The Three-Layer Architectural Pattern Applied to Plug-in-based Architectures: the Eclipse Case

David Ameller¹, Oriol Collell¹, Xavier Franch¹

¹Universitat Politècnica de Catalunya, Barcelona, Spain.

SUMMARY

The process of designing a software architecture using different kinds of components is often challenging. Different designs support some quality attributes whilst damaging others, therefore trade-off analysis is needed to make informed decisions. Moreover, analysis made in theory need to be complemented with observations in practice, especially when using a particular set of technologies to implement the system. In this paper we present a particular instance of this problem. We study how the Three-Layer architectural pattern may be developed using plug-ins. We compare two extreme alternatives according to several representative scenarios and their impact in some quality attributes. Then, we apply this theoretical knowledge to a case study, the implementation of a plug-in-based tool for managing architectural knowledge using Eclipse. We report some unexpected difficulties found that forced us to adapt the theoretical solution into an operative architecture. Copyright © 2010 John Wiley & Sons, Ltd.

KEY WORDS: Plug-in-based development, Three-Layer Architectural Pattern, Eclipse, Software Architecture, Architectural Knowledge

1. INTRODUCTION

Software architectures are the result of applying an architectural pattern to a certain problem instance. The concept of architectural pattern become widespread from Shaw and Garlan’s work at the mid-nineties [1] under the name of architectural style. Ever since, lots of pattern catalogue proposals have emerged, from Buschmann et al. catalogue [2] to more recent domain-oriented proposals (e.g., Erl’s catalogue for service-oriented systems [3]). In most of these proposals, the term “style” changed into “pattern”, which was already in use in the community for design patterns [4]. According to Bass et al. [5], both terms may be considered equivalent.

Paraphrasing the old pattern definition by Alexander [6], Buschmann et al. define an architecture pattern as “a description of a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution” [2]. Concerning the form an architecture pattern may take, Bass et al. enumerate as main parts (examples are ours) [5]:

- A set of element types (e.g., packages, components, nodes).
- A topological layout of the elements indicating their inter-relationships (e.g., distribution into nodes and their communication paths).
- A set of semantic constraints (e.g., a Persistence layer cannot invoke presentation services).

*Correspondence to: E-mail: {dameller, ocollell, franch}@essi.upc.edu

Contract/grant sponsor: This work has been partially supported by the Spanish MICINN project TIN2010-19130-C02-02
A set of interaction mechanisms that determine how the elements coordinate through the allowed topology (e.g., remote procedure call for service-oriented systems).

A cornerstone concept for software architectures is that of architectural view. An architectural view is a representation of the whole system from the perspective of a related set of concerns [7]. These views help in comprehending large and complex software architectures by focusing at once just dealing with a particular concern. Several views have been proposed so far. One of the most popular approaches is the 4+1 architectural framework proposed by Kruchten [8] consisting of a logical view, a development view, a deployment view, a process view, and a use case view that coordinates the first four views.

Most of the architectural and design patterns proposed in the literature apply to Kruchten’s logical view. Then, normally, the resultant logical architecture is mapped into the development view using architectural elements such as: package, component, plug-in, etc. Lago et al. [9] said that “each view is chosen to frame certain stakeholder concerns and depict how the system will address those concerns”, in most cases these concerns are related to quality attributes. For instance, maintainability may determine to a great extent how to organize logical elements in the development view, while availability and security may suggest the inclusion of components such as load balancers and firewalls in the deployment view.

Our work addresses one particular instance of this general situation that emerged when building a tool for managing and reusing architectural knowledge [10]. For the logical view, we decided to adopt one of the most widespread architectural patterns, the Layers architectural pattern [2] in the particular case of Three-Layer architectural pattern, and then map it into a plug-in-based development view using the Eclipse framework. We discovered that little is said in the literature about how to perform this transition of views in a systematic manner, especially because plug-in-based architectures have not been much analyzed. Therefore, the objective of this paper is to share our experience in the development of a plug-in based tool for Eclipse using a three-layer logical architecture. To reach this objective we had gone through an analytic process that include:

- Determine the most relevant software qualities that are supported or damaged by a plug-in-based solution. We have concluded that usability, productivity and interoperability are supported by plug-ins whilst reliability, installability, creativity and lifespan are damaged (see Section 2 for details).
- Determine the alternatives for developing a three-layer logical architecture with plug-ins can be considered and how do they compare to each other. In this case, we have made a trade-off analysis of two extreme alternatives: the three layers in one plug-in and each layer in one plug-in (see Section 3 for details).
- Find out the gap between theory and practice when considering not a general plug-in-based solution but a concrete technological solution (in our case, Eclipse). In our technological solution we have identified two issues in the Eclipse framework that affected our initial architecture design and we have proposed solutions for both issues that may be applied in similar cases (see Section 5 for details).

The rest of the paper is organized as follows. In Section 2 we introduce the fundamental concepts used in this paper, in Section 3 we discuss the possible alternatives to combine the Three-Layer architectural pattern and plug-ins, in Section 4 we introduce the case of study that motivated this work, in Section 5 we discuss the problems and solutions found for this case study, and finally, in Section 6 we draw the conclusions and future work.

2. BACKGROUND

In this section we introduce the Three-Layer architectural pattern, the plug-in concept as we understand it, and the Eclipse framework. In the Section 3 we will see how we propose to design a plug-in based solution using the Three-Layer architectural pattern, and in Section 5 we will see the
particular problems we found when integrating our architectural solution in a concrete framework, in this case Eclipse.

2.1. Three-Layer Architectural Pattern

The Three-Layer architectural pattern is a particular case of the generic Layered architectural pattern. It should not be confounded with the Multi-tier architectural pattern (also known as N-tier). While the Layered architectural pattern refers to the logical separation of the architecture (logical view), the Multi-tier architectural pattern does for the physical arrangement of the architecture (deployment view).

The Layered architectural pattern conforms to a very simple principle: the layer N should only use the services or interfaces provided by the layer N-1 (see Figure 1-a). The typical example for this pattern is the OSI Reference Model [11] that is used in the Internet communication. The particular variant studied in this paper is the Three-Layer architectural pattern (see Figure 1-b), which is widely used in applications such as transactional information systems. The three layers are: Presentation (also known as User Interface), that handles the interaction with the user; Domain (also known as Business), that provides the functionalities of the application; and Persistence (also known as Data), that provides access to the physical storage. A full description of each layer can be found in [12].

The benefits and drawbacks of the Three-Layer architectural pattern are well-known and are the consequence of applying the principle described above. The separation into layers improves the reliability, reusability and portability of the application, while efficiency is the most damaged quality attribute because most of the calls have to go through all the layers. We refer to [2] for more details.

2.2. Fundamentals of Plug-ins

A plug-in is an application that is developed to be integrated as part of a larger application or framework. Contrary to regular software modules, plug-ins do not require additional software compilation, and in most cases it can be added at runtime. To make this situation feasible, the plug-in should be developed in conformance with the API provided by the larger application. Mayer et al. [13] presented a design pattern to fulfill this objective. Plug-ins are not a new idea, plug-ins share many characteristics with other design ideas such as loadable kernel modules, which are several decades old. In the context of this paper we refer to plug-ins as they are nowadays understood: software component that adds or customizes a functionality or capability in a larger software application that is already designed to be extended by plug-ins (e.g., a web browser plug-in to manage the downloaded files).
Plug-in-based development shows some tradeoffs that need to be considered to assess its adequacy in a particular scenario. To determine the most relevant software qualities that are supported or damaged by a plug-in-based solution we present here the most relevant strengths and drawbacks. We consider three different perspectives coming from three different types of stakeholders that in our experience had been crucial to make the architectural decisions (see Table I):

- The user, who installs the system in an execution environment and runs it;
- The developer, a general term representing software architects, programmers, etc., responsible to build and deploy the system;
- The community, representing the set of potential users that are running similar systems or with a logical relation to the one of interest.

On the one hand side, 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, productivity is the main benefit, because the developer does not need to design a whole new application, instead s/he can just focus on the added functionality by reusing the functionality provided by the framework and other plug-ins, reducing thus the time and cost of the development (the time reduction is especially significant in the case that the developer is already familiar with the framework). From the community’s perspective, interoperability between plug-ins is the main objective to achieve. For example, in many research communities all tools are developed using the same framework.

On the other hand 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 or fads can be a drawback, if a community decides to change the framework totally or partially (which is quite common with emergent technologies) the plug-in will become obsolete in a short time while a standalone application would have had a longer lifespan.

### 2.3. The Eclipse Framework

Eclipse is a general purpose software development platform based on plug-ins [14], 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. A good starting point to understand the development of plug-ins for Eclipse is Gamma and Beck’s book [15].

In addition to the benefits mentioned above for every plug-in-based system, Eclipse offers some additional advantages: first, there is a large community that gives support to 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 those users familiar with Eclipse). But it also has some drawbacks, e.g., it has a steep learning curve because it provides several APIs and lots of configuration files, and the graph of dependencies between its base plug-ins is complex and sometimes confusing.
To determine the alternatives for developing a three-layer logical architecture with plug-ins, we have to observe which are all the possible ways of grouping the logical components into the deployment components and compare those that make sense and are relevant. In our particular case, layers (the three of them) are to be deployed using plug-ins.

Within all the possibilities, a good starting point is to compare the two extreme alternatives since both make sense and have relevancy. The most intuitive one is to have a single plug-in that contains three packages, one for each layer (see Figure 2-a, three-layered plug-in). The second one is to separate each layer in an independent plug-in (see Figure 2-b, one plug-in per layer). The intermediate alternatives (e.g., the two top layers in one plug-in and the bottom layer in another plug-in) are not described because the tradeoff analysis for these alternatives does not provide any remarkable (or unexpected) result. Table II summarizes the benefits and drawbacks of using three-layered plug-in or using one plug-in per layer.

Initially, to compare these two extreme alternatives, we have analyzed their suitability when facing different relevant scenarios, which are described below. These four scenarios are, according to our experience, the most recurring situations that may occur when using plug-ins. Table III summarizes the required quality aspects for each scenario.

**Scenario 1: Change of some technology.** At some point of the development it may be necessary to change some technology, e.g., the Database Management System (DBMS) may have to be changed due to a company policy, or it may be mandatory to upgrade some technology because some security flaw.

Given this scenario, having the whole application in a single plug-in (three-layered plug-in) means having to re-deploy it in its totality, which would be costly both for the developers who would have to deal with the whole source code and for the end-user who would have to
update the whole application. Moreover it would be error-prone since the whole system would be exposed to change, even though not all of its parts would be changed.

However, for example, in the case of DBMS change, having one plug-in per layer would overcome this situation since in this way all changes and updates would be reduced to the persistence plug-in (without affecting the rest of plug-ins) instead of a plug-in containing the whole application.

Scenario 2: Integration. There is no piece of software that is totally exempt of bugs, especially when it comes to complex applications. For this reason, applications have to be prepared, as far as possible, to bear with errors and to allow developers finding and solving them as fast as possible.

As shown by Prechelt and Beecher [16], working with plug-ins implies having to worry not only about your system and your functionality but also about the interaction with other systems. For example, an error in the database may show its effect in the user interface plug-in. In the case of having one plug-in per layer the developer does not know initially if the error is in the Presentation, Domain or Persistence layer plug-in while in the case of having a three-layered plug-in for the whole application it would reduce the search space to look for the error because he or she will not have to worry about the interaction between the plug-ins.

Scenario 3: Third-party change in the running environment. As stated by Dig in most of his works (e.g., [17]), APIs and frameworks are prone to change over time. Such changes can cause that plug-ins running upon them crash or require an adaptation to keep working properly. One typical case is the change of the signature of a method of the framework, e.g., adding or deleting a parameter, so that invocations to that method will no longer work.

In this scenario it is not clear which of the alternatives would work better since it depends on the type and dimension of the update. However, because of the same reasons stated in Scenario 2, for most of the framework changes the solution with a three-layered plug-in will work slightly better.

Scenario 4: Ability to switch between different ways of achieving a given functionality. Plug-ins give us the ability to be very flexible in terms of system changes or functionality adaptation since an application built upon a plug-in-based architecture achieves its functionality by means of those provided by the plug-ins that compose it. This means that when an alternative behavior is desired, only a subset of the plug-ins needs to be changed or adapted. Plug-in-based architectures have the ability to support scenarios such as having two Presentation layers, and let the user switch among them.

In this case, it is clear that the alternative of one plug-in per layer is highly more suitable than the three-layered plug-in. Since in the three-layered plug-in there is only one plug-in, an alternative behavior would mean having a whole new plug-in.

For example, by having one plug-in per layer, we could provide two versions of the plug-in that contains the Presentation layer: one that focus on accessibility for disabled people and one that focus on usability.

We obtained the Table IV from a tradeoff analysis between Table II and Table III (we also considered the comments made for each scenario). As result we can conclude that:

Table III. Quality aspects required by each scenario

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Required quality aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1</td>
<td>Changeability and adaptability</td>
</tr>
<tr>
<td>Scenario 2</td>
<td>Testability and maintainability</td>
</tr>
<tr>
<td>Scenario 3</td>
<td>Changeability and adaptability</td>
</tr>
<tr>
<td>Scenario 4</td>
<td>Portability and interchangeability</td>
</tr>
</tbody>
</table>

We obtained the Table IV from a tradeoff analysis between Table II and Table III (we also considered the comments made for each scenario). As result we can conclude that:
Table IV. Alternatives suitability per scenario

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Three-layered plug-in</th>
<th>One plug-in per layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Scenario 2</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Scenario 3</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scenario 4</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Using the three-layered plug-in alternative we have the usual advantages of the Three-Layer architectural pattern (remarkably, better reliability, maintainability and portability), the principal drawbacks being that this alternative is not very reusable since every change requires developing a new version of the plug-in.

- Using the one plug-in per layer 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 and also we can provide different solutions for the same functionality. As a negative effect, in this alternative reliability may be lower because the one plug-in per layer structure is more susceptible to have incompatibility issues derived from having different versions of plug-ins. To minimize this effect it is important to pay especial attention to plug-in interfaces design.

4. CASE OF STUDY: THE ARCHITECH SYSTEM

To illustrate this theoretical scenario, we present in this section ArchiTech, a system for supporting software architects in their decision-making processes in the context of Model-Driven Development (MDD) [18]. A detailed description of the system may be found at its site (www.upc.edu/gessi/architech/index.html), where the current version is available for download.

ArchiTech integrates two major subsystems: an architectural knowledge (AK) manager and an architectural decision-making assistant. We focus on the first one. The AK manager basically consists of a group of interrelated CRUD (Create-Read-Update-Delete) use cases applied to the concepts defined in an AK ontology [19]. 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 some 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 in [20].

When we started our project, we implemented a proof of concept with a throw-away prototype implemented on top of a tool called AndroMDA (www.andromda.org). 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: first, AndroMDA did not have a dominant position in the MDD community; and second, the AK was hard-coded, making it difficult to maintain or customize to different needs (e.g., different domains or projects).

Concerning the first reason, we may observe that most current MDD tools are deployed as Eclipse plug-ins. This is not a coincidence; Eclipse provides an open framework and a quite good community support which makes it a good candidate to many software research communities. We may say that in order to present a solution to the MDD community, the use of Eclipse seems currently a must if integration with other existing tools (e.g., code generators) is planned. The 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 its recognized drawbacks.

For the second reason, 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 graphical interface to dialogue with the user and the existence of a domain logic (the CRUD use cases), we decided to adopt the classical Three-Layer architectural pattern. This pattern also supports other stated requirements, e.g., usability benefits from 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 three-layer architecture using a particular plug-in-based framework, Eclipse.

At this point, we analyzed the two alternatives for organizing the plug-in-based architecture described in Section 3. The benefits of the second option (having a plug-in per layer) lead us to choose it, whilst being aware of the reliability-related drawback.

One of the reasons for choosing this alternative is because splitting the three layers in independent plug-ins provides more flexibility, which at the same time supports better the described scenarios. In consequence, it is not relevant to analyze and compare these two extreme alternatives with other intermediate ones (such as having the Domain and Persistence layers in a plug-in, and the Presentation layer in another plug-in) since none of them would be better (in terms of drawbacks and scenario suitability) than the chosen one.

Once the decision was made, it was necessary to implement the solution. In the next section we report the differences between the expected theory as explored above and the real practice.

5. DEALING WITH ARCHITECTURAL LIMITATIONS IMPOSED BY TECHNOLOGIES

The gap between theory and practice became relevant to our work when we started with the implementation of the plug-in and we began to find technological limitations.

As said in Section 4, we decided that ArchiTech must support a persistence mechanism to store AK in order to make it easily customizable by the user. As a result of this requirement, we decided to use some database for storing information. Since another requirement to satisfy was to have easy installation, we decided to provide ArchiTech with an embedded DBMS. The tradeoff of this decision is that it limits cooperative work since every user will work with his or her own data. As a mitigation measure we decided to apply the Data Mapper design pattern [12]. With this decision, apart from other benefits coming 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** as the supporting framework for our plug-in due to community dominance. [www.eclipse.org/pde](http://www.eclipse.org/pde)
- **JFace** as the technology for the Presentation layer because it is mandatory in Eclipse plug-in development. [wiki.eclipse.org/JFace](http://wiki.eclipse.org/JFace)
- **Hibernate** as the technology used to implement the Data Mapper pattern (it is widespread and we have successfully used it in the past). [www.hibernate.org](http://www.hibernate.org)
- **H2** as the embedded DBMS, since it is an easy to use DBMS that is also well accepted by the Java developers community. [www.h2database.com](http://www.h2database.com)

At this point, we spent some time developing prototypes to test the feasibility of the architecture. During this activity we discovered two important technological limitations that we describe in the next subsections.

5.1. The Libraries Problem

During the development of the CRUD (Create-Read-Update-Delete) use cases of ArchiTech, we 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 found out that the problem was related to the way in which the OSGi framework (which specifies how to deal with plug-ins, [www.osgi.org](http://www.osgi.org)) 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 has a different context, this means that it is
only able to load classes that belong to itself. In certain moments, there is a need to switch this context to be able to use external classes but this situation is not defined in the OSGi framework. The problem arises when the plug-in needs to invoke code from an external library that does not belong to the Eclipse framework. If it is not specified otherwise, the context classloader is set to the Eclipse context at the beginning of an Eclipse execution. Therefore, code from external libraries will not be reachable without a context switch.

We considered two solutions to this problem: First, identifying in the code the points in which context switches should take place, that is, manually change the context classloader before invoking code from an external library. Second, embedding the libraries into a different plug-in and use the Buddy Classloading mechanism defined by Equinox (the implementation of the OSGi framework used in Eclipse). Since the first solution has a very strong negative influence on maintainability we opted for the second solution.

The Buddy Classloading mechanism allows a plug-in to request for help to another plug-in when trying to load a class (see Figure 3). Basically it sets a policy that describes what to do (i.e., which plug-ins to request for help) when failing to load a class. This deals with the problem described above since the request for help to another plug-in will switch the context automatically.

The problem as described here is very specific to the Eclipse technology along with the OSGi framework. However, since the need of loading external libraries arises all similar frameworks, we can expect to find similar problems in other environments.

5.2. Coupling of JFace and the Domain Layer

JFace, being an integrated part of the Eclipse framework, is not affected by the previous problem but presents a different one. The adoption of JFace in the Presentation layer requires continuous synchronization between the user interface and the domain classes, which implies that the information shown in the user interface should be connected to the “real” objects of the Domain layer. Otherwise, the user interface may show inconsistent information. Despite of this fact, when retrieving objects from the database, in each subsequent call, Hibernate produce different instances of the same object.

To exemplify the issue we use a particular use case of ArchiTech, in which the user opens the window to manage “components” and selects one of them to edit its contents (see Figure 4, note that this is a simplification of the actual behavior, since the Presentation layer does not interact directly with the Data layer). The first step of the process happens during the initialization of the tool by retrieving all the components from the database and by placing objects in the views which are references to the retrieved components. Then, at any moment, the user may decide to edit a component called Layer. As a side effect of this action, the component needs to be retrieved again from the data base since it may be out-of-date, and this is precisely what produces the incorrect behavior, because now the user will be editing a component different from the one which is being shown in the user interface.
To avoid this behavior, we had to implement a class to represent the current state of the domain objects, that is, references to all the objects that are being used directly or indirectly by the views, so that when an object is retrieved from the database, the references can be updated to point to it. The drawbacks of this solution come from the replication of information: less maintainability and less efficiency in data management. However, since ArchiTech does not manage huge amounts of data, efficiency loss is not significant.

We found this issue within Eclipse and its technology used for building interfaces, however, the same problem may arise in other technologies that use the same techniques as JFace when dealing with model objects.

5.3. ArchiTech Final Architecture

As result of these detected technological limitations, the theoretical architecture initially designed for ArchiTech had to be refactored into another one that is able to cope with these limitations. The resulting architectural solution is shown in Figure 5.

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 shift to another DBMS if decided in the future. Notice that these plug-ins only contain the libraries provided by Hibernate in one case, and the libraries needed to work with the DBMS in the other, but they do not contain any specific functionality.

The second change was to put the Domain layer together with the Persistence layer, the reason being that the way Hibernate works 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 opted for facilitating the development, but, as told before, the coupling between JFace and the Domain layer has been isolated in a class that maintains the references to the domain objects. This decision is related to one of our future features, multiplatform support. Our intention is to transform the Domain and Persistence layers into services and keep the Presentation layer as a plug-in.

It is worth to remark that our solution is a particular instance of the mapping that fulfills the requirements for our project, and there may be other instances of the mapping that work better for other projects. This is an open question that we leave for further research.
6. CONCLUSIONS AND FUTURE WORK

In this paper we have analyzed the effects of mapping a three-layer logical architecture into an Eclipse-based plug-in solution. As a real example, we have finally applied both the theoretical and practical outcomes to develop a tool for architectural knowledge management. These outcomes can be summarized as follows:

- The most relevant software qualities that are supported by a plug-in-based solution are usability, productivity and interoperability, whilst it presents some drawbacks with respect to reliability, installability, creativity and lifespan.
- We have compared the alternatives for developing a three-layer logical architecture with plug-ins. We have formulated two extreme alternatives, one representing the three layers in one plug-in and the other representing each layer in one plug-in. We have compared the two alternatives with respect some quality attributes using four representative scenarios to drive the discussion.
- We have identified the gap between theory and practice when considering not a general plug-in-based solution but a concrete technological solution (in our case, Eclipse). We have confirmed that, as usually happens in the architecture design process, some aspects emerge when considering not just the theory, but a real case. We have identified two technological issues in the Eclipse framework that affected our initial architecture design and then we proposed solutions that may apply to similar cases.

Future work spreads into several directions. First, we want to define a Plug-in development architectural pattern and then reformulate our work as a transformation between the logical view and the development view. This pattern would apply to the development architectural view, which is a fundamental difference with the proposal made by Mayer et al. [13] which propose a Plug-in design pattern at the logical view. Second, we aim at developing more case studies for encountering more gaps between theory and practice in Eclipse and other frameworks that support plug-ins (e.g., .NET framework) and then provide a knowledge base to software developers. In this direction, even that our experience with other technologies is very limited, we believe that other technologies, such as those behind .NET framework, suffer from similar issues, as the ones described in Section 5, when they are integrated with database mapping technologies. The reason is that, in .NET, the presentation technology (Windows Presentation Foundation) uses the same techniques as JFace.
when dealing with domain model objects [21]. Last, we want to integrate this work into our architectural knowledge management line of research, by characterizing the elements that compose plug-ins solutions from an ontological point of view.

7. ACKNOWLEDGMENTS

We thank Cristina Gómez and Cristina Vicente for their comments on early versions of the paper. We also thank David Cerdán for the early implementations of the ArchiTech system.

REFERENCES