Programming ArchiTech

The intention of this document is to give a description of the way ArchiTech has been programmed, in order to make anyone who wants to take a look to the code easier to understand it, and to justify some decisions that have been taken.
To do that, a kind of top-down structure is used in the document (starting from the plug-ins structure and ending in the details of the implementation).

Plug-Ins Structure

Special thanks needs to be given to David Cerdan, who has managed to find the way to build a 4-plug-in structure which permit the integration of the use of Hibernate with the main plug-in. He, in turn, has had the help of a specialized web site[1] to get inspiration.

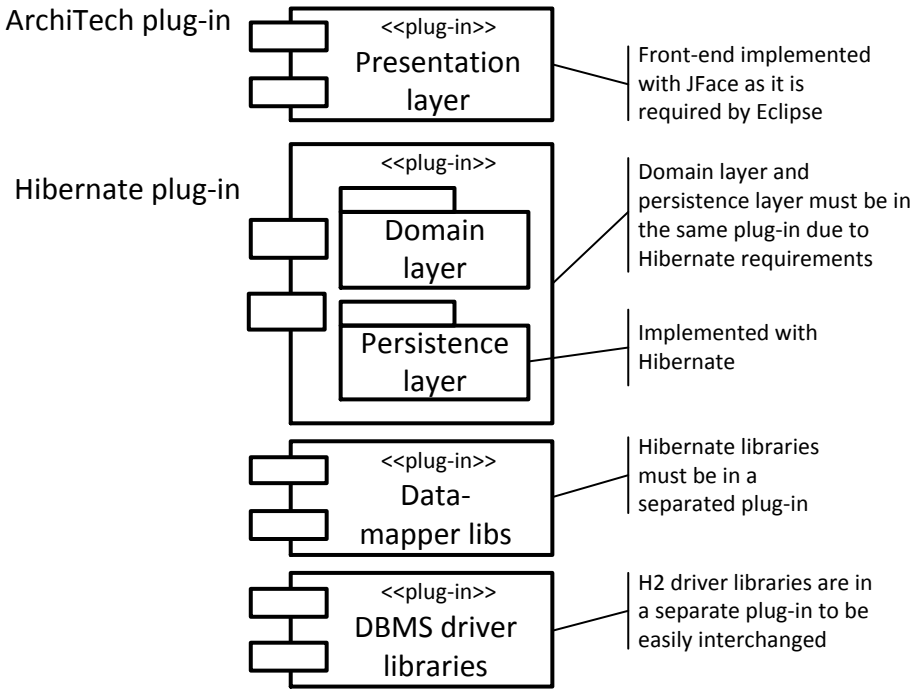


Fig. 1 Plug-ins structre

The structure shown in Figure 1 is formed by 4 plug-ins:

- DBMS driver libraries plug-in: The only content of this plug-in are the libraries (driver) needed to work with the DB. In the case of ArchiTech is the library "h2-1.1.118.jar". Is important for this plug-in to export the packages (included in the library) which need to be used to connect, access, ... the database. In the case of ArchiTech is the package org.h2. Doing it allows other plug-ins to use the functions residing in this plug-in, concretely the functions residing in the library, and more concretely the functions residing in the exported packages.
- Datamapper libraries Plug-in: This plug-in has the same purpose of the above one but now with Hibernate instead of the DB Drivers. The set of libraries which conforms Hibernate needs to be included in this driver and all the packages which contains useful classes need to be exported.

[1] http://entwickler.de/zonen/portale/psecom,id,101,online,1082,.html

- Hibernate plug-in: This is the first of the two main plug-ins of the system. The role of this plug-in is to encapsulate the use and configuration of Hibernate and the classes in the domain layer. A further description is given later. It is important to remember that this plug-in has to export all the packages which need to be used by other plug-ins (specially the ArchiTech plug-in). Another important point is that this plug-in has the Hibernate Libraries Plug-in as a dependency.
- Presentation layer plug-in (ArchiTech plug-in from now on): This is the second of the main plug-ins of the system. This plug-in contains the presentation logic and acts as a front-end to the user. A further description is given later.

Hibernate Plug-in

This plug-in contains four resource groups: Hibernate configuration and mapping files, the Domain Controllers, the Domain classes and the Persistence Controllers. All together form the Presentation and Domain Layer

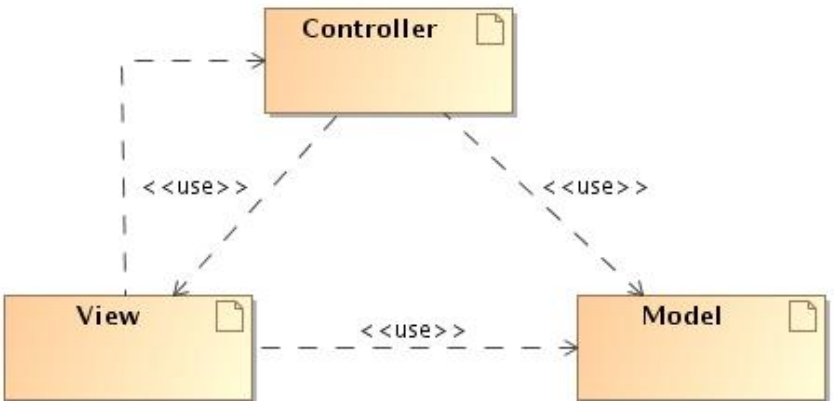
An XML document is used to manage all the Hibernate configuration, and there is also an XML mapping document for each of the domain classes (except the generalization hierarchies which are grouped in one file per each), which are placed in the same package as the classes that map.

There is a Controller which encapsulates all the operations that need to invoke Hibernate classes (build Configuration, begin Transaction, ...), used by the ArchiTech plug-in to work with Hibernate in an indirect way.

The third of the resources group are the domain classes, which are the main component in the Domain Layer.

ArchiTech Plug-in

This plug-in has been structured like a relaxed Model View Controller, a structure driven by the use of the JFace technology which almost constraints any plug-in to have an structure similar to that:



The Controller, in fact, is very light-weight; it only contains classes that describe how to present the model objects in the view, in contrast with what normally the controller does in a MVC (listen to changes, etc.). The controller, so, is coupled with the view and model parts.

The main parts of this structure, so, are the View and Model (I will call Domain to this from now on) parts:
The View is the responsible of displaying a representation of the Domain to the user and allowing him to operate with it. It has to be always synchronized with the changes made in the Domain, to do that, a coupling between the View and the Domain is permitted. The View also knows the Controller classes because the different Components used to show data use this classes to represent the data.

The Domain encapsulates the main functionality of the system. Noticed that this part is encapsulated in the Hibernate plug-in and, thus, is not described here.

Domain

Apart from the domain classes described previously, the Domain contains two main classes: the Model class and the Controller class.

The Controller class is a controller which acts the same way as the typical Domain Controller of a 3-layered architecture (it collects data from the View and inputs that data to the Persistence, it also works with the domain objects). In this case it is a façade controller, it encapsulates the operations of all the use cases. The decision of implementing it as a façade is motivated because there is not a lot of use cases and, apart from that, most of the use cases are similar, so it's easy to structure all the operations in the same class.

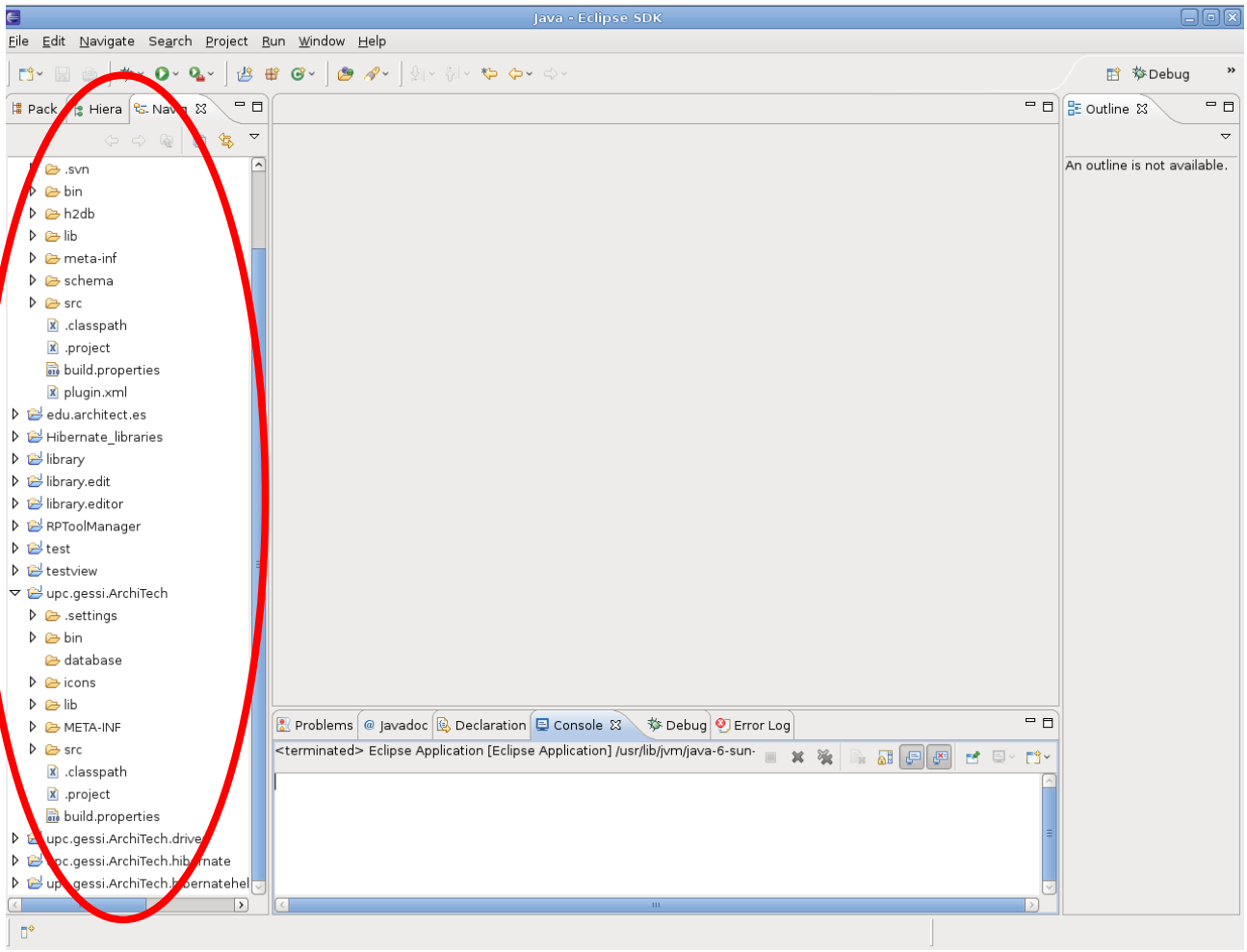
The Model class serves as a synchronization method for the View and the Domain parts. The Model only stores references to all the domain objects, and when a modification is made to any of this objects, the view apply this changes. The set of objects which are referenced from this class are all the set of objects which form part of the system, and if a new object is added, it has to be added to this class.

View

The implementation of the view classes is based in the Java JFace and SWT technologies, which provide classes and methods to build all kind of components able to show data to the user and let him manipulate it. The decision to use these libraries is motivated because the Eclipse IDE and all it's plug-ins use it, so it provides an easy way to integrate ArchiTech with the IDE, and apart from that, it has been proved that using other libraries (like Swing for example) can be problematic. Surprisingly several design decisions have arisen because of the use of these libraries.

The View is composed of two main groups of classes: The viewers classes and the dialog ones.

The JFace viewers look like that:



A viewer shows a set of domain objects in a list, a tree or a table, and the user can select those objects and run actions on them. ArchiTech has a viewer for each of the main domain concepts (Questions, Properties and Requirements, Decisions, Elements and DataTypes), all of them show the items as a list, except the Properties and Requirements one which shows the items as a tree. It has been chosen to use viewers because is a good and typical way to display information in Eclipse and the user feels comfortable using them.

After the information of the domain objects has been displayed in a viewers, the user may run actions on them, such as create a new object, edit the selected object, ... typical dialogs are used for doing so. The only function of the dialogs is to collect data from the user in an organized and clean way, and pass it to the Domain Controller.

Controller

The Controller has to types of classes: Content Providers and Label Providers.

The first ones are completely irrelevant in ArchiTech, they just make a required type casting. The second one tells how to present the objects, concretely tells what to use to identify the object (for example the attribute name of the object) and what icon to use when showing it.

Persistence

This part is composed of two groups of resources: The Hibernate Controller and the DAO classes.

The first one has been described previously. The DAO classes are based in the DAO (Data Access Object) design pattern, there is one class of this type for each domain class to which persistence operations are executed, and it encapsulate all this operation. The decision of implementing this type of classes makes the code of the Domain Controller cleaner and separate the persistence operations from it.

How the DomainController works?

The DomainController class has been the most hard class to implement. Several different solutions have been tried but it has taken hard work to find one solution that works.

First, I will try to describe the problem in order to make easier, then, to understand the decisions taken. As said, the View has always to be synchronized with the Domain, this include that all the object references should be the same in the objects represented by the View and the objects in the Domain, if not, when navigating the Model object graph we could end in objects that are different from the objects shown in the View. The problem is that, when retrieving different objects from the DB references are not shared between them, different instances of the same object are brought to memory.
Let's see this with an example. Suppose we have a Question Q that is of the Type T1. When the user first opens the Questions viewer, all the Questions are retrieved from the DB and stored in the Model, but also, as the Type T1 forms part of Q it is also retrieved and placed somewhere in memory with the only reference to it from Q. Now the user opens the Types view, so all the Types are retrieved from the DB and stored in the Model. So now, in the Model we have two sets of objects, one contains Q and the other contains T1, but Q does not reference to the T1 of that set! So all the changes made to the Type of Q will not be reflected in the Model set of Types.

All the main operations of this class start and end a Hibernate Transaction. The reason for doing that is because in each function several objects can be retrieved and modified and thisHow the objects travel from the DB to the View permits doing all kind of updates and gets from the DB while performing the application logic.
During the transaction all the operations with the affected objects are made.
To solve the problems previously described it has been decided to use a method in which the Domain and the View are not completely synchronized, in fact, the objects always have a synchronized values in their “main” attributes (name, length, ...) but the references to other objects are not synchronized.
To make all this work properly, every time an attribute of an object needs to be modified a 3-step process is followed:

- Remove the object from the Model class.
- Get the same updated object from the DB.
- Put the object again in the Model.

When an object has to be removed or updated from the Model the same process is followed to ensure this object is updated (it's references to other objects are updated).
So, all the functions of the Domain Controller work like that, the rest are minimal and not important implementation details.

How the objects travel from the DB to the View?

The domain objects are not loaded into local memory until they are needed. So the Model class retrieve the objects from the DB (through the Domain Controller and this through the DAO classes) the first time some class asks for them. Concretely when the user opens a view, the view takes as input a subset of the set of objects of the Model, and the Model retrieves them from memory if it's the first time those objects are asked.
Once the object are in memory, the Domain Controller is responsible of maintaining (with the limitations explained previously) the memory objects updated and the tables of the database (through the DAO classes) updated too. Doing it this way, when the users stops using ArchiTech, the DB is in a consistent state.

Something more about the viewers and the dialogs

The viewers, in fact, are components which are embedded in a more generic component called ViewPart.
The viewers have to define a Content Provider and a Label Provider, which are classes of the Controller and then has to define it's inputs elements, which are get from the Model.
The ViewParts, but, have also actions and menus.
The actions define a behaviour in their run methods, which let the user operate with the objects that are showing. The actions normally show a dialog to let the user input data and then invoke some methods on the Domain Controller with the that data, although some actions (such as delete) may not show any dialog.
The actions then are added to menus, which can show icons or text, or can be pop-up menus, ...
The rest of the code and the code of the dialogs is just the declaration of different type of SWT components and the location of them in the dialog (using a grid layout). The only, maybe, interesting thing of the dialogs is that to manage the rules information, a special data type is used which stores the informatino locally and then this type is passed to the Domain Controller, who creates the rules using the data there, so, when any of the components that composes the rules tab is used, this special data type have to be updated.

What about the DAO classes?

Those classes has a method for each relevant operation related to persistence that has to be done with the objects. Some of those methods define a query (to get all the objects from the database for example) and execute it others just call a “save” or “update” with the object. One important point about those classes is that the queries to get objects also have to bring all the objects associated to it in order to be able to navigate the object graph outside a database transaction (the viewers may do that), so “fetch join”s are used to achieve that.
Another point to take into account is that when an object is updated or re-associated with a session (lock) it may be the case that there is more than one object with the same identifier in memory (as a consequence of the problems described in the Domain) so, before updating or locking the most recent object has to be merged with the other forming just one object.