Adopting the Eclipse Communication Framework: The Case of eConference

Fabio Calefato[1], Mario Scalas[1]

[1] Università degli Studi di Bari, Dipartimento di Informatica, Bari, Italy
{calefato, scalas}@di.uniba.it

Abstract. eConference is a text-based conferencing tool, built upon the Eclipse Rich Client Platform (RCP), which has evolved over four versions since its first release in 2002. In the latest version, our tool has reached communication protocol independency thanks to the adoption of the Eclipse Communication Framework (ECF). This paper describes how the development of this new release of eConference has unexpectedly evolved due to the underestimated impact of adopting ECF as a network layer. The problems encountered have been tackled by developing an aspect-based framework, which promises to be applicable to other distributed applications built upon Eclipse RCP and with an emphasis on communication. Future improvements to both our tool and framework are also discussed.

Keywords: eConference over ECF, Eclipse, framework, design patterns, aspect oriented programming, dependency injection, objects weaving.

1 Introduction

eConference is a text-based conferencing tool that supports distributed teams in need for synchronous communication and structured discussion services. Other than offering communication services, it integrates an agenda and minutes editor, plus other control and coordination features like hand raising. Our tool has been successfully used at the University of Bari and at the University of Victoria, both as a training aid and in laboratory experimentations conducted to validate our hypothesis about the adequateness of synchronous lean communication media for distributed requirements workshops [3].

To date, three stable versions of eConference have been released. Since the third version our tool is a pure-plugin system, built on top of the Eclipse Rich Client Platform (RCP). Currently, a fourth version is being developed and it is near to completion. In [3] we have already reported on our work in progress to implement communication protocol independency thanks to the adoption of the Eclipse Communication Framework (ECF) [6]. ECF provides RCP-based applications with an abstract communication layer and some of the most common collaborative features, either in terms of API or visual components, such as whiteboarding and file transfer. Thus, ECF was chosen for replacing the communication layer of eConference and some domain-specific parts of our tool, with the promise of relieving us from the

burden of maintaining an abstract network layer to be communication-protocol independent and cope with future evolutions.

In this paper we describe how the development of this new release of eConference has unexpectedly evolved due to the underestimated impact of adopting ECF as a network layer. The problems encountered have been tackled by developing an aspect-based framework, which promises to be applicable to other distributed applications built upon Eclipse RCP and with an emphasis on communication, thus easing the adoption of ECF.

The remainder of this paper is organized as follows. Section 2 will outline the design of the previous version in order to summarize the design decisions made and lessons learned. Section 3 will deal with the architectural problems and solutions encountered when integrating ECF in the application, discussing how ECF has been integrated and Dependency Injection [21] has been implemented. Section 4 will outline the conclusions and future works.

2 The Motivation and Cost of Adopting ECF

eConference 3 was built upon Eclipse RCP with a strong focus on extensibility and flexibility. Eclipse RCP is a pure-plugin platform and, hence, fully extensible by architectural design since it is based on Equinox, the Eclipse implementation of the OSGi specs, which define a Java-based dynamic component model, making it possible to write new plugins for missing functions [5]. Besides achieving functional modularity through RCP, we also added network protocol independency by developing an abstract communication network infrastructure on our own, so that we would be able to add support to several protocols in the future, other than XMPP [27], without a severe impact on the code base.

However, eConference 3 suffered from some architectural drawbacks, namely 1) a low-level, abstract network layer too expensive to maintain on our own; 2) a burdensome publish/subscribe subsystem, in which every bundle implemented the Observer pattern [13] without taking advantage of the Event Admin Service, i.e., the Eclipse/OSGi internal mechanism for appropriately handling events dispatching in a dynamic pure-plugin environment [15], [22]. While the second drawback was due to our initial inexperience with the development of the Eclipse RCP platform, the first one was instead imputable to a design choice of ours.

All the domain-specific features were built on the internal API of the abstract network layer. As a side effect, the low-level network layer had to be maintained in addition to the application itself, although our main intention was to focus the effort on the development of domain components. Upon completing the development of eConference 3 we realized that we were not able to sustain the cost of maintaining network layer abstraction.

The Eclipse foundation hosts an internal project meant just to address this problem for any RCP-based application. In fact, ECF provides rich-client applications with an abstract communication layer that can replace the whole network infrastructure layer. The goal of this project is to introduce within the Eclipse platform typical collaborative services and features (e.g., presence, IM, file transfer, white-boarding),

bundled as set of plugins that can be reused by any RCP-based applications. Such components include core API definitions, graphical user interface widgets, and interfaces for specific network protocols. The ECF core includes an extensible framework, the Shared Object API, which provides a way for sharing data at application-level, without having to bother with protocol-specific details that are transparently handled by the underlying framework. ECF, in fact, already provides the implementations (called providers) of abstract interfaces for the most used communication protocols (e.g., MSN, Yahoo, and Skype), although, being already an IETF standard, XMPP is the most stable and advanced provider.

Hence, we decided to develop the fourth version of eConference using ECF to replace the communication layer and relieve us from the burden of maintaining an abstract network layer to cope with future evolutions. Initially, we thought to have two alternative solutions, i.e., either porting the earlier version to ECF or developing the new version from scratch. The preferred alternative was the porting because it was alleged to be faster and it would have allowed to retain a larger portion of the codebase we had already developed. Instead, a porting of eConference to ECF turned out not to be feasible for the proper adoption of ECF [4].

One of the aspects we overlooked when we decided to adopt ECF was that it is a "vertical" communication middleware, since it does not come only with a set of network services. Instead, ECF already provides several out-of-the-box graphical components, along with the respective services (e.g., contacts roster, chat editors, and user account management), which can be embedded in any Eclipse-based application. Consequently, between eConference 3 and ECF a large overlapping was found among the basic communication features they both provided, in terms of API's, visual components, and model objects. Thus, due to the larger than expected impact of adopting ECF, the efforts of porting and redeveloping were almost equivalent, since only a limited portion of GUI could be retained. Hence, we decided to rewrite the application from scratch, building upon the ECF API and services, only reusing the existing GUI code where possible. The cost of rewriting was partially paid back by employing a standard network technology, maintained separately from our tool.

3   eConference over ECF: Architectural Design

In this section we will discuss the architecture of eConference over ECF (ver. 4), which is depicted in Figure 1. eConference 4 is built on top of several other plugins coming from the Eclipse ecosystem, such as RCP and ECF, and other third party sources, such as Guice/Peaberry and AspectJ.

RCP/SWT, OSGi and the Java platform are mandatory parts of any Eclipse-based application. At least Java 5.0 is required since we do heavy usage of annotations within our codebase. The ECF components currently used are the core and presence sub-systems, the object-sharing infrastructure, and the XMPP provider, which we employ as the default communication protocol.
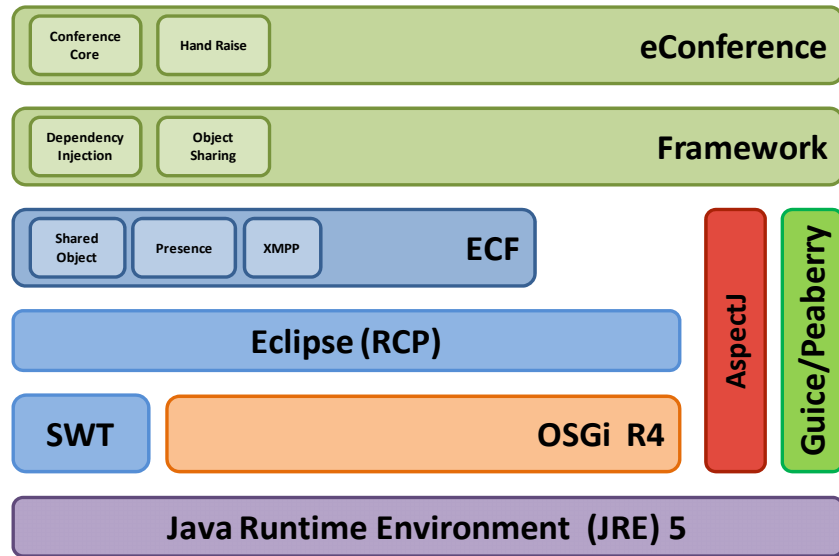
Figure 1. eConference architectural blocks.

The key to understand the rationale behind our design choices concerning ECF is the adoptions of two other well-known design patterns, described in the following subsections. First, the Proxy pattern was used for more easily handling the synchronization across remote clients through the ECF Shared Object API. Second, the Dependency Injection pattern was used to solve the problem of properly wiring together objects and favor a decoupled design and a better separation of concerns. AspectJ [18] and Peaberry [23] (an extension of Guice [16] capable of handling the specifics of OSGi) were used to provide the Dependency Injection solution.

Our tool, like any Eclipse RCP-based application, consists of a set of plugins, each providing the actual features. Most of those internally developed plugins are expected to comply with a basic MVC-pattern [12]. In fact, in eConference a plugin typically has one or more views, to show or edit data through user-accessible actions, a model representing the data on which the application operates, and finally has some logic for controlling and reacting to user interaction and model changes. MVC is a well-known design pattern and is not detailed here for the sake of space. However, it is worth noting that in eConference the model part is made more complicated by the need of keeping shared model objects in synch across the network between all the remote instances.

3.1 Integration of the Eclipse Communication Framework

Adopting ECF in an application rewritten from scratch is less troublesome than integrating it with an existing application that makes heavy use of other frameworks, as in the case of eConference 3. This happens because frameworks are typically designed for extension rather than for integration [19]. In addition, frameworks are in

general hard to learn and developers need some amount of time before they can use them effectively and efficiently [24]. In this sense, ECF is not an exception. However, its learning curve turned out to be particularly high due to the lack of documentation, which forced us to trace the framework source code to fully understand its behavior. Tracing is by itself a time-consuming activity, but in this case it took up even more because ECF code is designed to be multi-threaded and highly asynchronous. All the effort spent in understanding the ECF framework resulted in an internal document, initially meant for helping master students catching up with the eConference 4 project, which was later contributed back to the ECF community in form of an official Wiki page [7].

The main need we have in eConference is to share through the Shared Object API a common set of objects, containing information about the conference status across several clients. Another hurdle that we encountered was that a large amount of boilerplate code, devolved to standard initialization and monitoring of communication events, had to be cloned in every custom plugin that we were going to develop. Code cloning is a severe error-prone practice, which was avoided altogether by designing a common facade to be used for the task of sharing objects. This facade is implemented applying the Proxy adapter pattern [13], which can handle the ECF remote events, that is, notify remote clients about local changes and, vice versa, change local model according to remote changes.

Fundamentally, in eConference over ECF there are plugins that provide application-wide services and are exported to other plugins as OSGi services, thus providing access to eConference functionalities by the means of a public API. For example (see Figure 2), the *ConferenceManager* service provides clients with the possibility to create a new conference, join an existing one or leave an actual conference. The *ConferenceManager* API is exposed by a public interface, *IConferenceManager*, not shown in the figure, but assumed to be present. The *ConferenceManager* mainly has the duty to handle the model, which is an abstract *Conference* object containing all the conference information (e.g., participants, conference topic, agenda). Changes to the model are propagated to remote clients by the means of a transparent proxying mechanism. The manager just uses the *Conference* model's interface API and the underlying proxy changes the actual local model and propagates changes to the remote clients. Changes happening on remote hosts are handled in a similar way. The proxy listens for remote clients' change events and replicates by executing the methods on the local client.

The *ConferenceManager* object can listen to standard ECF events (thanks to a reference to the *IChatRoomContainer* object that is passed to it when a new conference is started) and also to model events by the means of *SharedObjectProxy* objects which is completely transparent to the manager since this can access the *IConference* interface methods.
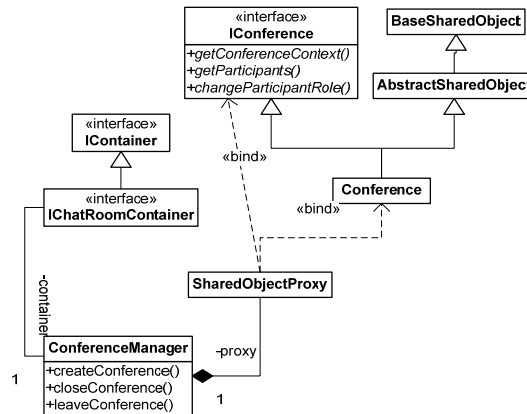
Figure 2. The conference manager provides a public API for managing a conference while the Shared object proxy object transparently keeps in synch changes to the Conference model object.

The designed proxies completely adhere to ECF Shared Object API with respect to the concept of primary and secondary clients: the former is a client that has initially created and shared an object (e.g., the case of a client starting a new conference), while the latter just waits for some remote primary client to notify about the availability of such objects (e.g., clients that have just joined to a chat room).

Using a proxying mechanism has made code simpler to write, understand, and maintain because the network synchronization concern is now encapsulated without having to the change the business logic on the client side. In fact, clients continue to use the model's API in a completely transparent manner. However, there is still the limitation that a manager must be explicitly aware about the existence of the proxy. We are currently working on an Aspect-based solution that will make proxy existence transparent.

3.2 Integration of the Dependency Injection

Although across the three previous releases of eConference the MVC architectural pattern had proven successful to cleanly separate the different concerns of software implementation, we were not completely satisfied because we had to manually assemble the MVC triplets together by the means of setter methods, thus generating much boilerplate and error-prone code.

Dependency Injection [21] is a software design pattern that separates the problem of objects collaboration from the problem of wiring them together. Dependency Injection ensures two main benefits: 1) less code to write to wire objects together; 2) the ability to provide different wiring configurations (e.g., one for testing and one for production).

This is possible because the software is composed by aggregating simpler, loosely-coupled objects that are more easily unit-testable [21], [26]. Additionally, by separating the clients by their dependencies, we also make their code simpler because there is no need for them to search for their collaborators. A third actor, called a *Container*, is configured to inject them into clients (see Figure 3).
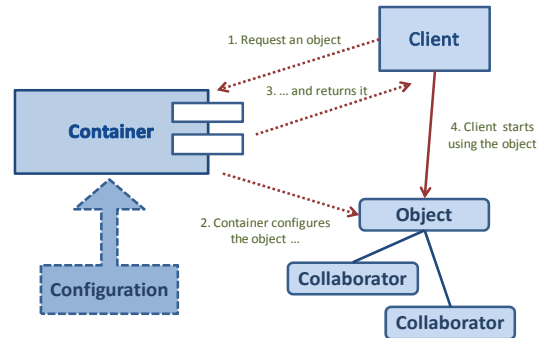

Figure 3. The client requests to the container a configured object. The container uses the configuration to build the object, find its dependencies and return it to the client.

For implementing a comprehensive Dependency Injection solution that would also fit well in the Eclipse technology ecosystem, we needed:

1. A container that allows configuring objects created out of container's control, like Eclipse views, editors, and user interface commands.

2. A way to decouple the clients from the container itself so as to avoid objects to be tied to a particular container instance.

3. Support for non-delegating classloaders used in Eclipse/OSGi.

As far as the first point is concerned, because eConference over ECF is an Eclipse RCP-based, many framework objects that we used were out of our control. Hence, we also needed a way to configure these objects out of container's control (e.g., the Eclipse UI items like toolbar buttons or menu items), which need additional dependencies to be satisfied before they could be properly used. Such objects need to find the container instance and ask to it to configure them. To make things clearer, their constructors will be similar to the following:

```
public MyActionDelegate() {
    Container.getInstance().configure( this );
    // Any other initialization code …
}
```

There are several Dependency Injection containers, mostly open source, like the Spring Framework Container [25]. Yet, we selected Guice/Peaberry [16], [23], which is a completely Java-based solution that uses annotations to mark dependencies, allowing field, method, and constructor injection. In fact, Guice/Peaberry container configurations, called modules, are simple Java classes, which results particularly helpful in terms of ease of refactoring. The other option, the Spring Dynamic Modules

for OSGi Service Platforms, was also taken into account at the time of architecture definition but it was discarded because configuration happens via XML files and no stable or complete release was available yet.

This approach, however, has one main limitation in the sense that it couples the object to the way the container actually works. Hence, with respect to the second point, we resorted to Aspect Oriented Programming (AOP) [17], and in particular to AspectJ [18], and implemented an aspect to overcome it. Such aspect captures injectable instances and weaves in the code necessary to call the container for configuring any new instance. By packaging the Dependency Injection aspect, support classes and bundle metadata together, we modularized the Dependency Injection concern and, thus, we are now able to reuse it across multiple Eclipse-based plugins, or even applications, without establishing any hard dependency at compile-time.

Finally, the third points regards the classloading policy of Eclipse that differs from the default Java policy, which dictates that the classloader, before attempting to load a class itself, should ask to the parent classloader first. In Eclipse, instead, each plugin has its own classloader, which can be called as a non-delegating classloader, that exhibits a different behavior, dictated by the OSGi specifications. A bundle classloader delegates to the classloaders from required and imported bundles first. Because of this particular behavior within Eclipse, the standard AspectJ implementation, which is unaware of the change in the classloading policy, is not able to find the classes to be woven. In fact, a cyclic dependency problem arise when AspectJ runtime tries to weave an aspect from one bundle to a class in another target bundle since the target bundle needs to depend on the aspect bundle, which at the same time needs to have the class to be woven within its own class scope.

Equinox Aspects [9] is an incubator project of the Eclipse community that fixes this kind of issues and, thus, makes real weaving possible in an Eclipse-based application. More importantly, from our perspective, we can now write aspects that can access the BundleContext object of each bundle and track services by a per-bundle policy. This means that we can inject OSGi services, that is, objects whose lifetime is linked to the lifetime of their hosting bundles. In fact, we already encountered this kind of problem when, in order to inject OSGi services, the OSGi API required registering listeners objects that can be only accessed through the BundleContext API. The OSGi framework passes this object to the bundle's activator life-cycle methods only. Then, our aspect has to capture such methods as well and track the bundle context object for the dependency injection mechanism to work.

In our solution (see Figure 4), we have defined an *AbstractDependencyInjection* aspect where the only the *withinScope()* pointcut is to be defined by sub-aspects in order to define which Java packages must be woven in the target bundle.

When used at compile time (e.g., for creating a new aspect by inheriting from the abstract dependency injection aspect), sub-aspects can provide their own configuration by overriding the *getModules()* method, which also means that the AspectJ compiler must be used to compile code (classes and aspects). By using Equinox Aspects, the sub-aspect is synthesized at load-time by the AspectJ runtime weaver. At the cost of a bit slower start-up time, this solution eliminates the need for
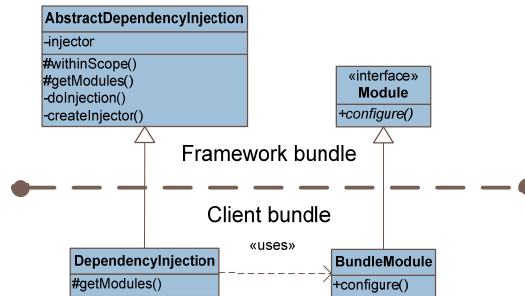
Figure 4. Compile-time weaving requires client code to define a concrete aspect that can be woven with the other classes and/or aspects belonging to a bundle.

compile-time dependency upon AspectJ and allow using any Java 5-compliant compiler.

4   Conclusions & Future Work

In this paper we have described the problems encountered for integrating the Eclipse Communication Framework (ECF) in the fourth release of the eConference project. The issues of weaving objects together and remote object synchronization were respectively overcome using two design patterns, namely, the Dependency Injection and Proxy patterns, and Aspect Oriented Programming (AOP).

The approach taken suffers from no particular drawbacks, apart from the additional dependencies on third party software, such as Guice as dependency injection container. Nevertheless, our solution is general enough to the point that we have developed it as a general purpose framework, called Penelope because of its ability to weave objects together, which can be reused by any Eclipse-based application.

Although the presentation of our Penelope framework's specifics is beyond the scope of this paper, our next goal is aimed to improve it by 1) providing an infrastructure to ease up the usage of the Model-View-Presenter [11] architectural pattern within Eclipse RCP and, consequently, adopt a Presenter-first [1] approach to support test-driven development; 2) adding support to Guice/Peaberry for Eclipse extensions that hook up into views, editors, commands, and similar.

References

1.   Alles, M., Crosby, D., Harleton, B., Pattison, G., Erickson, C., Marsiglia, M., Stienstra, C., "Presenter First: Organizing Complex GUI Applications for Test Driven Development", Proceeding of the Agile Conference, 23-28 July 2006.
2.   AspectJ Development Tools, http://www.eclipse.org/ajdt
3.   Calefato F., Lanubile F., and Scalas M. "The Evolution of the eConference Project", Proc. Int'l Conf. on Eclipse Technologies (Eclipse-IT 2007), Naples, Italy, 4-5 October, 2007.

4. Calefato F., Lanubile F., and Scalas M. "Porting a Distributed Meeting System to the Eclipse Communication Framework", Proc. OOPSLA Workshop on Eclipse Technology eXchange (ETX 2007), Montréal, Canada, October 21-22, 2007.

5. Clayberg E., and Rubel D., "Eclipse: Building Commenrcial-Quality Plug-ins", 2nd edition, Addison Wesley Professional, 2006.

6. Eclipse Communication Framework (ECF), http://www.eclipse.org/ecf

7. Eclipse Communication Framework Wiki, "Sharing Objects over XMPP", http://wiki.eclipse.org/Sharing_objects_over_XMPP

8. eConference over ECF, http://econf.di.uniba.it/econference-over-ecf/

9. Equinox Aspects, http://www.eclipse.org/equinox/incubator/aspects/

10. Fowler, M, "Inversion of Control Containers and the Dependency Injection pattern", http://martinfowler.com/articles/injection.html

11. Fowler, M., "Model View Presenter", http://martinfowler.com/eaaDev/ModelViewPresenter.html

12. Fowler, M., "Patterns of Enterprise Application Architecture", Addison Wesley Professional, 1st edition, 2002.

13. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.M., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional, 1994.

14. Garlan, D., Allen, R., Ockerbloom, J., "Architectural Mismatch or Why it's hard to build systems out of existing parts", Proceedings of the 17th International Conference on Software Engineering, Seattle WA, April 1995.

15. Gruber, O., Hargrave, B.J., Mcaffer, J., Rapicault, R., and Watson, T. "The Eclipse 3.0 platform: Adopting OSGi technology," IBM Systems Journal, Vol. 44, No. 2. (2005), pp. 289-299.

16. Guice Container, http://guice.googlecode.com

17. Kiczales,G., Lamping, J. et Al., "Aspect Oriented Programming", Proceedings of the European Conference on Object-Oriented Programming, vol.1241, pp.220-242, 1997

18. Laddad, R., "AspectJ in action", Manning, 2004.

19. Mattson, M., Bosch, J., Fayad, M. E., "Framework Integration: Problems, Causes, Solutions", Communications of the ACM, October 1999, Vol. 42, No. 10.

20. McVeigh, A., "The Rich Engineering Heritage Behind Dependency Injection", http://www.javalobby.org/articles/di-heritage/

21. Meszaros, G., "xUnit Test Patterns", Addison Wesley, 2007.

22. Open Service Gateway initiative (OSGi), http://www.osgi.org

23. Peaberry, http://peaberry.googlecode.org

24. Schmidt, D.C., Gokhale, A., Natarajan, B., "Leveraging Application Frameworks", Queue, July/August 2004.

25. Spring Framework, http://www.springframework.org

26. Weiskotten, J., "Dependency Injection and Testable Objects", Dr. Dobbs Journal, http://www.ddj.com/development-tools/185300375

27. XMPP - eXtensible Messaging and Presence Protocol, http://xmpp.org