Weaving Eclipse Applications

Fabio Calefato, Filippo Lanubile, Mario Scalas,

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

Abstract. The Eclipse platform fully supports the ideas behind software components: in addition it also adds dynamic behavior allowing components to be added, replaced or removed at runtime without shutting the application down. While layered software architectures may be implemented by assembling components, the way these components are wired together differs. In this paper we present our solution of Dependecy Injection, which allows to build highly decoupled Eclipse applications in order to implement real separation of concerns by systemically applying Aspect Oriented Programming and the Model-View-Presenter pattern, a variant of the classic Model-View-Controller.

Keywords: Eclipse, Aspect Oriented Programming, Dependency Injection.

1 Introduction

The Dependency Inversion Principle [19] (DIP) states that (both high and low level) software parts should not depend on each other's concrete implementation but, instead, be based on a common set of shared abstractions: one application of the DIP is the Dependency Injection, also called Inversion of Control (IoC) [11]. From an architectural perspective, DI allows to explicit the dependencies between software components and provides a way to break the normal coupling between a system under test and its dependencies during automated testing [25].

This is possible because the software is composed by aggregating simpler, loosely coupled objects that are more easily unit-testable [32]. 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.

The Eclipse Platform [3],[8] is a collection of frameworks for building integrated development environments that has expanded to cover also the development of Rich Client applications [21]. Its building blocks are the Open Services Gateway Initiative (OSGi) [26] specifications, which define a dynamic module system for Java so as to offer a plugin-based component model, and the Standard Widget Toolkit (SWT), a graphic library which provides native application look and feel. However, the Eclipse platform does not have Dependency Injection built-in.

Dependency Injection has proved to be a valuable architectural asset [11],[30]. In particular, according to our own experience [4],[5], during the development of the eConference over ECF [6], a text-based conferencing tool based on Eclipse technologies developed internally, we integrated this pattern as a common asset to be

used for developing every plugin. Rather than creating yet another Dependency Injection framework, we decided to reuse an already existing solution, while only providing the necessary glue-code. In this paper we present how we have used Aspect Oriented Programming to implement Dependency Injection and support the Eclipse dynamic component model.

The remainder of this paper is structured as follows. Section 2 will present an outline of the Dependency Injection and its use cases; section 3 will describe the issues involved in implementing it within Eclipse; section 4 will present the broader context in which we are applying it. Finally, section 5 will present conclusions and future work.

2 Dependency Injection

Dependency Injection comes from the research field of Architecture Description Languages (ADLs), which attempts to assemble or wire components together via configuration mechanisms.

A component is a unit of software that can be instantiated and is insulated from its environment by explicitly indicating (via interfaces) which services are provided and required [23]. The idea of software component comes from the field of electronics engineering: building software should be like wiring electronics components. As long as interfaces are compatible, we should be able to replace old components with new ones, an idea as old as 1968 [22].

The rest of this section provides a brief introduction about the Dependency Injection in general and the Eclipse component model.

2.1 Background

Component Based Software Engineering (CBSE) has two basic concepts, Component Types and Component Instances, which can be respectively mapped to Classes and Instances in Object Oriented Programming (OOP).[1]

More specifically, in Java a class may be seen as a component declaration, thanks to the definition of the implemented interfaces, which can be used as a description of the services it provides. Nonetheless, a class definition fails to declare its dependencies and some kind of convention is required to describe which interfaces are required. This is where containers and configuration mechanisms kick in (see Figure 1).

Clients relinquish to directly instantiate objects and, instead, request them to the container. The latter will use its own configuration, describing the object dependency graph, to retrieve such an instance and return it to the client for usage.

[1] Within the rest of this paper we will use the terms "objects" and "components" as synonyms unless we explicitly provide a different meaning for the different cases.

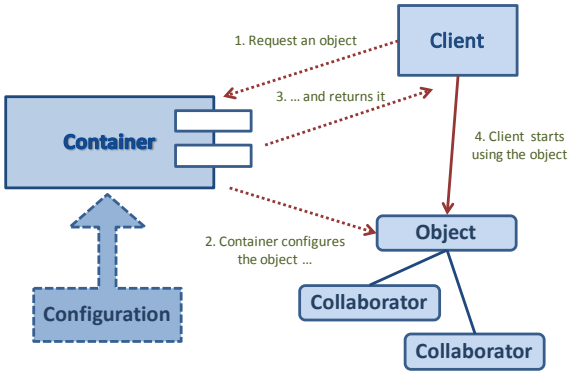

Fig. 1. Dependency Injection

In this scenario, the container itself becomes a key component of a software architecture: it must be boot-strapped before the application starts running and its lifecycle is parallel to the application's. When included in full-fledged web application frameworks, like Spring [30], the container is transparent to the application code: the application must be still aware of the container services but must not care about bootstrapping it since the web framework is handling this task by integrating itself within the application server infrastructure (i.e., a J2EE Application Server). We will call these *managed containers*.

In other uses cases, the container must be explicitly started by some initialization code before it can be used by the client: in this case the client must have direct access to container instance in order to perform requests for objects. We will call these *unmanaged containers*. Integrating a container within the Eclipse Platform is such a case.

Historically, three ways that allow clients to explicit their required dependencies are used:

- Type 1 or Interface-based injection, where clients must implement specific interfaces in order to tell the container which collaborators they need.
- Type 2 or Setter Injection, where clients declare their dependencies by the means of setter methods, which accept specifics collaborator types.
- Type 3 or Constructor Injection, where clients' constructor parameters are their dependencies.

Type 1 is nowadays an inheritance from the past. Setter injection supports the Java Beans convention about class' properties: the setter methods will be used by the container to inject the dependencies. While this is a simple solution, it also opens the class contract by allowing the dependency to be changed at a later time. Constructor injection is stricter about the class contract: dependencies are provided at object instantiation-time and can never be changed as long as an object is alive.

In order to write container configurations, a Domain Specific Language [10] (DSL) is required. A DSL (such as CSS, regular expressions and SQL) is a language

targeted for a particular and limited purpose, not a fully fledged programming language.

A DSL can be *internal*, that is, implemented by using an host language, an approach popularized by the Ruby language, often providing a fluent API.

*External DSL*s, instead, use their own syntax and require a parser to be used. In the case of Dependency Injection, XML has been the most used language, although its syntax badly suits the purpose because of its verbosity-over-expressiveness ratio.

The appearance of built-in annotations within the Java platform from the release 5.0 has enabled an additional way for declaring dependencies, thus pushing several container projects, like Google Guice [15], Pico Container [28] and even Spring, to opt for internal DSLs. The client code will then use library-provided annotations to mark methods or even fields that have to be used to inject required objects whereas the container will use class introspection to scan for annotations and set the required object references.

Internal DSLs have multiple advantages over external DSLs. With an internal DSL: (1) developers have just a single source file to track; (2) the fluent interface is written in the same programming language of the application (e.g., Java), which typically benefit from strong refactoring tools available in many modern IDEs; (3) there is early syntax check. By converse, with internal DSL an abuse of annotations may produce a less readable source code.

Hence, we decided for an internal DSL-based solutions and opted in particular for the Google Guice framework because of the existence of an extension, Peaberry [27], which supports the OSGi component model.

2.2 The OSGi Component Model

OSGi is a set of specifications that define a dynamic module system for Java. In OSGi, components may hide their implementations from other components by the means of *Services*, objects shared across several components (see Figure 2).

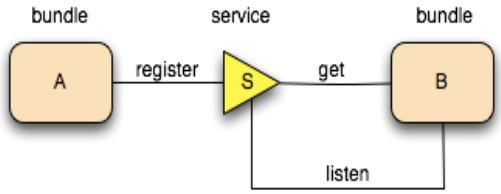

Fig. 2. OSGi publish-subscribe mechanism (from http://www.osgi.org)

Services use a publish-subscribe pattern: components start listening for specific services registered by other bundles. The *Service Registry* framework takes care of tracking down the service instances while specific API is to be used by subscribers, to get actual service instances, and publishers, to make service implementations available to the rest of the system.

Services are deployed within bundles (a synonym of plugin) and the latter can be installed, removed, or updated without shutting down the whole system. Hence, because services can become available or unavailable over time, a service tracking API is needed.
The vision that OSGi designers intended to endorse is that of a *collaborative environment* where applications emerge by dynamically assembling different components with no *a-priori* knowledge of each other (see Figure 3). One of the biggest advantage of OSGi consists in the ability to update, change or introduce new functionalities in a running software system without shutting it down, which is a why OSGi is interesting for application server vendors.

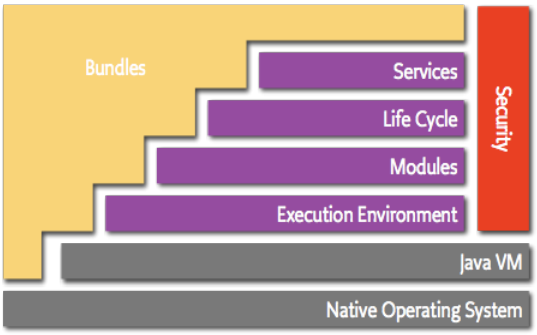

Fig. 3. The OSGi architecture layers (from http://www.osgi.org).

Application bundles use the framework services, such as the publish-subscribe mechanism, the dynamic lifecycle management, and standard Java security. The whole system is based on the concept of modularity: bundles are just plain JAR files with additional OSGi metadata, which define public and private parts. By versioning bundles and, therefore, services, it is possible to have within the same Virtual Machine different versions of the same classes.

Having to deal with dynamic services poses an important question when thinking about a Dependency Injection of OSGi services. In this case, infact, a static injection is not suitable since object structure graph is going to change over time. A simple solution is the introduction of service proxies, as implemented by the Guice/Peaberry. Service proxies act like placeholders for real services: when the actual service component is available, then the proxy passes the call on, otherwise it throws a Service Unavailable exception.

While there are several implementations of the OSGi platform specifications, the current reference implementation is the Eclipse Equinox runtime, the core on top of which the whole Eclipse eco-system is built. In addition to OSGi services, the Eclipse platform historically supports another mechanism for extending software functionalities through platform extensions (plugins). Extensions allow components to be declared and made available to the system, without the need to be loaded until they are actually used (lazy loading).

3 Weaving Dependency Injection

The Eclipse Platform does not support Dependency Injection out-of-the-box: integrating it becomes a framework integration problem, in this particular case, of the Guice and the OSGi frameworks. This section first describes the usage of AOP and then outlines the problems of integration and related solutions.

3.1 AOP and Eclipse

Aspect Oriented Programming [16] (AOP) is a programming paradigm addressing the separation of concerns into reusable modules called *aspects*. AOP complements classical OOP rather than replacing it: while classes modularize primary application concerns (like domain entities, business services, or user interface views), aspects encapsulate secondary, or system, concerns, such as transactions, tracing, security policy enforcement, or performance monitoring.

Merging classes and aspects together is a process called *weaving* and it is usually performed at bytecode level. The weaving process may be executed at compile time (compile time weaving, CTW), by the means of an ad-hoc compiler, or at load time (load time weaving, LTW), by a *weaving agent* that intercepts class loading operations performed by the Java Virtual Machine. At the base of AOP there is the *Join Points Model*, an abstraction for the OOP language constructs, which exposes where aspects can be hooked in the code (e.g., method calls or constructor invocations). An aspect, then, is a construct composed by two parts: a rule-based section, specifying which joint points to capture, and a body part, containing which code to apply when the rules match.

AspectJ [17] is an AOP solution for Java that has tooling support within the Eclipse IDE [2]. Supporting AOP within a dynamic environment as Eclipse poses issues with the aspects weaving: (1) plugins hosting aspects that were woven on classes belonging to other plugins may be become unloaded (i.e., because updated) so the original unwoven classes should be restored before any other re-weaving is possible; (2) new bundles hosting new aspects may be installed within the system and needed to be woven on already loaded classes. All of these cases can only be supported through a careful implementation of LTW, which is the purpose of the Equinox Aspects project [9], which provides new metadata for supporting the two aforementioned scenarios, a set of bundles exporting the weaving service as an OSGi-compliant weaving agent, and a bytecode caching service to improve runtime performance.

The most recent implementation also supports language metadata (through to Java annotations) enabling a declarative way for expressing concerns ([18]).

When implementing Dependency Injection as a system concern, the primary domain concern is the application code requesting the provisioning of collaborators. A possible implementation of the former is detailed in the next section.

3.2 AOP as gluecode

The idea of using Dependency Injection as a system-wide cross-cutting concern and as a reusable abstract base aspect is not new: frameworks like Spring already use it [31]. In particular, programmers mark fields to be injected with ad-hoc annotations like @Autowired so that a special Spring facility, called a weaving agent, will scan components and provide the required dependencies at objects' instantiation time. AOP is then used in order to match the annotations and wire the required code to perform the operation. Nevertheless, implementing the same idea in a dynamic component architecture like OSGi (and Eclipse) requires, instead, special care dealing with the services' dynamic behavior and the different classloading architecture. In fact, an aspect performing Dependency Injection needs to: 1) have access to the BundleContext objects (different for every plugin) in order to access the OSGi services; 2) be provided with a configured container instance (i.e, a Guice container instance); 3) support plugins loading/unloading and, consequenty, aspects corresponding weaving/unweaving (for example, by using Equinox Aspects). In this context, such an aspect will contain all the code necessary to wire objects together with their container, with concrete aspects only differing for the scope of its application (i.e., the packages to weave).

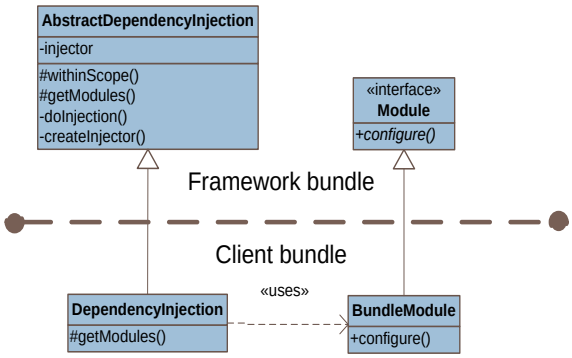

Fig. 4. Modularization of Dependency Injection.

The ability to reuse a common implementation for different contexts is really useful when we have to deal with plugins. Because of the Eclipse platform specifics, in fact, we need to have different concrete Dependency Injection aspects, one for each plugin.

In this model a plugin may publish one or more service objects implementing a contract, that is, a standard Java interface. These services are tracked by the OSGi Service Registry and made available to the rest of the system. Client plugins may request implementations of such contracts and use them as seamless Java Objects with no overhead (apart from the retrieval operations). By intercepting framework events, clients may track their needed dependencies but, in this model, application code intermixes business code with system code. Often, it may be simpler to wrap the objects behind a Proxy and have the latter deal with the OSGi behavior, throwing

exceptions if clients try to use unavailable objects. This is also the solution adopted by Peaberry.

Additionally, to track service objects, the OSGi API is accessible only through the *BundleContext* object which is passed to the plugin Activator's *start()/stop()* methods: this is the standard mechanism provided by the framework to enable client bundles to be notified about events. The bundle context is obviously different for each plugin, so we have to implement a different Dependency Injection aspect for each plugin in order to capture the right bundle context.

In Eclipse-based application, developers are required to provide implementations of standard framework interfaces or classes in order to take advantage of the Eclipse facilities. Frameworks are designed for adaptation and extension, not for integration [20] and Eclipse is no exception since there is little room for configuring the objects that are being created by the platform.

One solution would be to employ the Singleton pattern for locating the container instance and have the newly instantiated object to inject itself, as shown in Listing 1.

```
public class MyCommandHandler extends AbstractHandler {

    @Inject private SomeService someService;

    public MyActionCommand()  {

        // Use Singleton to retrieve the container

        // and call its services ...

        Container.getInstance().configure( this );

    }

    public Object execute ( ExecuteEvent event ) {

          someService.doSomething();

          return null;

    }

}
```

Listing 1. Usage of the Singleton pattern to perfom injection of platform created objects

At runtime, when the default constructor is invoked by the Eclipse framework, the container is also invoked and the dependency injected. Employing Singletons to gain access to the container instance is simple to implement but also defeats the decoupling we are searching in our software system because we are tightly wiring the specific container instance with the client code. Though there is no real other way out with standard OOP but it is still possible to achieve the same effect without any

“hardwiring” of the dependency between the client code (our command handler) and the specific container instance.

The basic idea behind this is to employ the (concrete) Dependency Injection aspect to effectively act as glue-code between the application code instantiated by Eclipse and the container while keeping both separated.

The first action of the Dependency Injection aspect is to intercept the call of the *start()* method to capture the *BundleContext* object and the *stop()* method in order to release service objects when they are no more needed (because OSGi uses reference counting to know when a service object can be released). After this, the Dependency Injection aspect will intercept the creation of instances of classes annotated with the *@Injectable* annotation and configure them. The resulting effect at runtime is the same as in previous solution (i.e., the constructor will get modified at runtime by the weaving agent), but the code concerns remains separated and testable in isolation. Thus, we are able to inject even objects that are written by developers, but instantiated by the Eclipse Framework (e.g., views or command handlers). Doing so, we are using AOP as an integration layer for different frameworks (Eclipse and Guice) in order to bind the application components together [29] (i.e., views with their business service objects), which is also one of the basic steps we need in order to proceed towards further developments, as outlined in the next section.

4 Implementing Model-View-Presenter

Separating presentation from domain means ensuring that no part of the domain code refers to any part in the presentation code [14]. This means that, when writing a WIMP (Windows, Icons, Mouse and Pointer) GUI application, it should also be possible to write a command line interface with the same functionalities without touching the domain code.

Systematically applying the Model-View-Controller [13] (MVC) architectural pattern is a way to enforce separation of concerns since it organizes GUI applications along three primary concerns:

- Model, encapsulating the domain logic behind a set of abstractions (classes and interfaces);
- View, showing the Model's content and notifying the input events to the Controller;
- Controller, which reacts to Model and View events according to some behavior.

Model View Presenter [12] (MVP) is an MVC-variant which further separates Model and View so that they no longer knows about each other; instead, the Controller (called Presenter) is the only listening to both layers' events, driving them according to some application logic, which can be tested.

Separation of presentation and domain logic means not only a way to increase the reuse software parts, but also to design better testable software. In fact, while tools exist to capture mouse clicks for web user interfaces, the resulting macros are tricky to maintain. Separating the domain code improves testability: the greater testability is, the better design becomes.

Additionally, MVP can be applied it in a test-driven process by using the Presenter-first technique [1]. Because this approach avoids dealing with the UI directly, the views must be simple as they only present results or perform data-binding. Testing the presenter means unit-testing it. Dependency Injection finds its application also during testing to assemble the right MVP triplets.

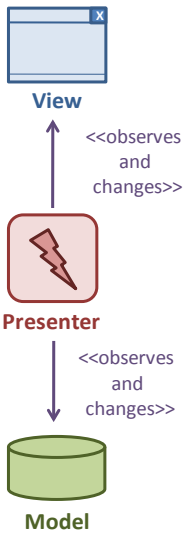

Fig. 5. Modularization of Dependency Injection.

5 Conclusions and future work

At this time we have implemented the Dependency Injection bundle in a project of ours, eConference [4], [5], [6]. eConference is an Eclipse RCP-based distributed meeting system. The primary functionality provided by the tool is a text-based group chat, augmented with agenda, meeting minutes editing, and typing awareness capabilities. Around this basic functionality, other features have been built to help organizers to control the discussion during distributed meetings. The tool has been successfully used to offer the students the opportunity to experience development of software in geographically, distributed multi-cultural teams [7]. The current generation of eConference, eConference-over-ECF, is built on top of the Eclipse Communication Framework and has won the 2006 Eclipse Innovation Award.

As future work, we expect to proceed through the following steps:

1. Extract the framework bundles (like Dependency Injection) from eConference in an order to define a reusable tool for other applications.
2. Perform an architectural check-up of eConference.

3. Design and implement the MVP test and runtime bundles by using eConference-over-ECF as a proof of concept (e.g., the whiteboard and file transfer bundles)
4. Extend Guice and Peaberry in order to support Eclipse concepts and make the process of writing tests for this environment a streamlined process.
5. Get feedback from academic as well as industry projects.

Acknowledgement

This work has been supported by the 2008 IBM Faculty Award.

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. Birsan, D., “On Plug-ins and Extensible Architectures”, Queue, ACM, vol. 3, n. 2, March 2005, pp. 40-46.
4. Calefato, F., Lanubile, F., Scalas, M., "Porting a Distributed Meeting System to the Eclipse Communication Framework", Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange. p. 46-49, New York, 2007
5. Calefato, F., Lanubile, F., Scalas, M., "Evolving a Text-Based Conferencing System: An Experience Report", Collaborative Computing: Networking, Applications and Worksharing. p. 427-431, Los Alamitos, 2007
6. Calefato, F., Scalas, M., "Adopting the Eclipse Communication Framework: The Case of eConference", Proceedings of the 3rd Italian Workshop on Eclipse Technologies (Eclipse-IT 2008). Bari, Italy, 2008
7. Damian, D., Lanubile, F., Mallardo, T., "An empyrical Study of the Impact of Asynchronous Discussions on Remote Synchronous Requirements Meetings", Lecture Notes in Computer Science, Vol. 3922, 2006
8. Eclipse Platform, http://www.eclipse.org
9. Equinox Aspects, http://www.eclipse.org/equinox/incubator/aspects/
10. Fowler, M., "Domain Specific Language" (Book web draft), http://martinfowler.com/dslwip/
11. Fowler, M., “Inversion of Control Containers and the Dependency Injection pattern”, http://martinfowler.com/articles/injection.html
12. Fowler, M., “Model View Presenter”, http://martinfowler.com/eaaDev/ModelViewPresenter.html
13. Fowler, M., “Patterns of Enterprise Application Architecture”, Addison Wesley Professional, 1st edition, 2002
14. Fowler, M., "Separating User Interface Code", IEEE Software, March/April 2001
15. Google Guice, http://guice.googlecode.com
16. Kiczales,G., Lamping, J. et Al., “Aspect Oriented Programming”, Proceedings of the European Conference on Object-Oriented Programming, vol.1241, pp.220-242, 1997
17. Laddad, R., “AspectJ in action”, Manning, 2004

18. Laddad, R., "AOP and metadata: A perfect match", http://www.ibm.com/developerworks/java/library/j-aopwork3/
19. Martin, R. C., "Dependency Inversion Principle", http://www.objectmentor.com/resources/articles/dip.pdf
20. Mattson, M., Bosch, J., Fayad, M. E., "Framework Integration: Problems, Causes, Solutions", Communications of the ACM, October 1999, Vol. 42, No. 10.
21. McAffer, J., Lemieux, J-M., "Eclipse Rich Client Platform: Designing, Coding, and Packaging Java™ Applications", Addison Wesley Professional, 2005.
22. McIlroy, M. D., "Mass Produced Software Components", "Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968", Scientific Affairs Division, NATO, Brussels, pg. 138-155, 1969 (a transcript can be found at http://www.cs.dartmouth.edu/~doug/components.txt)
23. McVeigh, A., "The Rich Engineering Heritage Behind Dependency Injection", http://www.javalobby.org/articles/di-heritage/
24. Melnik, G. , Maurer, F., Chiasson, M. Executable Acceptance Tests for Communicating Business - Requirements: Customer Perspective. In Proc. of the Agile Conference (AGILE'06), IEEE Computer Society, pp. 35-46, July 2006.
25. Meszaros, G., "xUnit Test Patterns", Addison Wesley, 2007
26. OSGi Consortium, Open Service Gateway initiative (OSGi), http://www.osgi.org
27. Peaberry, http://peaberry.googlecode.org
28. PicoContainer, http://www.picocontainer.org
29. Schmidt, D.C., Gokhale, A., Natarajan, B., "Leveraging Application Frameworks", Queue, July/August 2004.
30.Spring Framework, http://www.springframework.org
31. Walls, C., Breidenbach, R., "Spring in Action", Manning Publications, 2008
32.Weiskotten, J., "Dependency Injection and Testable Objects", Dr. Dobbs Journal, http://www.ddj.com/development-tools/185300375