Reconciling the 3-layer Architectural Style with the Eclipse Plug-in-based Architecture

David Ameller  
Universitat Politècnica de Catalunya  
Barcelona, Spain  
dameller@essi.upc.edu

Oriol Collell  
Universitat Politècnica de Catalunya  
Barcelona, Spain  
ocollell@essi.upc.edu

Xavier Franch  
Universitat Politècnica de Catalunya  
Barcelona, Spain  
franch@essi.upc.edu

ABSTRACT
Software architecture construction is the result of a complex decision-making process, in which competing alternatives need to be compared. For example, deciding between a web-based application or a plug-in-based application has a significant impact on the architecture, therefore in order to make the right choice all possible tradeoffs between them must be considered. Decisions need to be made in all architectural views, from the logical view in which architectural styles are chosen, to the development view in which types of modules are decided, to the deployment view where physical allocation is determined. In this paper we analyze the interactions between a 3-layer architecture at the logical view, and a plug-in-based development view implemented in Eclipse, focusing on the difficulties we overcome in a research project in order to make it work.

Categories and Subject Descriptors  
D.2.10 [Software Engineering]: Design; D.2.11 [Software Engineering]: Software Architectures—Patterns

General Terms  
Design

Keywords  
Plug-in development, 3-Layer, Architectural Knowledge

1. INTRODUCTION
Architectural styles [6] determine the set of principles that rule the final form that a software architecture takes. If we refer to the 4+1 architectural framework [5], architectural styles apply to one architectural view, the logical view. The resultant logical architecture must then be mapped into the development view using architectural elements such as: package, component, and plug-in; which at their turn are allocated to nodes, servers, etc. in the deployment view.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Each architectural view requires architectural decisions to be made, e.g., do we use Domain Model or Transaction Script (logical view) ?, do we map this layer into one package or two (development view)?, do we deploy together in the same node these two packages or not (deployment view)? But more fundamentally, architectural decisions are not independent at each layer: since all these views are referring to the same system, any decision made at one architectural view may constraint or collide with other views. For instance, when two classes are mapped into the same package, their deployment is almost forced to be in the same node. Therefore, exploring and understanding these interactions among views may help in clarifying architectural decision-making.

This paper addresses one particular instance of this general problem that we faced in one research project, a tool for managing and reusing architectural knowledge. We decided to adopt one of the most widespread architectural styles, 3-layer [4], for the logical view, and then map it into a plug-in-based development view using the Eclipse framework. We will present the different alternatives that we considered and the difficulties we faced when the characteristics of Eclipse did not allow to implement directly the designed solutions. As a main result, we show a general solution for mapping a 3-layer architecture using Eclipse plug-ins.

2. BACKGROUND: PLUG-IN TRADEOFFS
As stated above, each architectural view requires some decisions to be made. From the software architect’s viewpoint, it must be ensured that the elicited requirements are fulfilled by the architecture. For example, a requirement such as “the system shall be able to run using different data base technologies with little adaptation effort” will make portability a priority in the architecture decision-making process and would be a clear indicator of the adequacy of a 3-layer architectural style in the logical view. Of course, it is common to have requirements that cannot be easily satisfied at the same time or even that are impossible to reconcile. In these cases, software architects should consider all the tradeoffs to make a decision. In this section we present the tradeoffs that were considered with respect to plug-in-based solutions when we chose the development view of our 3-layer logical architecture. We consider three different perspectives: the user, who installs the system in an execution environment and runs it; the developer, a general term representing software architectures, programmers, etc., responsible to build and deploy the system; the community, representing the set of potential users that are running systems similar or with a logical relation to the one of interest (see Table 1).
On the one hand, plug-in-based development shows some benefits. From the user’s perspective, usability is the principal advantage, it implies that the user does not need to learn to use a brand new environment for the system functionality. From the developer’s perspective, reusability is the main benefit, it implies that the developer does not need to design a whole new application, instead he can just focus on the added functionality, reducing thus the time and cost of the development especially in the case in which the developers are already familiar with the plug-in framework. From the community’s perspective, interoperability between plug-ins is the main objective to achieve. In many research communities, all tools are developed using the same framework.

On the other side, plug-in-based development suffers from some drawbacks. From the user’s perspective, compatibility issues among different plug-ins hamper installability (when first installing the system) and reliability (since the system may stop delivering the promised functionality due to some pernicious interactions with new, incompatible plug-ins). From the developer’s perspective, the adequacy of the chosen framework is very important, since plug-in development is restricted to the possibilities offered by that framework and it may be the case that it does not support adequately all the necessary functionality and/or technologies. From the community’s perspective, trends can be also a drawback due to their changeability nature. This changeability means that a plug-in becomes obsolete in a short time while a standalone application would have a longer lifespan.

### 3. ARCHITECH, AN ARCHITECTURAL KNOWLEDGE MANAGEMENT TOOL

In this section we motivate the main goal of our paper with a software development scenario we are currently facing, namely the development of ArchiTech, a system for supporting software architects in their decision-making processes in the context of Model-Driven Development (MDD) [2]. A detailed presentation of the system may be found at its site\(^1\), where the current version is available for download.

ArchiTech integrates two major subsystems: an architectural knowledge (AK) manager and an architectural assistant in the decision-making process. In our first stage of development, we are building the AK manager. It basically consists of a group of interrelated CRUD (Create-Read-Update-Delete) use cases applied to the concepts defined in an AK ontology. This AK will be used by the decision-making assistant that, using the answers provided by the software architect to some selected questions, will determine the non-functional requirements of the system under construction and then will use them to suggest to the software architect relevant architectural decisions. More details about our vision can be found at [1].

When we started our project, we implemented a proof of concept with a throw-away prototype implemented on top of a tool called AndroMDA\(^2\). As proof of concept, the experience was satisfactory since it demonstrated the feasibility of our approach. But for the next stage, we discarded the prototype mainly because of two reasons:

- In the MDD community, AndroMDA did not have a dominant position.
- The AK was hard-coded, making it difficult to maintain or customize to different needs (e.g., different domains or projects).

Concerning the first point, we may observe that most current MDD tools are deployed as Eclipse plug-ins. Eclipse is a general software development platform based in plug-ins [3], i.e. plug-ins are the basis of Eclipse, not just an extension method. Eclipse embraces many projects that deal with different topics, e.g., Eclipse Web Tools Platform Project, Eclipse Runtime Project, Eclipse Modeling Project, etc. In fact, all these projects are a set of interrelated plug-ins. We may say that in order to present a solution to the MDD community, the use of Eclipse seems currently a must: first, there is a large community that gives support for the development of tools; second, it provides a well-known user interface that is widely used (this means that the resulting plug-ins are easy to use by the users familiar with Eclipse); third, integration with other existing tools (e.g., code generators) becomes much easier. This community acceptance was mentioned in Section 2 as one of the strongest points supporting plug-in-based development, and has been the main driver for deciding the adoption of this plug-in-based technology, Eclipse, for our solution, even considering some recognized drawbacks, e.g. it has a large learning curve because it provides several APIs and lots of configuration files.

For the second point, in order to provide management facilities, the AK must be persistent, i.e. stored in a durable support. Due to the need of persistence, together with the need of providing a user-interface to dialogue with the user and the existence of a business logic (the CRUD use cases), we decided to adopt the classical 3-layer architectural style. This style also supports other stated requirements, e.g. usability benefits of the fact of having a dedicated presentation layer that may evolve to adapt to enhanced presentation technologies when they emerge. Therefore, we finally end up with the situation of developing a 3-layer architecture using a particular plug-in-based framework, Eclipse.

### 4. 3-LAYER ARCHITECTURE USING PLUG-INS

Two extreme positions are possible on the development of a 3-layer architecture using plug-ins. The most intuitive one is to have a single plug-in that contains three packages, one for each layer (see Figure 1-a). The second one is to separate each layer in an independent plug-in (see Figure 1-b). Using the first alternative we have the usual advantages of the 3-layer style (e.g., better reliability, maintainability and portability), the principal drawback being that this alternative is not very reusable since every change requires developing a new version of the plug-in.

Using the second alternative it is possible to take the most of plug-in based solutions. First of all, we can adapt our solution to technological changes in an easier way. For example,

<table>
<thead>
<tr>
<th>Table 1: Tradeoffs of using plug-ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>User’s perspective</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>Developer’s perspective</td>
</tr>
<tr>
<td>Community’s perspective</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
IMPOSED BY TECHNOLOGIES

5. ARCHITECTURAL LIMITATIONS

As said in Section 3, we decided that ArchiTech must support a persistence mechanism to store AK in order to make it easily customizable by the user. Another requirement that we wanted to satisfy was to have an easy installation, this is why we provide ArchiTech with an embedded DBMS. The tradeoff of this decision is that it limits cooperative work since every user will work with his own data. As a mitigation measure we decided to apply the Data-mapper design pattern. With this measure, apart from other benefits from this pattern, we can exchange the embedded DBMS just by modifying a configuration file.

After some research we decided to use the following technologies in ArchiTech: Eclipse framework\(^3\) as the supporting framework for our plug-in due to community dominance. JFace\(^4\) as the technology for the presentation layer. In fact, JFace is mandatory in Eclipse plug-in development. Hibernate\(^5\) as the technology used to implement the Data-mapper.

\(^3\)http://www.eclipse.org/pde
\(^4\)http://wiki.eclipse.org/JFace
\(^5\)http://www.hibernate.org

\(H_2\)\(^6\) as the embedded DBMS.

At this point, we spent some time developing prototypes to test the feasibility of the architecture. During this time we discovered two important technological limitations:

- Eclipse plug-ins that need extra libraries should be split in two plug-ins, one of them to hold the libraries and the other one to provide the functionality. This limitation affected the persistence layer where we use Hibernate as a Data-mapper.
- JFace, being an integrated part of the Eclipse platform, did not have the previous problem. But it presented another one. JFace needs access to the domain classes, so in order to have the presentation and the domain layer in separated plug-ins we must provide public access to the domain classes. Of course, this is not a good programming practice.

The next subsections report these problems in more detail.

5.1 The Libraries Problem

During the development of the CRUD part of ArchiTech, we have found that the Hibernate technology presents incompatibility issues with the Eclipse plug-in technology, in fact, at a first development stage, Hibernate did not work at all, it seemed that the plug-in was completely ignoring it, the calls to the Hibernate API were not producing any result, and they were even not producing exceptions.

After some research through some specialized sites we managed to find out what was happening. The problem was related to a quite well-known issue in the way in which the OSGi framework (which specifies how to deal with plug-ins) manages the loading of classes. Currently, the majority of libraries use a class loading mechanism based on a context classloader, which basically knows how to load only certain classes depending on the context of execution. Each plug-in means a different context, so that it only knows how to load classes within it. The point is that there is a need for switching this context in certain moments and this is not defined in the OSGi framework. The problem arises when having to invoke code from external libraries, that is, that are not part of the Eclipse application itself, since if not specified otherwise, the context classloader is set to the application classloader at the beginning of an Eclipse execution. Therefore, code from external libraries will not be reachable without this context switch, which will never occur. We considered two solutions to this problem:

- Define programatically the points in which context switches should take place, that is, manually change the context classloader before invoking code from an external library.
- Split the libraries into another plug-in and use the buddy classloading mechanism defined by Equinox (the implementation of the OSGi framework).

The buddy classloading mechanism allows a plug-in to request for help to another plug-in when trying to load a class. Basically it sets a policy that tells what to do (which plug-ins to request for help) when failing to load a class.

\(H_2\)\(^6\)http://www.h2database.com
This deals with the problem described above since when requesting for help to another plug-in, the context will be automatically switched. We opted for this solution.

5.2 JFace and Domain Layer Coupled

The problem with JFace is that the user interface has to be always synchronized with the domain classes, which implies that the objects shown in the views should be references to the “real” objects of the domain layer. Otherwise the user interface could show incorrect information. Despite of this fact, when retrieving different objects from the database, references are not shared between them but different instances of the same object are brought to memory. As a result of this, we had to implement a singleton class that represents the domain classes to maintain the reference to the objects which are being used in the presentation layer. Moreover, when objects are retrieved from the database the references to the equivalent objects maintained by the domain layer need to be updated. The drawbacks of this solution are: replicated information and less efficiency in data management. However, since ArchiTech does not manage huge amounts of data, efficiency loss is not noticeable.

5.3 Architectural Solution Used in ArchiTech

As result of the detected technological limitations, our theoretical architecture for ArchiTech had to be refactored into another one that deals with these limitations. The resulting architectural solution is shown in Figure 2.

We had to split the persistence plug-in to solve the libraries problem, and we took the option to separate the libraries for Hibernate and the libraries for the H2 support, because making this extra separation facilitates the integration of our solution with other DBMSs. Notice that these two new plug-ins only contain the libraries needed to work with the DBMS in one case, and the libraries provided by Hibernate in the other, they do not contain any logic.

The second change was to put the domain layer together with the persistence layer, the reason being the way Hibernate works that makes hard to maintain the domain layer totally independent from the persistence layer. Since we do not expect to change this technology in a near future, we preferred to facilitate the development task in this part of the plug-in, but as told before the coupling between JFace and the domain layer has been isolated using an extra work in the singleton class. The reason for this decision is that we plan to convert this plug-in in a front-end of an external service (see Section 6), so we prefer to force this separation for an easy transition.

6. CONCLUSIONS AND FUTURE WORK

This paper has several outcomes. From a very practical point of view, it provides a reusable Eclipse implementation of the 3-layer architecture that overcomes some identified obstacles. From a more design-oriented perspective, we have explored the mapping of one particular architectural style to a plug-in-based solution, belonging both to different architectural views. We believe that this line of research is fundamental in order to gain knowledge about the AK discipline, which is a cornerstone in modern software architecture decision-making. From a lessons-learned perspective, we have reported that the usual gap among theory and practice is also present when considering plug-in development in a particular framework and we have demonstrated in a particular project how it may be bridged.

Concerning future work, we aim at transforming the presentation plug-in into an external service that provides the functionality. Not only useful for our particular project, it will be also a particular case of combining two dominant development styles (plug-ins and services) for implementing the 3-layer architectural style. Also, we want to incorporate the architectural knowledge gained in the development into the ArchiTech tool.

7. ACKNOWLEDGMENTS

This work has been partially supported by the Spanish MICINN project TIN2010-19130-C02-01.

8. REFERENCES