Web Applications: A Simple Pluggable Architecture for Business Rich Clients*

Duncan Mac-Vicar and Jaime Navón

Computer Science Department, Pontificia Universidad Católica de Chile
{duncan,jnavon}@ing.puc.cl

Abstract. During the past decade we have been witnesses of the rise of the Web Application with a browser based client. This brought us ubiquitous access and centralized administration and deployment, but the inherent limitations of the approach however, and the availability of new technologies like XML and Web Services has made people start building rich clients as business applications front ends. But very often these applications are tied to the development tools and very hard to extend. We propose a clean and elegant architecture which considers a plugin based approach as a general solution to the extensibility problem. The approach is demonstrated by refactoring a simple application taken from a public forum into the proposed architecture including two new extensions that are implemented as plugins.

1 Introduction

The complexity of today's business information technology infrastructure made Web applications reintroduce the server-based model but with a zero footprint on the client, offering centralized administration and deployment. Users can benefit from ubiquitous access from any internet-connected device using a web browser, which lack the functionality, offline operation, flexibility and performance of desktop applications as a result of a limited user interface technology and multiple round trips to the server to execute trivial tasks.

In the past, rich client and fat client were almost synonyms. Nevertheless in the last few years the emergent technology of Web Services eliminates the need for rich clients to be fat applications. Rich clients can integrate into SOA environments accessing both server located data and local acquired data (hardware devices, etc).

The requirements of next generation rich clients include the ability to use both local and remote resources, offline operation, simple deployment and real time reconfiguration support.

In this paper we will discuss such architectural and extensibility problems and propose an architecture that could be applied to any platform where rich clients are being deployed.

* This research is supported in part by the Chilean National Fund for Science and Technology (Fondecyt Project1020733)

D. Lowe and M. Gaedke (Eds.): ICWE 2005, LNCS 3579, pp. 500–505, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 Rich Clients Architectural Problems

In rapid application development environments, visual form editors are used to design view components. The editor generates the user interface code as you change it and mixes that code with your event handler.

Once the application has been deployed it may suffer enough change so that it could justify a redeployment, but sometimes, adding a few new small features is all we might want. In the case of mission critical applications that require continuous operation, rebuilding and restarting the application is not an option, and therefore even a good architecture is not enough to ensure extensibility.

Furthermore, we would like that third party vendors or even the user itself be able to build small extensions to the original application as an alternative to wait for a new release from the original vendor.

3 A Proposed Simple Rich Client Architecture

We propose a simple architecture (Figure 1) to build applications which do not suffer the problems described above. For solving the architectural problems we leverage the MVC[1] design pattern splitting the application in three layers in a very clean and elegant manner. But perhaps our main contribution is the way we handle the extensibility problem through the use of plugins managed at the controller layer by a plugin engine.

3.1 The View

A view in our model is represented as a decoupled class containing only visual elements. No event handling is provided at this level. This allows for generating the views from metadata. There are several XML-based technologies like Mozilla XUL[1] and Qt UI[10] schemas that can be used to describe the user interface.

3.2 Controller

There is no single Front Controller[5][2] but as many controllers as view components we have. This kind of micro-controller is modeled as a class inheriting from the view. This allows for simple modeling and decoupling.

The controllers implement event handlers to manage the run-time plugin addition and removal events. Plugins can use the Action (modelled after the command design pattern[5]) pattern to encapsulate GUI commands to avoid coupling.

3.3 The Model

The business model is represented as a set of classes and optionally a database backend used for persistence. In simple scenarios, a service layer to interact with

[1] Model View Controller
[2] Front controller pattern is mostly used in the Web application domain

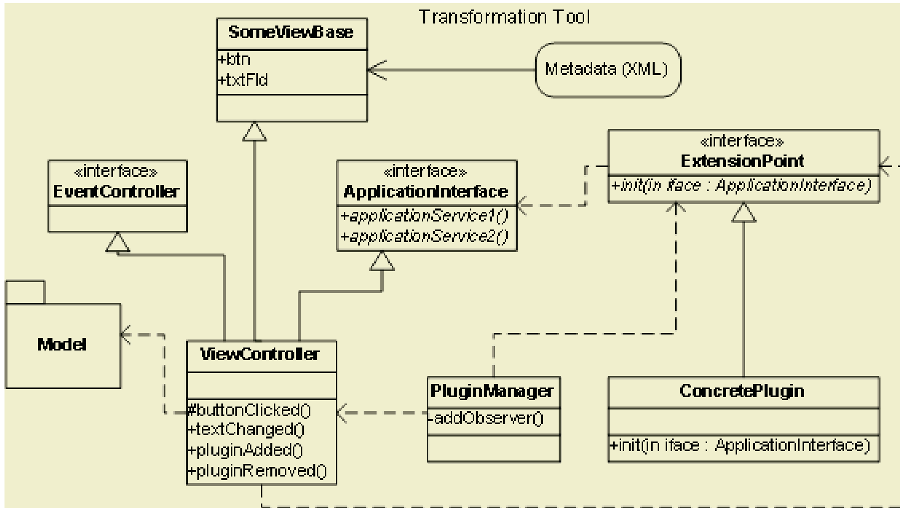

Fig. 1. Proposed Architecture

the database is enough to keep the architecture clean. If there is a complex business model, the Data Access Object pattern[7] is recommended to decouple the data backend handling from the business classes.

3.4 Supporting an Extensible Architecture

Plugins Technology. Plugins are a special type of components that can be optionally added to en existing system at runtime to extend its functionality (relationship between the plugin and the host application is stronger). The system doesn't know about the plugin and all communication happens through well defined interfaces.

Plugins are very popular among modern desktop applications. A good example is the variety of graphic filters and special effects available for Adobe Photoshop.

A plugin architecture avoids the huge monolithic applications as we see them now, allowing real-time deployment, easy maintenance and isolated component development.

Using Plugins in a MVC Architecture. Extension points are explicit holes in the applications where a plugin can plug its functionality a plugin can fill various extension points at the same time). The limits in what a specific plugin can do are set by the host application API and defining a good API for each extension depends a lot of the application context.

Adding plugin support in the application requires some extra code. This could vary from a simple dlopen C function to access a function in a shared library at run-time to a full featured framework to load components dynamically (e.g.:

MagicBeans [3] and JLense [9]). Modern languages like Java support dynamic loading of classes by name, a very handy feature for the needs of building a plugin infrastructure.

We model the plugin engine as a singleton using the observer pattern to generate events (pluginAdded and pluginRemoved) to subscribed controllers. The plugin event is then handled in the controller that manages the view where the plugin might add graphical components.

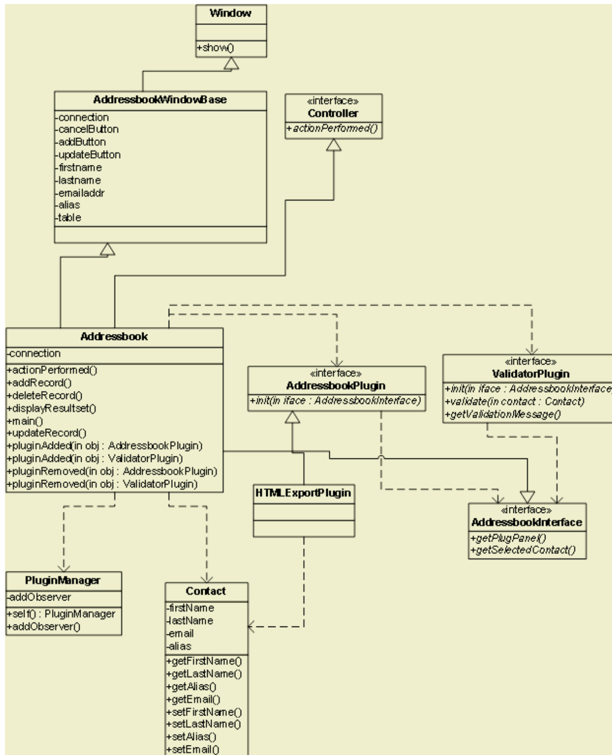

Fig. 2. Addressbook using the proposed architecture

4 An Example Application

As an example of our architecture, we took an addressbook application from a public forum in the Internet and proceed to refactor it to fit our proposed architecture. We set the goal of adding two new features to the application: exporting contacts to HTML and adding arbitrary constraints to new contacts in the entry form.

The original application was coded in Java and used a MySQL database as the backend. The user interface was done using the Swing API[3]. There is a single

[3] Application Programming Interface

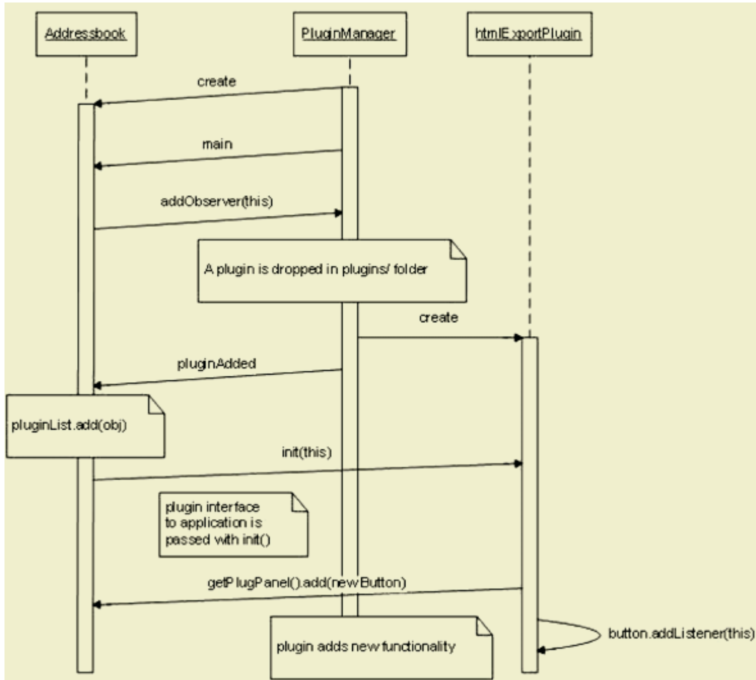

Fig. 3. Sequence diagram of interaction with the plugin manager when a new plugin is found

class which acts as a controller (handling events) and as a view (inheriting from a GUI window), a perfect example of no view/logic separation. Any tool used to assist user interface creation would need to parse the source. There is no model and the business logic is done in the same layer too.

The first step of the refactoring then, was to move all the user interface to a new class called AddresbookWindowBase and keep the existing class as the controller part of the architecture inheriting from the user interface. The new clean controller registers listeners for each user interface controls for each event it wants to handle.

For plugins to communicate with the application, an interface is needed. The interface should provide all the services needed by a plugin in the context. Two services were identified: retrieving contacts and adding actions to the user interface. We created a basic AddressbookInterface which offers a service to retrieve a panel where the plugin can add new actions. This application interface is passed to plugins during loading (Figure 3).

We chose MagicBeans [3] plugin engine because its simple design. Other plugin engine approaches could require wrapping the manager with our model.

The extension points were defined considering the context of the applications. The Validator extension encapsulates a generic algorithm with a contact as input and a boolean as output. Access to the addressbook services are done trough the Application interface see Figure 2).

5 Related Work

Smart Client Model [6] is an architecture proposed by Microsoft to develop thin rich clients. It is tied to the Windows platform and it does not suggest an architectural solution for extensibility.

MagicBeans [3] is a plugin engine developed by Robert Chatley. We use it in our example as the plugin engine.

Eclipse [4] is the most prominent example of a sole plugin based application. As it uses a central plugin registry and relies on XML configuration files, it is too complex and heavyweight for simple applications.

6 Conclusions and Future Work

We have shown here a RAD friendly simple architecture based on the MVC design pattern that allows for easy extensibility of rich clients using a plugin based approach. This gives us many additional benefits:

- New functionality can be quickly added without application redesign .
- Problems are isolated easily (because plugins are separate modules).
- The end user can customize a system without access to the source code.
- Third party developers can add value to the application.

We have built a simple example refactoring an application built using a common bad design to illustrate the level of extensibility that is possible switching to this model. Future work will be focused in generating complete frameworks for various architectures and languages using this concept. Currently we have Java and Qt proof of concepts.

References

1. Xul, the xml user interface language. http://www.mozilla.org/projects/xul/.
2. Robert Chatley, Susan Eisenbach, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Predictable dynamic plugin systems. In *Proceedings of FASE'04*, 2004.
3. Robert Chatley, Susan Eisenbach, and Jeff Magee. Magicbeans: a platform for deploying plugin components. In *Proceedings of CD'04*, 2004.
4. Eclipse Foundation. Eclipse technical overview. Technical report, Object Technology International, Inc., 2001. http://www.eclipse.org/whitepapers/eclipseoverview.pdf.
5. Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.
6. Microsoft. Smart client application model. http://msdn.microsoft.com/netframework/programming/winforms/smartclient.aspx.
7. Core J2EE Patterns: Best Practices and Design Strategies. Deepak Alur and Dan Malks and John Crupi. Prentice Hall PTR, 2001.
8. Susan Eisenbach Robert Chatley and Jeff Magee. Modelling a framework for plugins. In *Proceedings of the SAVCBS'03 workshop at ESEC/FSE '03*, 2003.
9. Ted Stockwell. Jlense application framework. http://jlense.sourceforge.net, 2001.
10. Trolltech. Ui, the qt toolkit user interface language. http://www.trolltech.com.