Archimedes – The Open CAD
A software for technical drawings based on
Eclipse's Platform

Hugo Corbucci, Mariana V. Bravo

27 de agosto de 2007

Resumo

Archimedes is a free software for computer aided design (CAD) focused on architecture. The project was recently moved to Eclipse's Rich Client Platform. In this article, the result of such migration will be presented, as well as the extension points that allow new funcionality to be added to the program.

1 Introduction

The Archimedes project ([?]) was born from a challenge made by Jon "Maddog" Hall during FISL 6.0 (6th International Forum of Free Software) to create an open source solution for professional users of AutoCAD. The initial goal of the project was to become a multi-platform FOSS (Free and Open Source Software) with architects as target users. In order to meet their needs, besides the developers, the team counts with architects (students and professionals). They help to define what should be done on each iteration of the development according to eXtreme Programming ([?], [?]).

The software's development started in march 2006 at the eXtreme Programming Laboratory at IME-USP. At that time, the goal was to allow architects to work on the computer just like they would do in a plane table. With this software, it would be possible to elaborate technical projects without changing the professional's concept of their work, which would fulfill their initial needs.

After almost a year of work, the project reached a reasonable state, however it was clear that there were some lacks. It was not friendly to new users because of the poor menus as well as the absence of embedded help in the

software. Besides, as it was expected that the software allowed different forms of drawing and working from the existing one, both developers and users wished the software was more flexible. This was the reason why the team decided to migrate the Archimedes to the Eclipse's ([**?**]) RCP (Rich Client Platform), since it is widely adopted by the FOSS comunity and because it had several features required.

In the next section, the initial system of Archimedes, which should be kept; and in the next, the architecture resulting of the migration.

2 An overview of the system

The software works using a command prompt with a drawing area as you can see in Figure 1. Commands are input through the keyboard and the parameters can be sent in a text way as well as with the mouse. This way, the professionals can change their work context quickly without having to search for shortcuts in big complex menus. What follows will describe the interface layer that is responsible of the data flow regarding the user.

Â Â [width=.5]Screenshot.png Â

Figura 1: The main window of Archimedes in MacOS X
Â

2.1 The interface layer

To handle two kinds of inputs in the same way, the informations received by the mouse are transformed in text. Those data, just like the ones obtained from the keyboard, are sent to an input controller (`InputController`).

Those informations can be used to activate a command, send parameters to this command until it ends and then execute the action it will generate. From the system's point of view, what is executed is the action, that implements the `Command` pattern ([**?**]), where its name comes from. The class that generates this action is a `Command` factory, therefore is called a `CommandFactory`. For example, to create a line, first you must activate line factory, then two points must be defined as parameters and finally, the action that will create the line.

To process this information, the `InputController` depends on its state. There are two main possibilities:

- There is no active factory

Â In such case, the `InputController` asks to the command parser
Â (`CommandParser`) to send him the factory associated to the user's
Â input in order to activate it.

- There is an active factory

Â If that's the case, the active factory should deal with the inputs.
As many Â parameters may exist to create a command and many
factories receive very Â similar parameters, the translation from the
user input to system objects is Â done by an object called `Parser`.
On each step, the factory informs which Â `Parser` should handle the
next inputs and awaits for it to return a specific Â object. This way,
the factories can work with higher level objects and reuse the Â input
processing work realized by the parsers.
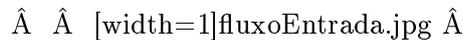
Â Â [width=1]fluxoEntrada.jpg Â

Figura 2: Sequence diagram for the data input
Â

Once the user sent all necessary parameters to a factory, it creates a
`Command` that is received by the `InputController`. This controller send the
command to the main system controller (`Controller`). The flow described
so far can be seen in the Figure 2. At this point, the software leaves the
interface layer to enter the model layer which will be described shortly. An
important note is that the flow from the interface level to the model level
can only be made through the controller layer according to the `MVC` pattern
([?]).

2.2   Model layer

Before analysing the received command, the controller is charged to holds
information about the current state of the model to allow the action to be
undone. After that, the command is executed on the active drawing. This
drawing (`Drawing`) is the main work unit of the model. It contains work
layers (`Layer`) that contains elements (`Element`), like lines, arcs, etc.

The model system uses the drawing as a `Composite` ([?]) delegating all
the render requests to the elements below on the tree. That way the drawing
is rendered from the components.

3 RCP architecture

The RCP architecture ([?]) is organized into *plugins*. There is a common core to all the application based on this platform and this core is responsable to load a *plugin* defined as the initial one. In Archimedes' case, this plugin, identified as `br.org.archimedes`, contains only the flow control explained in the last section. Alone, this *plugin* won't enable the user to create anything but an empty drawing. However, it will be used as a base for all the other *plugins* so that they add the desired feature, like elements and command factories.

This structure grants a good flexibility, since with it one can create, for example, a set of *plugins* to work on a two dimension space and another one to work on three dimensions. Besides, the RCP allows to easily organize the menus and provides a good base to build an organized help system.

To be loadable, each *plugin* must have a file named "plugin.xml" in its root. This XML file ([?]) describes the contributions this *plugin* does to the application. Those contributions are called extensions, and each extension corresponds to some exntesion point from another *plugin*. The Archimedes' core defines a few extension points so that new features can be added.

3.1 Available extensions

Beside the extension points defined by Archimedes, there are others already defined by RCP like `org.eclipse.ui.actionSet`, that allows to create menu elements, or `org.eclipse.ui.view`, that allows to add graphical elements to the interface. The following sections will explain the extension points defined by Archimedes.

3.1.1 Element (`br.org.archimedes.element`)

This point allows to create new elements in the drawing, such as simple elements, like lines and circles, more complex ones, such as walls and windows, or even elements that should not be shown in a final drawing, like notes. An element extension should have an identifier that allows the core to keep a record of existing elements. Besides that, the *plugin* must point one class that extends `br.org.archimedes.interfaces.Element`, so that the system can load the element when needed. Optionally, one can define a factory and her shortcut which will be responsible to create that element.

According to what was presented from the system so far, this extension point is not entirely necessary since the program can lead with the interface

`Element` without having to know which implementation of it is instanciated. This point, however, is required by the exporting system that will be explained below.

3.1.2 Exporter (`br.org.archimedes.exporter`)

To enable *plugins* that export a drawing to new file formats, there is an extension point for exporters. The *plugin* that extends this point should only export the drawing itself and the layers. It must not have any code to expot the specific elements. Archimedes' core is encharged to look out for exporters of that format for each element (see section 3.1.3). Besides, the *plugins* registry is used to list all available exporters to the user.

The advantage of that architecture is that if an element or a specific exporter is not available, the exporter globally still works. It just ignores that specific element. This means that there is absolutely no dependency between the exporter and the elements that it can export.

3.1.3 Element exporter (Â `br.org.archimedes.elementExporter`)

A *plugin* that extends this point must define what file format it exports and for what element id it works. Besides, it needs an identificator and a class that implements `br.org.archimedes.interfaces.ElementExporter`. This interface contains only one method that will be invoked by the export system when the user exports a drawing in that file format and contains that kind of element. The export flow can be seen in the sequence diagram at Figure 3.

Â Â [width=1]fluxoExport.jpg Â

Figura 3: Exporting flow sequence diagram
Â

3.1.4 Importador (`br.org.archimedes.importer`)

The import extension has the same goals as the export one: allow new importers regardless of the available elements. However data reading may not have a very defined structure that allow delegation of tasks to specific importers. That is why the same solution does not fit.

Because of this, the import system is much less restrictive. It just delivers an input stream with data and expected a Drawing in return. No mather how the responsible class does to produce this drawing. The core provides a

small element factory that allows to request the creation of an element with certain parameters. This allows importers to create elements without having to know if they are present or not.

3.1.5 Native format (`br.org.archimedes.nativeFormat`)

A native format is just a format that has both an exporter and an importer. The contract of a native format is that no information should be lost between the export of a drawing and its importation. However, this cannot be tested automatically by the software since it might not (and will not) know what was the previous state of the drawing. Therefore it falls back to developers to follow that rule.

3.1.6 Factory (`br.org.archimedes.factory`)

At last, there is an extension point that allows the addition of new command factories. Those factories allow developers to implement system commands to the users, either new or compositions of existing commands, to perform new operations on existing drawings. This way, it is possible to easily increase the number of available commands to the user.

This extension point requires only an identificator and a class that implements `br.org.archimedes.interfaces.CommandFactory`. A name and a set of suggested shortcuts are optional itens for this extension. Both the identificator, the name and the shortcuts are strings that the user can use to activate this factory.

4 Conclusion

Thanks to the migration of the project, we expect that the amount of code contribution increases since the developers no longer need to work with the central team to implement their own modifications to the software. This flexibility could only be obtained using the *plugins* system that is the core of Eclipse's Rich Client Platform.

With time, we hope that developers ask for other extensions so that the central team can make the software more flexible and gather all those improvements in a central repository related to Archimedes.