A Gentle Introduction to OSGi

Andre L. C. Tavares, Marco Tulio Valente
Institute of Informatics, PUC Minas
30535-610 - Belo Horizonte, MG Brazil
andrelct@gmail.com, mtov@pucminas.br

Abstract

The Open Services Gateway Initiative (OSGi) is a framework that supports the implementation of component-based, service-oriented applications in Java. The framework manages the life-cycle of modules (called bundles in OSGi) and provides means to publish and search for services. Moreover, it supports the dynamic install and uninstall of bundles. Nowadays, OSGi is used in many application domains, including mobile phones, embedded devices, and application servers. In this paper, we provide a gentle introduction to the basic services provided by OSGi. The presentation is guided by a simple case study, involving the implementation of a Dictionary System.

1 Introduction

Since the seventies, modularization is considered a key property for improving flexibility, comprehensibility, and reusability in software projects [12]. In general, modules correspond to units of work that can be independently implemented, compiled and deployed. For example, in Java classes encapsulate both data and services, according to information hiding principles. Packages are used to group and organize related classes, thus making reuse easier. Furthermore, packages can be distributed and deployed in the format of JAR files.

However, the modular system employed in Java has at least two problems. First, information hiding principles are only applied to classes, but not to the level of packages and JAR files. For example, it is not possible to restrict access to public classes defined in available packages. The absence of such visibility control rules can easily lead to highly coupled, "spaghetti-like" systems. On the other hand, the modular system employed in Java is inherently static. For example, modules can only be updated at deployment time, which requires stopping and starting again the system. This is a serious limitation in many domains, such as consumer electronics, industrial automation, application servers etc.

In order to address the limitations of the standard Java module system, the OSGi Alliance has proposed in 1998 the OSGi framework and programming model [6, 11], which provides a dynamic, service-oriented module system for the Java platform. According to OSGi principles, software systems must be structured around independent modules, called bundles, that provide well-defined services. The OSGi standard also specifies a runtime infrastructure for controlling the life cycle of bundles. This infrastructure allows developers to dynamically add and remove existing bundles.

In this paper, we provide a gentle introduction to OSGi principles, components and concepts. In order to guide our presentation, we rely on the motivating example of a flexible, bilingual Dictionary System, that supports translating words from a source to a target language. We show how OSGi allows developers to clearly define boundaries among the modules of such system. We also demonstrate that the architecture of the proposed system is flexible to support the insertion and removal of new components. Particularly, new language databases can be transparently included in the system, without restarting. After the inclusion of a new language database, it is automatically possible to perform translations to and from this language. Finally, we evaluate and compare the proposed solution with respect to an implementation solely based on the standard Java module system.

The remainder of this paper is organized as follows. Section 2 presents a description of the OSGi programming model and framework. Section 3 describes our OSGi motivating example in details. Section 4 compares and evaluates the example with standard Java. Section 5 briefly describes other OSGi services. Section 6 discusses related work and Section 7 concludes the paper.

2 OSGi Framework

The OSGi framework supports the implementation of applications from modular units, called *bundles* [6]. The framework also controls the life-cycle of bundles, which includes adding, removing and replacing bundles at run-time, while preserving the relations and dependencies among them. Currently, the following open source implementations of the OSGi Specification Release 4 are available: Knopflerfish [4], Apache Felix [1], and Equinox [3]. Oscar [5, 9] and Concierge [2, 13] implement OSGi Specification Release 3 (R3).

Basically, bundles are regular Java JAR files containing class files, other resources (images, icons, required APIs etc), and also a manifest, which is used to declare static information about the bundle, such as the packages the bundle import and export. Furthermore, bundles may provide services to other bundles. In the OSGi architecture, a service is a standard Java object that is registered using one or more interface types and properties (that are used to locate the service).

The OSGi run-time and installed bundles reside inside a single JVM. Running multiple applications in the same JVM has some benefits. It increases performance, minimizes the memory footprint and provides near zero-cost inter-application communication. But it also creates a num-

DOI: 10.1145/1402521.1402526 http://doi.acm.org/10.1145/1402521.140256

ber of sharing and coordination issues. In order to address such questions, each bundle is assigned to a different class loader, thus creating a particular address space for resources and classes packaged in bundles.

Another key component of the OSGi run-time is the Service Registry, which keeps track of the services registered within the framework, as illustrated in Figure 1. The Service Registry provides the necessary means for bundles to publish (step 1 of the figure) and retrieve services (step 2). Once a bundle has retrieved a service, it can invoke any method described in the interface of this service (step 3). A distinguished aspect of OSGi Service Registry is its dynamic nature. As soon as a bundle `A` publishes a service that another bundle `B` is looking for, the registry will bind these two bundles.

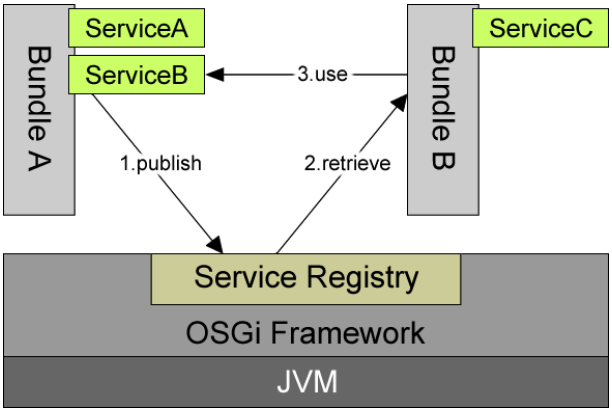

Figure 1: OSGi Service Registry (adapted from [14])

3 Example: Dictionary System

In order to better describe OSGi features, we will use as example a flexible, bilingual dictionary system. The proposed system is flexible because it relies on OSGi features to seamless support the introduction of new languages. Section 3.1 describes the translation algorithm implemented by the proposed dictionary. The architecture of the system is discussed in Section 3.2 and implementation details using OSGi are discussed in Section 3.3.

3.1 Translation Algorithm

A bilingual dictionary associates words in a source language to their corresponding translation in a target language. For example, a dictionary that translates between two languages must have two lists of words. As the number of supporting languages increases, the number of required databases increases exponentially. For example, to translate between two languages we need two databases, between three languages six databases, an so on.

In order to reduce the number of databases, we propose a translation algorithm based in an universal index. More specifically, we assume that a dictionary is composed by databases that associate words in a given language to an universal index. In this way, it will not be necessary to have databases like "English-to-Portuguese" and "Portuguese-to-English". Instead, there would be only "English" and "Portuguese" databases. With this approach, the number of databases no longer increases exponentially, since a single dictionary is required for each language.

Using this approach, a search for a given word will involve exactly two databases, i.e. the source database, which represents the language to translate from, and the destination database, which represents the language to translate to. As described in Figure 2, when a user asks for a translation of the word "you" from English to Portuguese, we will search for this word in the English database, obtaining its universal index (number 1 in the Figure). Using this index, the system makes a reverse search in the Portuguese database, obtaining the word "você".

Figure 2: Universal databases

This approach makes it easy to extend the dictionary system. For example, in order to provide support to a new language, we just need to include its own database, and we will be able to translate from this new language to any available language and vice-versa.

3.2 Architecture

The architecture of the proposed dictionary system is organized in three layers, as described in Figure 3. The first layer includes the database and their associated Data Access Objects (DAO). The second layer contains the handler in charge of implementing the translation algorithm. The third layer represents the system interface (including for example applications that demand translation services, such as text editors).

In the proposed architecture, databases are independent from each other, and are only accessed by the dictionary handler. This handler represents the main component of the proposed system. It is responsible for translating words to the upper layer. Basically, it receives a translate request, queries the source database for an universal index, queries the destination database for the destination word, and returns it to the third layer. Moreover, the handler must deal with a variable number of databases, without the need of restarting or recompilation.

The last layer in the proposed architecture contains the clients of the dictionary system. Because the handler provides a well-defined interface, any application that requires

DOI: 10.1145/1402521.1402526 http://doi.acm.org/10.1145/1402521.140256

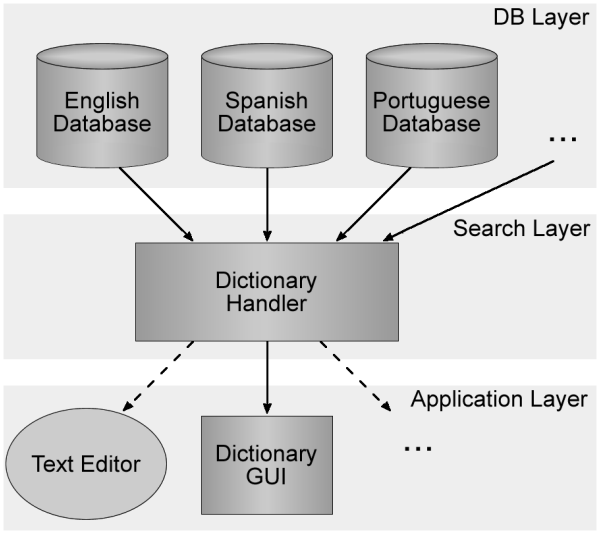

Figure 3: Dictionary architecture

translations may communicate with it (for example, GUI interfaces, text editors etc).

The proposed architecture supports two interesting characteristics: a high degree of decoupling and extensibility. More specifically, dependencies between modules are strictly regulated by interfaces. Therefore, modules can be updated at any time, without impacting the rest of the system. Moreover, the architecture is extensible because new databases and clients can be added without requiring changes in the system source code. The only requirement is that new components follow the handler's required interfaces.

3.3 OSGi-based Implementation

In an OSGi-based implementation for the proposed Dictionary System, the components described in Figure 3 correspond to bundles with well-defined interfaces. For example, bundles in the DB Layer must provide (or export) the following interface:

```
interface Search {
  public int[] toIndex(String[] phrase);
  public String[] fromIndex(int[] index);
}
```

In this interface, the `toIndex` method maps an array of strings into a correspondent array of universal indexes. The `fromIndex` method does the conversion in the opposite way.

Because all database bundles provide a service according to this interface, there must be a way to distinguish the provided services. This is purpose of the service properties. When a database bundle provides a service, it identifies that service with the language the bundle is providing (i.e. a string). With this information the handler is capable of acquiring all available services providing the mentioned interface.

In an analogue way, the handler provides a service for the Application Layer. Any bundle residing in the Application Layer should use this service to translate words. The interface provided by the handler is the following one:

```
interface Translate {
  public String translatePhrase(String dicInStr,
    String dicOutStr, String phrase);
}
```

This interface contains a single method that is responsible for translating a phrase (`phrase`) from an input dictionary (`dicInStr`) to an output dictionary (`dicOutStr`). This method splits the phrase into an array of words. With this array it queries the input dictionary for an array of indexes. With this second array, the handler queries the output dictionary for an array of destination words. This last array is concatenated and returned as the translated phrase.

This approach assumes that bundles in the Application Layer know the available languages. In order to solve this problem, the handler publishes the `Translate` service with a list of available languages (as one of the service properties). Therefore, as soon as a bundle retrieves this service, it will know what languages are available. Moreover, when a new language becomes available, the handler updates the `Translate` service properties and the bundles attached to that service will be notified of this change.

In OSGi, if a bundle depends on a service, it needs to import the service's interface. In other words, suppose a bundle `S` that provides a service described by interface `I`. In this case, any client bundle `C` must import `I`. However, since `I` is an interface that is packaged with `S`, this strategy requires that `S` is available in the system (i.e. that `S` has already been installed in the OSGi framework). Clearly, this is an impacting constraint. For example, using this strategy in our Dictionary System, a Text Editor will not be able to run if a Dictionary Handler has not been previously registered.

To solve this coupling issue, we must package the interface `I` both in bundles `S` and `C`. The server bundle `S` will export `I`, as before. However, the client bundle `C` will export and import `I`. Following this strategy, if `C` starts and no service providing interface `I` is available in the system, `C` will match its own exported version of `I`. On the other hand, if `C` starts and `S` is available, the `I` interface required by `C` will match the `I` interface provided by `S`.

The manifest of the Dictionary Handler is shown next:

```
 1: Manifest-Version: 1.0
 2: Bundle-Name: dictionary_handler
 3: Bundle-Activator:
 4:   org.dictionary.handler.activator.HandlerActivator
 5: Bundle-SymbolicName: dictionary_handler
 6: Bundle-Version: 1.0.0
 7: Bundle-RequiredExecutionEnvironment:
 8:   CDC-1.0/Foundation-1.0
 9: Export-Package: org.dictionary.db.search,
10:  org.dictionary.handler.translate
11: Import-Package: org.dictionary.db.search,
12:  org.osgi.framework;specification-version="1.0.0"
```

In the import section (lines 11-12), package `org.dictionary.db.search` denotes the package with the interface pro-

DOI: 10.1145/1402521.1402526 http://doi.acm.org/10.1145/1402521.140256

vided by DB bundles (and required by the Dictionary Handler). In order to allow the Handler to execute independently of the availability of any DB bundle, this package is also listed in the export section (line 9). Another package exported is `org.dictionary.handler.translate`, which contains the interface provided by the handler.

In the proposed implementation, bundles are black box modules, i.e. their implementation details are kept hidden and clients can only access the bundles well-defined interfaces. Moreover, the proposed implementation follows a strictly layered architecture. For example, by correctly defining its manifest, it is impossible for a bundle in the Application Layer to bypass the Handler and communicate directly with DB bundles. And lastly, bundles are independent from each other, since package dependencies are solved with the mentioned export/import strategy.

4 Evaluation

In this section, we evaluate OSGi with respect to plain and old Java implementations. The evaluation is focused on qualitative requirements rather than on performance. The requirements considered are: extensibility and modularity.

4.1 Extensibility

It is expected that our motivating example can be dynamically extended with new language databases and client applications. In other words, it should not be necessary to stop and start again the system to include a new module.

However, in standard Java implementations, extensions can only be accomplished at compile time. For example, if a new dictionary becomes available, it would be necessary to change the handler's code and recompile the application. The reason is that the handler must import and instantiate each DAO, as exemplified by the following code:

```
import org.dictionary.search.spDAOimpl; // Spanish
import org.dictionary.search.enDAOimpl; // English
import org.dictionary.search.ptDAOimpl; // Portuguese
...
DAO esDAO = new spDAOimpl();
DAO enDAO = new enDAOimpl();
DAO ptDAO = new ptDAOimpl();
```

OSGi addresses this issue through the Service Registry. In our OSGi implementation for the Dictionary System, communication between layers is solely accomplished through services. Each layer provides a well-defined interface to communicate with its superior layer. Database bundles register the same service, under the same interface, so the Dictionary Handler can find all database services and bind to all of them. In order to distinguish the databases, a property denotes the language it represents. The same idea is used for communication between the Handler and client applications. The Handler provides a service that any client bundle can access in order to translate words. Under this perspective the system can be dynamically extended on both ends. If a new database becomes available, all it has to do is to register a service using the well-defined database interface. On the other hand, if a new dictionary client becomes available, it just need to retrieve the translation service provided by the Dictionary Handler

4.2 Modularization

Modularization is usually a challenge attribute in software projects [12]. When developers do not follow a well-planned modularization strategy, the result is usually a convoluted "spaghetti" code that hinders reusability and comprehensibility. This occurs because modules do not have well defined boundaries. In plain and old Java systems, developers do not have to compulsorily reason about modularization strategies, which usually results in applications that are organized from the point of view of developers, but are not modularized enough to foster reusability and other benefits that modularization can achieve. On the other hand, OSGi developers are forced to modularize their applications, since modules are key concepts in OSGi systems. However, it is important to mention that this can be a challenging task for complex systems.

In our Dictionary System the benefits from modularization are evident. Its layered architecture allows the assignment of precise responsibilities to bundles. It also minimizes coupling and maximizes cohesion. In this way, bundles can be independently implemented by different developers. Furthermore, developers can not erode the proposed architecture, for example bypassing layers.

5 Other OSGi Services

5.1 Declarative Services

Usually the implementation of a bundle must carry code to publish itself, which raises some issues for software architects. The first issue is that the publication of a service is not the main concern of a bundle, therefore it should be left off its code. There are also performance issues, since publishing a service with no clients consumes resources.

Declarative Services address such issues by providing means to specify a service statically, in an `XML` file. In this file, developers indicate which part of the bundle code is called when the service is requested. When the bundle becomes available in the system, OSGi just register the information provided in this `XML` file, without starting the module. When a client bundle requests the declared services, the module is activated, the requested method is executed, and after that the module goes back to its standby state.

In our Dictionary System, database modules were implemented using declarative services, since they are essentially stateless bundles. On the other hand, the Handler component was implemented using normal services, since it must

DOI: 10.1145/1402521.1402526 http://doi.acm.org/10.1145/1402521.140256

execute continuously to keep track of what databases are available.

5.2 Whiteboard Pattern

Instead of searching for a service, the availability of the service can be informed by the framework to a client bundle. This is achieved through the Whiteboard Pattern [10]. Typically a listener is used for each individual event. Since the objective is to track services, this approach is not adequated since the source could be unavailable upon the creation of the listener. The OSGi registry is thus the whiteboard to which all listeners may subscribe, and will be responsible to track all events, and notify any listener when it is necessary.

5.3 Advanced Service Filtering

The simplest way to retrieve a service is by its interface. For improved selectivity, a bundle can also provide a filter expression that is matched against the Service Registry properties. In our case study, a filter expression such as `!(language=English)` would match all database services except those that represent the English language.

6 Related Work

Well-known examples of components models include Microsoft Common Object Model (COM) [8], Enterprise JavaBeans (EJB) [16] and CORBA Component Model (CCM) [15]. However, such frameworks usually rely on a complex programming model, that requires developers to follow particular programming conventions. Moreover, they usually do not provide support to the dynamic update of modules. On the other hand, they provide support to distribution. However, there are efforts to create a distributed version of OSGi. For example, R-OSGi leverages concepts developed for the centralized management of modules to create distributed applications [14]. Using R-OSGi, developers can create OSGi applications without concerning about distribution. At deployment time, R-OSGi distributes the available bundles among network devices in a transparent way.

Eclipse is experimenting OSGi as a dynamic component model for its plug-in system, using the Equinox OSGi implementation [3]. Each plug-in installed in the Eclipse environment is a bundle. Thus, plug-ins can be installed and uninstalled without restarting Eclipse. Another well-known system that is planning to use OSGi is the Spring application framework for the Java platform [7].

7 Conclusions

OSGi provides a simple, lightweight, and dynamic component model for creating service-oriented applications. The system is gaining momentum in the consumer electronics market, and in enterprise and software development domains, as demonstrated by its use in the Spring and Eclipse projects. In this report, we have provided a gentle introduction to the basic features and principles behind the OSGi programming model and runtime system. In order to make the presentation more clear, we have illustrated the use of OSGi in a simple Dictionary System. The source code of this system is available at: http://www.inf.pucminas.br/prof/mtov/osgi_example.zip.

Acknowledgment: This research was supported by a grant from FAPEMIG.

References

[1] Apache felix. http://felix.apache.org.

[2] Concierge. http://concierge.sourceforge.net.

[3] Equinox. http://www.eclipse.org/equinox.

[4] Knopflerfish. http://www.knopflerfish.org.

[5] Oscar. http://oscar.objectweb.org.

[6] OSGi Alliance. http://www.osgi.org.

[7] Spring OSGi. http://www.springframework.org/osgi/specification.

[8] D. Box. *Essential COM*. Addison Wesley, 1997.

[9] R. S. Hall and H. Cervantes. An OSGi implementation and experience report. In *IEEE Consumer Communications and Networking Conference*, pages 394–399, 2004.

[10] P. Kriens and B. Hargrave. Listeners Considered Harmful: The Whiteboard Pattern. Technical report, OSGi Alliance, 2004.

[11] OSGi Alliance. *OSGi Service Platform: The OSGi Alliance*. IOS Press, 2003.

[12] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[13] J. S. Rellermeyer and G. Alonso. Concierge: a service platform for resource-constrained devices. In *EuroSys Conference*, pages 245–258, 2007.

[14] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed applications through software modularization. In *8th International Middleware Conference*, volume 4834 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2007.

[15] J. Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, 2nd edition, 2000.

[16] Sun Microsystem. Enterprise Java Beans specification (version 3.0), Dec. 2005.

DOI: 10.1145/1402521.1402526 http://doi.acm.org/10.1145/1402521.140256