Contents

Preface

Nowadays, software is a vital and inevitable part of the modern world.Presently, software is being used in many fields, including entertainment, financial and business or national defence and so forth, to simplify a large amount of hard work, reduce cost as well as increase productivity.

In fact, although software engineering has been improved and modernized continuously with so many new and advanced technologies and techniques, it is not able to completely satisfy the demand for the use of software. The problem above exists because a software system is always expected to be reliable, easy to change and expand, but has to meet all the requirements or rules of the real system which it represents, meanwhile the real system changes so quickly and unpredictably.

Actually, real systems tend to be much larger, more complex during their performances. Thus, to develop compact, understandable, high-quality software for such systems, appropriate methods and right strategies should be applied. Recently, a terminology has been mentioned as an approach to the mentioned problem, modularity - meaning a massive system will be divided into a number of parts or modules and the whole system can be seen as a combination of multiple modules.

Since many big and complicated enterprise software systems have been built using Java technology, Java is now said to be the most suitable platform for such systems. The fact is that Java platform provides a dedicated edition for that kind of development, Java Enterprise Edition or Java EE, which contains a set of standard technologies to support software developers. Despite the big success of Java EE, it is still criticized for the lack of modularity as systems become tremendously large and complex.

To solve the problem of modularity for Java software systems, a technology has been introduced, the OSGi technology, which offers a style of developing Java software systems in a modular way and it opens up a new

direction to building software applications.

With the devoted help and useful advice from Ph.D. Tran Thi Minh Chau at Software Engineering dept., Faculty of Information Technology, University of Engineering and Techonology, this report is written to introduce the OSGi technology and a demonstration application developed with OSGi.

Chapter 1

An introduction to OSGi

1.1 Java's limitations of modularity

Java platform, which was first introduced in 1995, has become highly popular. It is used to develop many kinds of applications, from mobile to desktop, enterprise. Java programming is highly object-oriented, it contains a number of good features, which are encapsulation, abstraction, inheritance, polymorphism. Those features make applications more dynamic and modular, compared to others used older technologies than Java. However, when applications become massive in size, more complex in logic, such features are not sufficient to scale these applications. The reason is that Java platform itself does not fully support modularity.

1.1.1 Java's encapsulation

With the principle of encapsulation, at class-level, Java have a mechanism to control visibility of classes and class members, which are properties and methods or classes themselves, by using access modifiers including public, protected, private or none indicating package level. This appears that we are able to create more independent classes, then the application tends to be more modular since it is made of many classes?

However, this level of encapsulation is only for low-level, individual classes. As classes are logically grouped together into packages that are typically used in Java organize code which is related in some way, this kind of encapsulation does not work well to make the whole application more modular. For code

to be used from one package to another, the code must be declared public or protected using inheritance.

Normally, a package uses code from different others, which means the implementation in outside packages must be exposed as public. So it turns out that when two packages communicate with another one, they have to know public APIs (Java interfaces) and private implementation as well. Which proves that packages are more dependent on each other the whole system is hard to change or less modular. The problem here is how two modules collaborate just by knowing a public APIs.

1.1.2 Java class-path

As mentioned above, Java code is usually placed into packages, then after being compiled, packages are also packed into JAR files which are a way to use code as libraries or parts of an application.

Since an application is composed by JAR files, when classes are about to be used, it must be located in a bundle of JAR files by the compiler to set up Java class path, which tells where the needed classes are. Unfortunately, a JAR file does not give any information about the code inside it, such as versions, dependencies. The process of finding classes in JARs to set up class path is completely trial and error, and it continues until the required classes are found. In fact, Java class path returns the first version of the code it finds.

To give an example, a Java class in an application requires a set of classes organized in a package called A, while package A is packed in JAR file 1 and JAR file 2 with two different versions and both files are declared in the application's class path, maybe use class-path variable or MANIFEST file. The question is which version of package A will be used, from JAR 1 or JAR 2, the fact is that if JAR 1 appears first in the class path, then undoubtedly package A in JAR 1 is going to be used. It proves that other versions of package A are overlapped by the one that first appears. Apparently, Java class path pays no attention to the logical version of the code or JAR files. This possibly leads to errors, like NoSuchMethodError exception, when a class from a JAR file interacts with an incompatible version of code from another. As a result, the consistency of source code is not ensured. Therefore, although a Java application is formed of JAR files, it does not has a highly modular architecture.

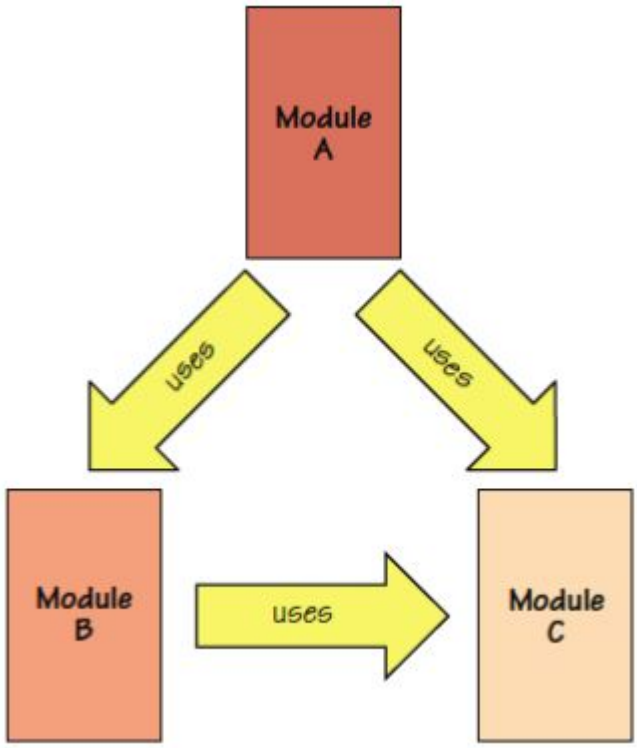

Figure 1.1: A large system divided into collaborating modules

1.1.3 What is modularity?

So what is modularity? And why it is so important to large applications?

The term modularity can be explained that a big system is broken in to a number of parts called modules. Each module contains a separate logical content or has a logical boundary, which means that a piece of code is inside only one module and not others. In addition, the details of implementation in a module are visible only to code that is part of the same module. Modules in a system coordinate and communicate in standard ways to make the whole system work.

Since modules all have a boundary, they are less dependent on each other, then replacing a module is easy and causes little effect on other parts. Therefore, the system is easy to change or expand, just by replacing or adding modules. Developing software modularily means writing complete modules and assembling them together, which would simplify development and improve

maintainability.

1.2 The OSGi framework

1.2.1 What is OSGi ?

OSGi is dynamic modular system for Java used to solve the problem of lacking modularity of Java applications. OSGi was first formed in 1999 but for a different purpose, now it is for building dynamic modular Java applications.

OSGi plays a central role for OSGi-based applications, it provides a run-time environment in which modules can be plugged in and interact with others. In fact, OSGi is a specification which defines and describes how OSGi-based applications work, the way OSGi modules interact; thus there are many implementations for the OSGi core framework, such as Apache Felix, Eclipse Equinox, so we can simply choose one to use.

At the present time, there are several non-trivial applications developed with OSGi platform, for example Eclipse IDE, one of the most popular Integrated Development Environments mainly for Java programming language and many others, or Glassfish Application Server v3, the one uses Apache Felix as its run-time.

1.2.2 Benefits of OSGi

So why is OSGi able to solve the problems that pure Java cannot?

OSGi reduces the complexity of applications by dividing them into many modules, but this is not the key factor that leads the applications to be more modular. OSGi goes further than that, it has some techniques to make modules less dependent on others, more encapsulated.

Fist of all, each OSGi module has a explicit logical version or code in a module is always assigned a clear version, which is convenient for managing and using code later.

Secondly, a module can explicitly exposes any part of code or none to the public, which means other modules only have the access to the part that is published. Typically, Java interfaces will be seen from the outside, but the implementation will not and two different modules still work well together, whereas we cannot achieve this with pure Java.

Thirdly, an OSGi module also have the control of which version of code it intends to use, which means that a module will refuse to interact with another unless the right version of the code it required is provided. This mechanism avoids the possibility of being inconsistent when incompatible code versions are given to a module.

Along with the above things, OSGi modules communicate in a standard way using exposed interfaces. With each public interface, a service is provided, which is simply a Java object and can be only used through the interface that the object is registered under. Therefore, modules in OSGi have loose relationships, fewer dependencies. This helps the whole system to be more dynamic, more modular, then easy to change, expand as well as develop.

Moreover, another remarkable advantage of OSGi is that removing, adding or replacing modules can be done on the fly or dynamically. There is no need stop a running system to perform the above operations.

Chapter 2

Fundamentals of OSGi

2.1 An overview of architecture

The OSGi framework's specification defines three conceptual layers, figure 2.1

- ***Module Layer*** - Concerned with packing and sharing code
- ***Lifecyle Layer*** - Concerned with managing modules
- ***Module Layer*** - Concerned with interaction and communication among modules

Each layer is dependent on the layers beneath it.

2.2 Module Layer

The module layer defines the OSGi module concept. An OSGi module is terminologically called*a bundle*

2.2.1 OSGi bundles

Basically, OSGi bundles are JAR files with extra meta-data. A bundle consist of Java classes, a meta-data file called MANIFEST in which every information related to the bundle is clearly defined, other resources like xml files or images are also packed into a bundle, figure 2.2

Figure 2.1: Three layers OSGi architecture

OSGi bundles are more powerful than normal JAR files because they are able to explicitly declare which internal packages are externally visible, exported packages. With standard Java, only classes have this mechanism by using keywords public, private or protected to indicate which class members can be used from outside classes, or which classes from one package are visible to others. Additionally, since a bundle always has a specific version, defined in MANIFEST file, all exposed packages are accompanied by the version of the bundle that they are from.

On the other hand, when a bundle needs to use code from other bundles, it also explicitly declares which packages with a specific version or a range of version must be provided, these packages are called imported packages. As long as the right code versions are unsatisfied, the bundle will not be able to deploy correctly.

The main benefit of explicitly declaring imported packages and exported packages for bundles is that the consistency of their versions is managed and verified automatically by the OSGi framework; hence, code versions are always guaranteed to be consistent before code is executed at run-time, which cannot be ensured with Java class path.

2.2.2 Bundle meta-data

All information of a bundle is included in the MANIFEST file. Normally, the MANIFEST file stores basic information about version, name, exported and imported packages of a bundle.

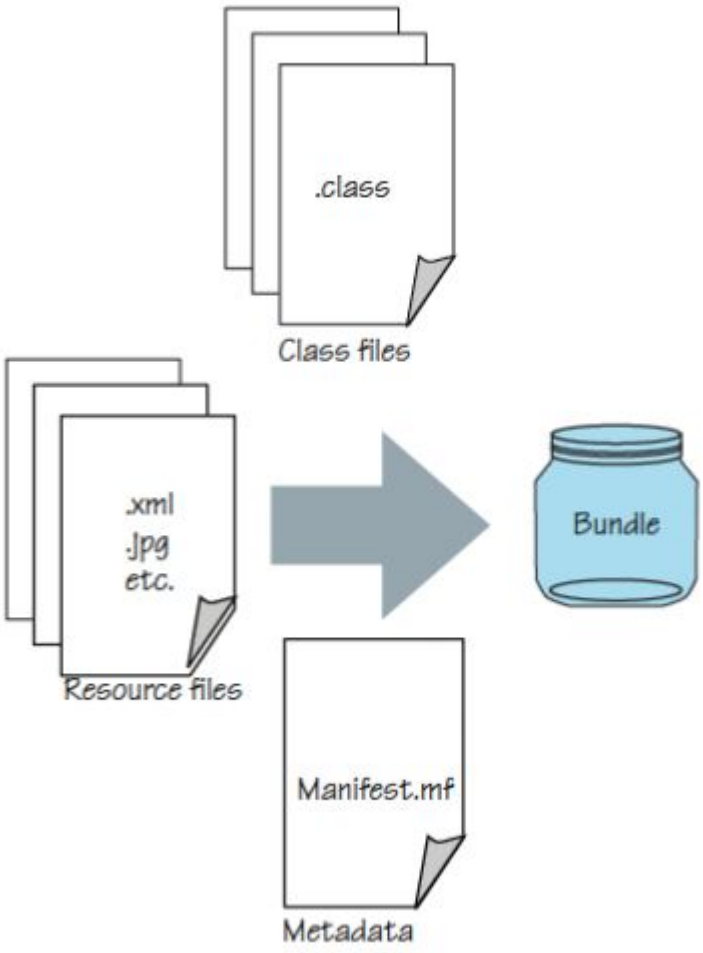

Figure 2.2: An OSGi bundle's composition

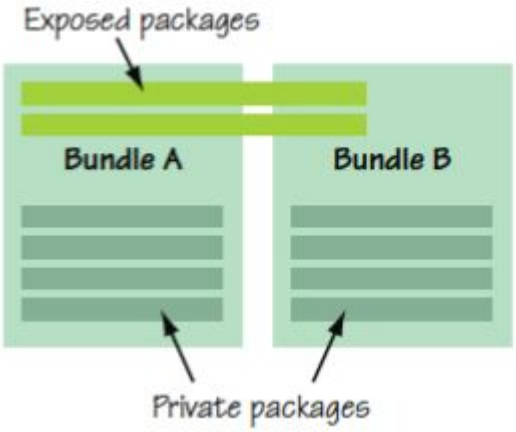

Figure 2.3: Exposed packages from Bundle A are visible to Bundle B, while unexposed packages are private

2.3 Life-cycle layer

bundle life-cycle

2.4 Service layer

2.4.1 Publishing services

2.4.2 Consuming services

2.5 Bundle activation

2.5.1 Bundle activator

2.5.2 Blue-print framework

2.6 Typical architecture of OSGi-based applications

Chapter 3

Example Application

Bibliography

[1] http://www.osgi.org/Technology/WhatIsOSGi

[2] Software engeering - 9th Edition, Ian Sommerville