall how painless it was to create a rough prototype of the GUI working with the relational database and bind it to the actual data. That was possible due to the layer of easily configured data aware classes and components. And once the working prototype had been ready its refinement went smoothly.

Having more than a yearlong experience in the development and the maintainance of the Swing GUIs built on top of the Cayenne framework with one of the major requirements the GUIs must be quickly and seamlessly integrated with the database schemas that have similar structure in the main and still may greatly differ in what extra data they store and how these data should be rendered and modified interactively, the author decided to try and come up with an approach to deal with the problems enumerated above. One can imagine the idea of Cayenne Data Views as a bridge between the domain defined in terms of Cayenne DataObjects and a presentation layer built with Swing. In fact, they may be used in the Web environments as well but Swing interfaces is the main direction of the ongoing effort. Conceptually, Data Views are close to the application facades described, for instance, by Martin Fowler. The following figure outlines the responsibilities and the applicability area of the "Cayenne Data View" subproject.



The validation rules mentioned in the Use Cases are meant to be "lightweght", i.e. it is generally agreed the validation related code to enforce the business rules should be located in the domain area but, still, there are various checks one could prefer to perform right in the presentation layer like inclusion in a predefined range of values, correct input format, sometimes, even preliminary credit card number validity check with the well known Luhn algorithm, the other kinds of sanity checks. Presently the validation is on the list of the features to be added. See the "To Do List" at the end of the document.

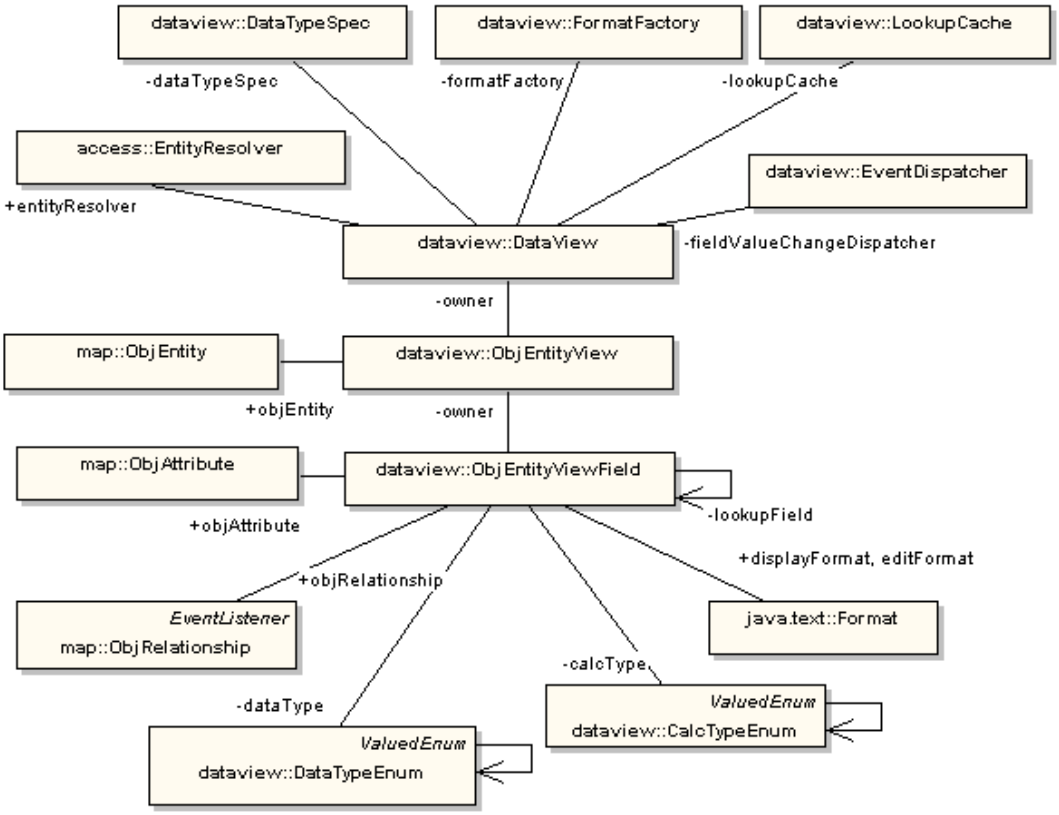
Perhaps, it should be emphasized this project is not an attempt to create a plug to any barrel with the ready made opinions and cures for every kind of blind alley sometimes you happen to find yourself in when working on the next Cayenne based Swing GUI. On the contrary, all its features and the overall

structure grows and is dictated by the everyday problems and solutions arising in the GUI development. You can use as much of Data View functionality as you like and omit the rest and you can adjust or reconfigure data types, formats, etc. to your liking due to the highly modular structure of the library. In the nearest future the author plans to add quite realistic examples of the library usage, they all will take their origin from the real-world applications. The project is work in early progress and thus immature in many points, however it already found its uses in the commercial system the snapshots from which you can see in chapter "Data Views in Action".

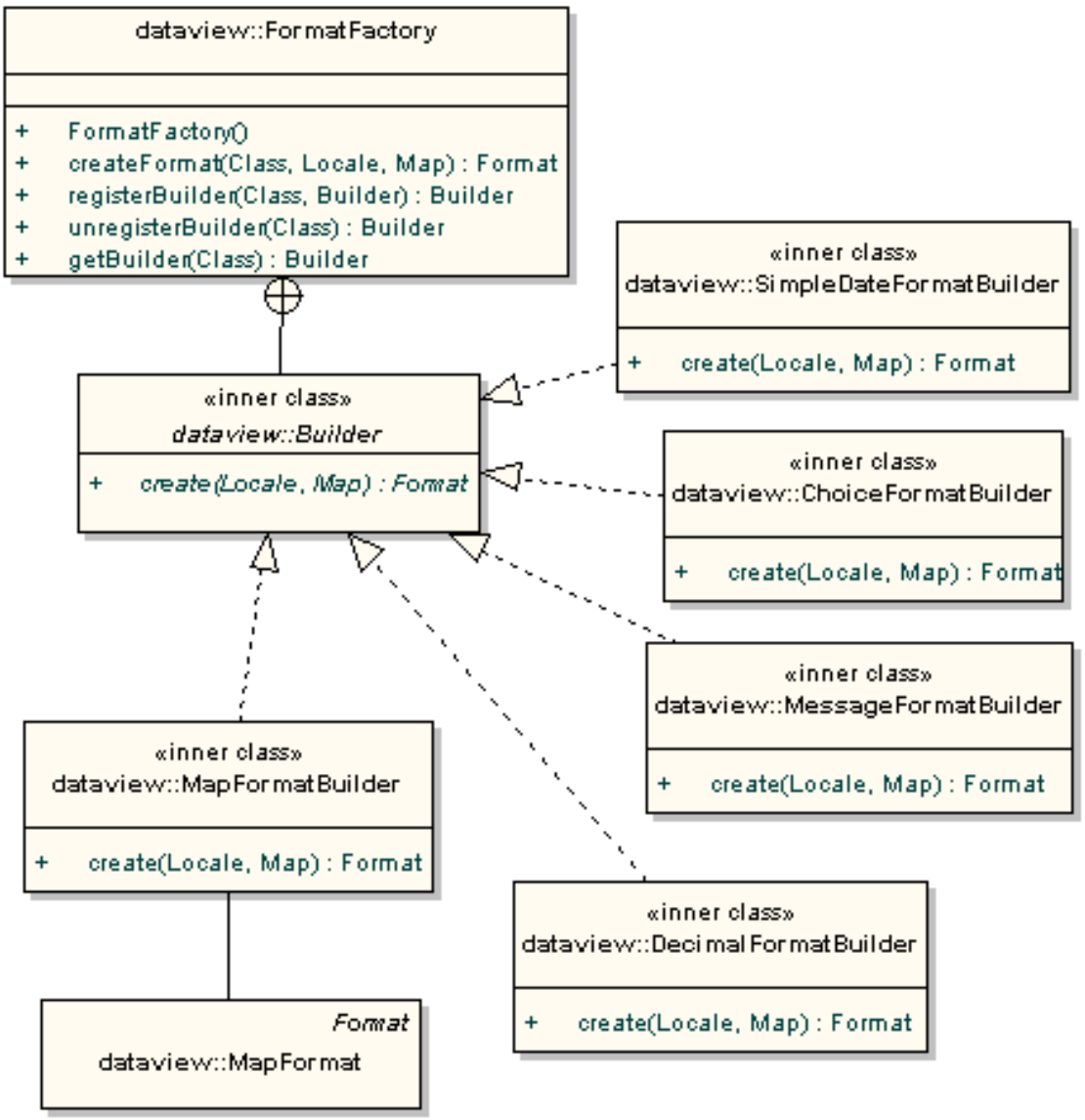
Data View Concepts

The class diagram on the figure below captures the structure of Data Views and how they rely on the classes in the Cayenne subpackages. As you can see class DataView is the root of the Data View hierarchy. It serves as a container for ObjEntityViews. While the Data View is represented by a single object of type DataView it can incorporate the ObjEntityViews from several Data View configuration files. This approach is different from the one taken in the case of Cayenne Data Maps where DataMaps are created per data map file. Thus, as opposite to the case of Data Maps ObjEntityViews can freely refer to each other whether they are defined in the same configuration file or different ones. The recommended practice is to store closely related ObjEntityViews in the same XML "module" and define several "modules" based on the criterion of such closeness. When you load the list of XML files into the DataView all the (lookup) relationships are resolved automatically and the ObjEntityViews share the same namespace (so give them different names even if they are located in different XML files). The diagram also shows that DataView is associated with several classes such as org.objectstyle.cayenne.access.EntityResolver, org.objectstyle.cayenne.dataview.DataTypeSpec, org.objectstyle.cayenne.dataview.FormatFactory. The instances of these classes participate in the process of loading actual Data View files. The main unit of any Data View is ObjEntityView. It always refers to an ObjEntity defined in one of the used data maps and defines various presentation rules for this entity. There can be several ObjEntityViews for an ObjEntity, each of them utilized by an application when appropriate. EntityResolver finds the corresponding ObjEntities by the names as the Data View is being loaded. Every ObjEntityView must have a name unique in the DataView context. ObjEntityViews contain fields called ObjEntityViewFields. They must be named uniquely within an ObjEntityView. The fields can be of two sorts. The regular "nocalc" fields reference ObjAttributes in the ObjEntity referred by the parent ObjEntityView. A field describes how the related ObjAttribute will be rendered and edited, the data type and the formats to use in the presentation layer. It also defines the caption that is regularly used to name a table column or label the corresponding input control on the form. It also configures the editability and the visibility of this attribute. You can setup an order in which the fields appear in a JTable (as columns) or on a form with the preferred index. You can define the default values for the fields if desired. The FormatFactory described below is used to create actual Format classes used by the fields. There may be several fields for an ObjAttribute in the ObjEntityView. The field's data type deserves paying special heed. Cayenne defines the mapping between several Java Class types and JDBC types. It is convenient to use and sufficient in many cases but there are all sorts of situations when one would like to define more specific data types to make use of, especially for the presentation purposes. In many cases when Integer is used as a flag you would like to map it to Boolean. Or you may have a Money type or Date type that holds only year, month, and day values, and so forth. The ObjEntityViewFields give you this opportunity. They provide the easily extensible system of data types often used in the business applications. This system takes care of converting values back and forth between Cayenne data types and application specific data types. It relies on two classes DataTypeEnum and DataTypeSpec, and both of them can be extended to define new types of any sorts. While the entire thing may seem redundant at the first glance, actually, it is a powerful concept that can save you a lot of time usually spent on the manual conversions. Back to the kinds of fields. The other sort of ObjEntityViewFields is "lookup" fields. They point to the fields defined in other ObjEntityViews so the actual values to display, edit, or select from come from those referenced fields. Such a lookup field corresponds to an ObjRelationship with the ObjEntity referred by this its ObjEntityView as a source and the ObjEntity referred by the lookup ObjEntityView as a target instead of an ObjAttribute. Next the field identifies which lookup ObjEntityView and which particular field it wants to use as a lookup. These dependencies are resolved when the data views are loaded into memory. Class LookupCache helps maintain and map to data objects lists of values used in lookup combo boxes and lists. Thus you can describe

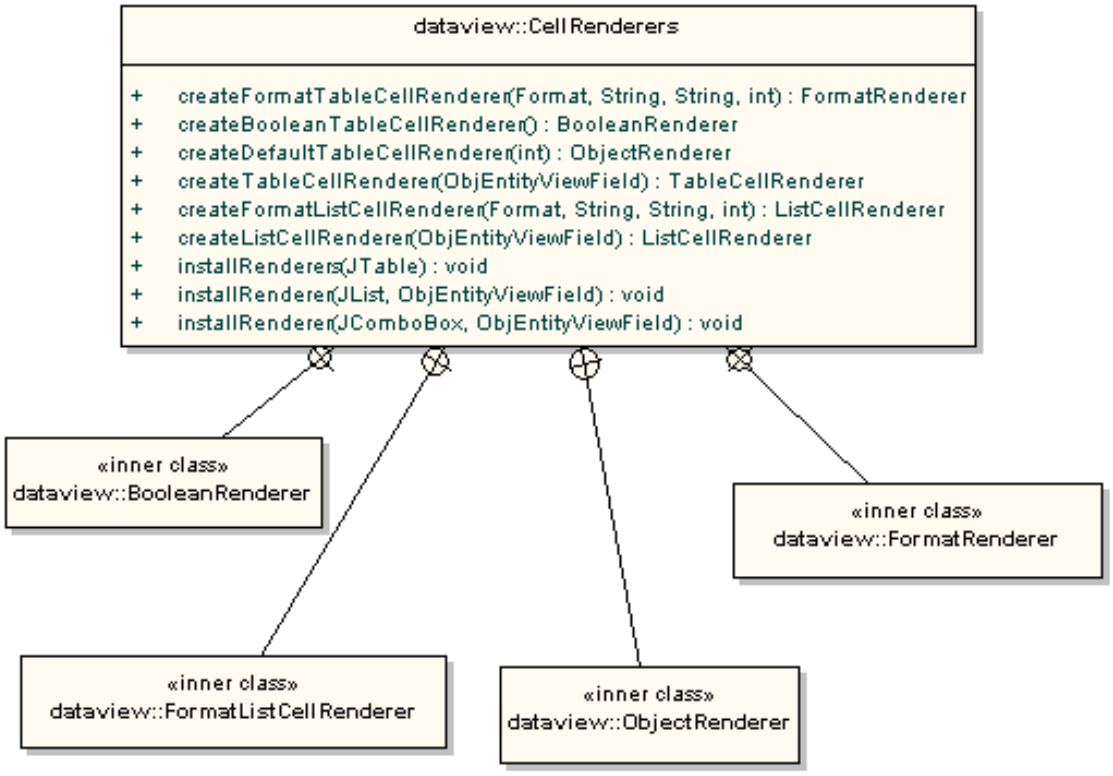
the rules of visual editing the relationships or make aggregated views base on the related entities. One more useful feature is a centralized EventDispatcher within the DataView class. It is used by the fields to fire the events whenever their values are changed. Thus the ObjAttribute/ObjRelationship modifications can be easily propagated to all the listening components.



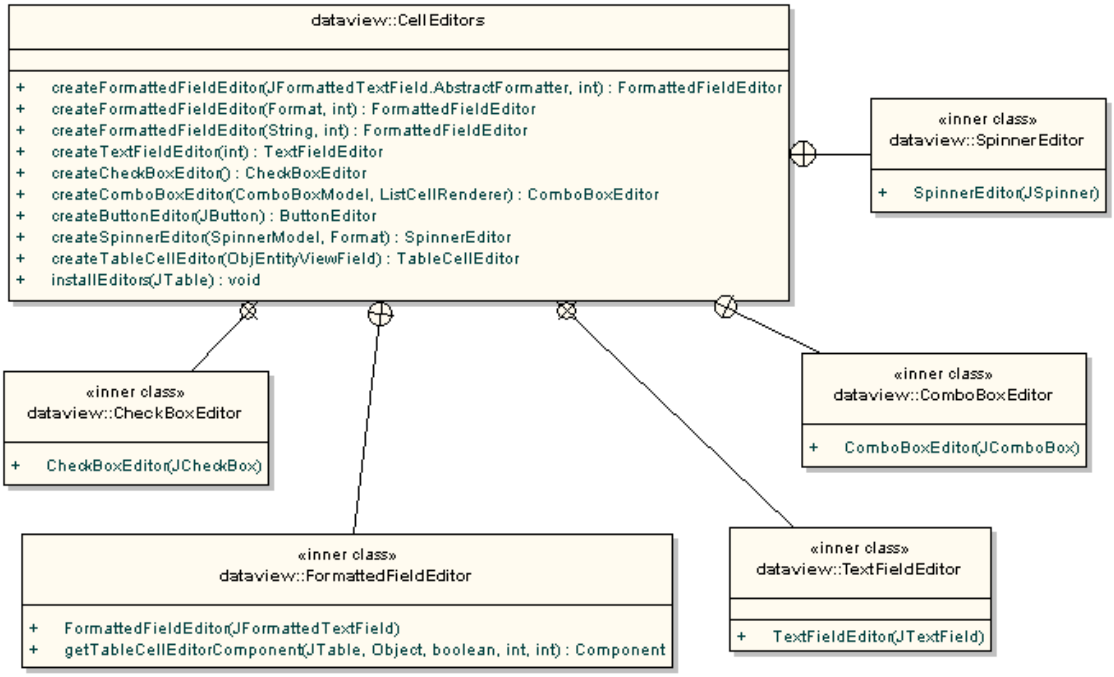
org.objectstyle.cayenne.dataview.FormatFactory depicted on the next figure provides the means to create instances of the edit and display formats used by ObjEntityViewFields. There are several formats defined in the java.text package. In turn, the dataview library defines convenient MapFormat somewhat similar to java.text.ChoiceFormat. instead of formatting doubles by inclusion in range criterion as ChoiceFormat does it maps a set of objects of any type to string values. For example you can map Booleans to "Sure" and "No way" strings or you can map String keys to some descriptive String values, etc. It is easy for users to add custom formats of their own. They should subclass FormatFactory and create new Builders for their formats. That done, all they need is to configure the display and edti formats in ObjEntityViewFields with the class names of their Format classes and the patterns (and, maybe, some other properties).



The following figure presents the utility class CellRenderers and several ready-to-use cell renderers for different types of ObjEntityViewFields. They are used in JTables, JLists, and JComboBoxes. The methods defined in the CellRenderers class will save your time when configuring a JTable to render values of the types available for use with ObjEntityViewFields

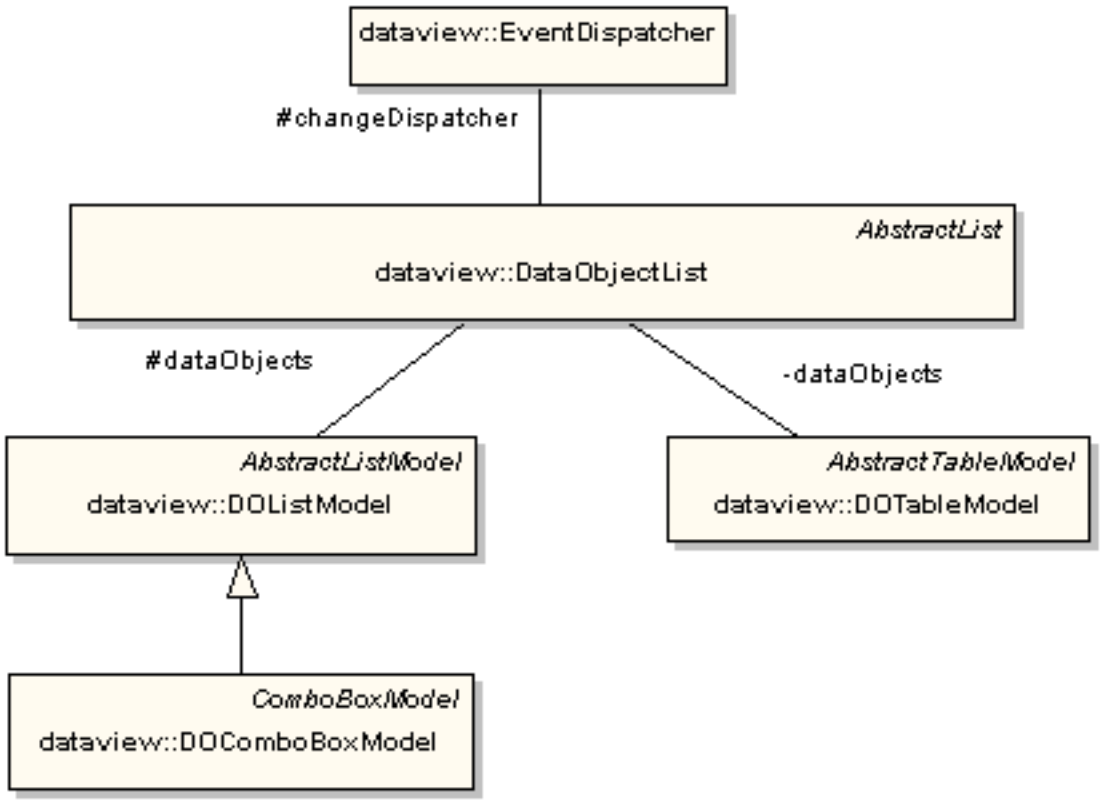


The purpose of CellEditors is the same as that of CellRenderers. The editors are used when there is a need to edit data in JTables and such.



The convenient notion of DataObjectList is defined in the library. It is a specialized container to store Cayenne DataObjects (usually of the same type). It fires events when modified and DOTableModel,

DOListModel, DOComboBoxModel wrap it and being configured with an ObjEntityView / Field are handy to provide access to these data objects with JTables, JLists, and JComboBoxes. In fact, they make the visual components data aware.



This is the Data View DTD in full. There are no comments in it yet. On the other hand it is small. And the names are self-explanatory once you got to know the Data View structure. You can see the examples of its usage in the next chapter.

```
<!ELEMENT caption ( #PCDATA ) >

<!ELEMENT data-view ( obj-entity-view+ ) >

<!ELEMENT default-value ( #PCDATA ) >

<!ELEMENT pattern ( #PCDATA ) >

<!ELEMENT edit-format ( pattern? ) >
<!ATTLIST edit-format class NMTOKEN #REQUIRED >


<!ELEMENT display-format ( pattern? ) >
<!ATTLIST display-format class NMTOKEN #REQUIRED >

<!ELEMENT lookup EMPTY >
<!ATTLIST lookup obj-entity-view-name NMTOKEN #REQUIRED >
<!ATTLIST lookup field-name NMTOKEN #REQUIRED >

<!ELEMENT field ( caption?, lookup?, edit-format?,
                  display-format?, default-value? ) >
<!ATTLIST field obj-relationship-name NMTOKEN #IMPLIED >
```

```
<!ATTLIST field pref-index NMTOKEN #IMPLIED >
<!ATTLIST field name NMTOKEN #REQUIRED >
<!ATTLIST field editable ( false | true ) #REQUIRED >
<!ATTLIST field obj-attribute-name NMTOKEN #IMPLIED >
<!ATTLIST field calc-type ( nocalc | lookup ) #REQUIRED >
<!ATTLIST field data-type ( Object | String | Money |
                           Integer | Double | Percent |
                           Date | Datetime | Boolean ) #REQUIRED >
<!ATTLIST field visible ( false | true ) #REQUIRED >

<!ELEMENT obj-entity-view ( field+ ) >
<!ATTLIST obj-entity-view name NMTOKEN #REQUIRED >
<!ATTLIST obj-entity-view obj-entity-name NMTOKEN #IMPLIED >
```

Data Views in Action

This chapter shows the results of application of Data Views in a visual administration tool. The peculiarity was that while the main participating ObjEntities were known at the time of development the lists of their attributes would vary and contain different attributes for different client systems. Also the lookup ObjEntities (database tables) referred by the main ObjEntities were not defined preliminarily. Therefore a simple enough way to reconfigure the interface without actual recoding was required. Here the administration tool knows the names of the data views corresponding to the ObjEntities in question only. At the runtime it uses the information about their fields and various classes from the dataview subpackage to build the dialogs automatically. It tries to choose the most appropriate editor components (text fields, formatted fields, combo boxes, choice boxes, etc) and renderers based on the field properties. It also pays attention to the visibility and editability of the fields and orders the Swing components used to display or edit the data in the data objects based on the preffered indices defined for the fields. The lookup entitties are recognized and the data from the corresponding tables are used to provide lists of values in combo boxes or find the value to display in the label bound to the lookup field.

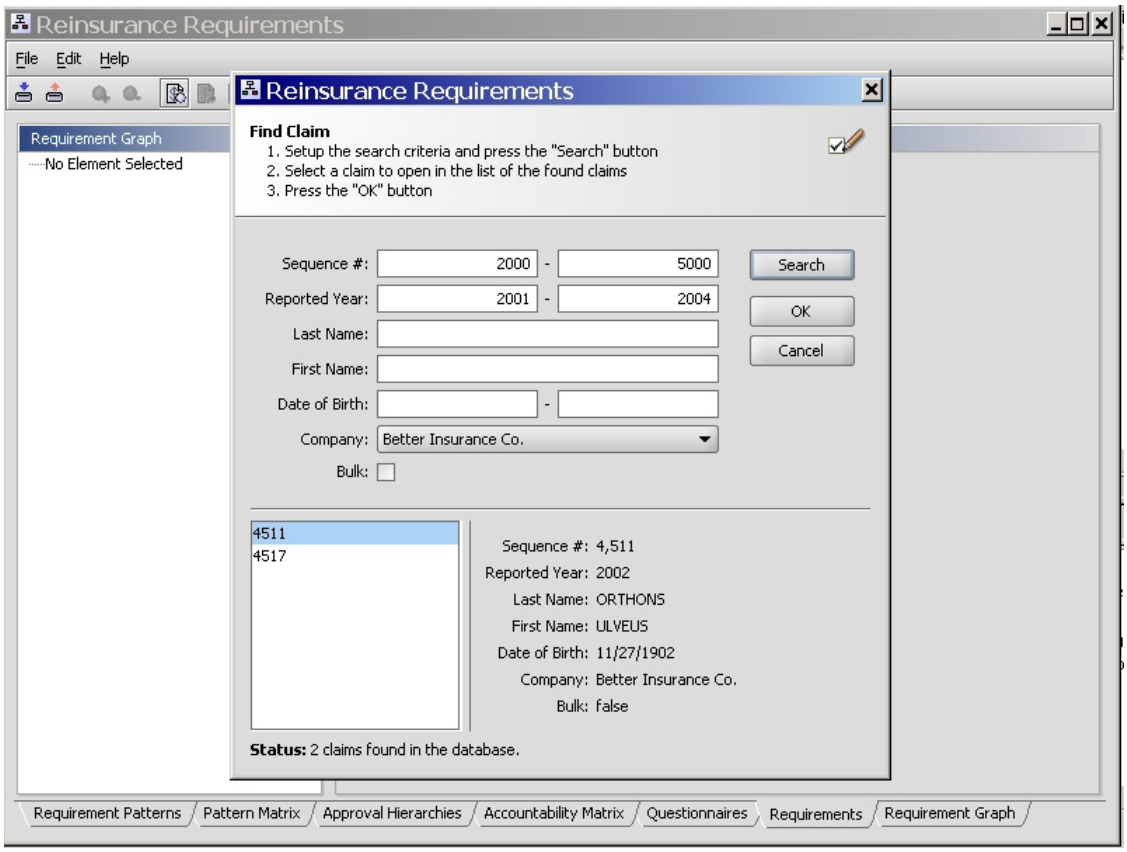
The first example shows an extract from the data view xml file that defines a ObjEntityView and the relevant lookup ObjEntityView for the Claim ObjEntity. The screenshot gives a standard look of the form used to search for claims in the database by several criteria. This form builds itself dynamically and recognizes what input fields and labels to display, what formats to use, and even what qualifiers to define in the SelectQuery.

```
  <obj-entity-view name="ClientClaimView" obj-entity-name="ClientClaim">
    <field name="claimSequence"
        obj-attribute-name="claimSequence"
        data-type="Integer"
        editable="true"
        visible="true"
        pref-index="0"
        calc-type="nocalc">
      <caption>
        Sequence #:
      </caption>
      <edit-format class="java.text.DecimalFormat">
        <pattern>#######</pattern>
      </edit-format>
      <display-format class="java.text.DecimalFormat">
        <pattern>#,###,###</pattern>
      </display-format>
    </field>
    <field name="reportedYear"
        obj-attribute-name="reportedYear"
        data-type="Integer"
        editable="true"
        visible="true"
```

```
      pref-index="1"
      calc-type="nocalc">
    <caption>
      Reported Year:
    </caption>
    <edit-format class="java.text.DecimalFormat">
      <pattern>0000</pattern>
    </edit-format>
    <display-format class="java.text.DecimalFormat">
      <pattern>0000</pattern>
    </display-format>
  </field>
  <field name="lastName"
      obj-attribute-name="lastName"
      data-type="String"
      editable="true"
      visible="true"
      pref-index="2"
      calc-type="nocalc">
    <caption>
      Last Name:
    </caption>
  </field>
  <field name="firstName"
      obj-attribute-name="firstName"
      data-type="String"
      editable="true"
      visible="true"
      pref-index="3"
      calc-type="nocalc">
    <caption>
      First Name:
    </caption>
  </field>
  <field name="birthDate"
      obj-attribute-name="birthDate"
      data-type="Date"
      editable="true"
      visible="true"
      pref-index="4"
      calc-type="nocalc">
    <caption>
      Date of Birth:
    </caption>
    <edit-format class="java.text.SimpleDateFormat">
      <pattern>MM/dd/yyyy</pattern>
    </edit-format>
    <display-format class="java.text.SimpleDateFormat">
      <pattern>MM/dd/yyyy</pattern>
    </display-format>
  </field>
  <field name="clientCompany"
      obj-relationship-name="clientCompany"
      data-type="Object"
      editable="true"
      visible="true"
      pref-index="5"
      calc-type="lookup">
    <caption>
      Company:
    </caption>
    <lookup obj-entity-view-name="ClientCompanyLookup"
        field-name="companyName">
    </lookup>
```

```
  </field>
  <field name="bulk"
      obj-attribute-name="bulk"
      data-type="Boolean"
      editable="true"
      visible="true"
      pref-index="6"
      calc-type="nocalc">
    <caption>
      Bulk:
    </caption>
    <default-value>
      false
    </default-value>
  </field>
</obj-entity-view>


<obj-entity-view name="ClientCompanyLookup"
                 obj-entity-name="ClientCompany">
  <field name="companyName"
      obj-attribute-name="companyName"
      data-type="String"
      editable="false"
      visible="true"
      pref-index="0"
      calc-type="nocalc"/>
</obj-entity-view>
```



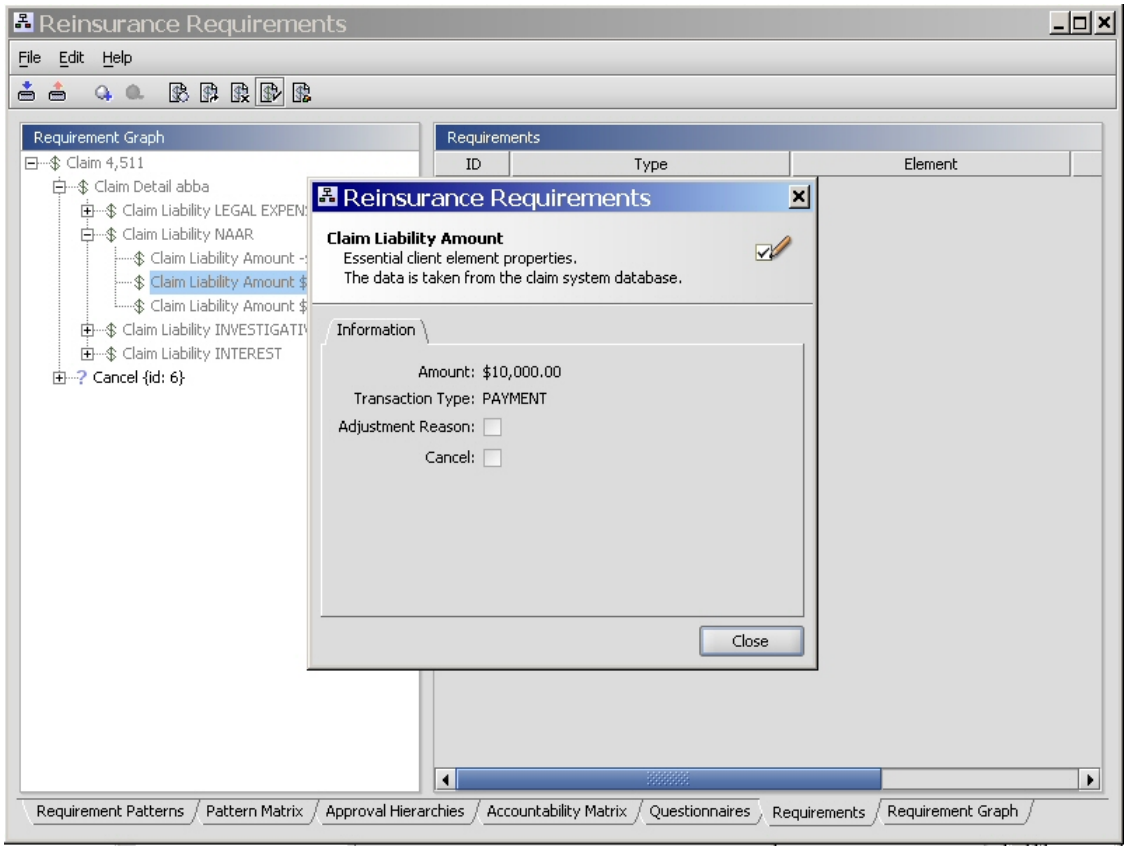
The next example is of similar kind. It demonstrates how several lookup fields can be configured to point to the different fields of intrest in the same lookup view. The resulting ObjEntityView is used to

build the dialogs of read-only properties for the user's reviews.

```
  <obj-entity-view name="ClientDirectLiabilityAmountView"
                   obj-entity-name="ClientLiabilityAmount">
    <field name="amount"
        obj-attribute-name="amount"
        data-type="Money"
        editable="false"
        visible="true"
        pref-index="0"
        calc-type="nocalc">
      <caption>
        Amount:
      </caption>
      <display-format class="java.text.DecimalFormat">
        <pattern>$###,###,##0.00</pattern>
      </display-format>
    </field>
    <field name="transactionType"
        obj-relationship-name="transactionType"
        data-type="String"
        editable="false"
        visible="true"
        pref-index="1"
        calc-type="lookup">
      <caption>
        Transaction Type:
      </caption>
      <lookup obj-entity-view-name="ClientTransactionTypeLookup"
          field-name="description">
      </lookup>
    </field>
    <field name="transactionAdjustmentReason"
        obj-relationship-name="transactionType"
        data-type="Boolean"
        editable="false"
        visible="true"
        pref-index="1"
        calc-type="lookup">
      <caption>
        Adjustment Reason:
      </caption>
      <lookup obj-entity-view-name="ClientTransactionTypeLookup"
          field-name="adjustmentReason">
      </lookup>
    </field>
    <field name="transactionCancel"
        obj-relationship-name="transactionType"
        data-type="Boolean"
        editable="false"
        visible="true"
        pref-index="2"
        calc-type="lookup">
      <caption>
        Cancel:
      </caption>
      <lookup obj-entity-view-name="ClientTransactionTypeLookup"
          field-name="cancel">
      </lookup>
    </field>
  </obj-entity-view>


  <obj-entity-view name="ClientTransactionTypeLookup"
```

```
                obj-entity-name="ClientTransactionType">
    <field name="description"
        obj-attribute-name="description"
        data-type="String"
        editable="false"
        visible="true"
        pref-index="0"
        calc-type="nocalc"/>
    <field name="adjustmentReason"
        obj-attribute-name="adjustmentReason"
        data-type="Boolean"
        editable="false"
        visible="true"
        pref-index="1"
        calc-type="nocalc"/>
    <field name="cancel"
        obj-attribute-name="cancel"
        data-type="Boolean"
        editable="false"
        visible="true"
        pref-index="2"
        calc-type="nocalc"/>
</obj-entity-view>
```



You might notice the XML configuration files have very simple and easily understood structure and the configuration can be used in a number of ways. It is needless to say the manual editing of such files would be the most cumbersome task that no one could be content with. Luckily the GUI tool called "Data View Modeler" and intended to simplify the editing of the configuration files is about to come into good shape. Then the work with data view configuration files will be no more difficult than the Cayenne data map modification in the Cayenne Modeler.

To Do List

The list given below summarizes the directions of the further development and the useful features not supported yet but already kept in mind.

- Extend the DTD and java library to support validation rules, at least the range validation.
- Finish the Data View Modeler
- Make dataview configuration an optional part of the Cayenne project, extend the project's DTD accordingly.
- Give a better thought to the internationalization. Support internationalized field captions at least.
- Create or import specialized classes for the Money, Date, Datetime, Percent (?) types. In fact, it was planned to add the home grown Money type to the framework but as this type is going to be implemented in Apache Commons Lang, the author has decided to sit on his back and wait the results for a while.
- Consider the possibility of adding the Text Mask format to the library. Swing has MaskFormatter but there is no corresponding format in java.text.
- Lookups can be improved on to listen to the data changes in the corresponding data object lists.
- Add the notion of the order based on the field values (lexicographic, natural, etc) so whenever a user has a need to sort some table of data objects by values in a column we would know how to do that.
- Implement custom tree renderers and a tree model.
- Add easy to use GUI factories to build forms for ObjEntityViews dynamically, based on the JGoodies forms library.
- Implement conditional data view configuration loading so the application could choose which data views are appropriate for a given set of ObjEntities.
- Introduce calculated fields. They will not correspond to predefined ObjAttributes, instead, their values would be calculated on the fly.
- Extend the data view DTD to give more hints to the GUI components how to build themselves and function in different situations. This must be well pondered over.

Dependencies and Requirements

There are several things that are needed to compile and use the Data View code located in the org.objectstyle.cayenne.dataview package.

- JSDK 1.4. Normally the bulk of Cayenne needs JSDK 1.3 but "Cayenne Data Views" is assumed to be used for the Swing GUI development and so the choice of Java version should not be an issue.
- Cayenne 1.0. In fact, the code should work with the earlier beta versions as well. When the support of Data Views is added to Cayenne projects the library will stop working with the previous versions of Cayenne

- Apache Jakarta Commons-Lang 2.0 or later.

- JDOM beta 9 or later.