FAKULTÄT FÜR !NFORMATIK

Modularization of Enterprise Applications - An Analysis using Enterprise-Technologies and a Module Management System

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

ausgeführt von

Julio César Vergara Heinrroth
Matrikelnummer 9227495

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gerald Futschek

Wien, 28.10.2008

(Unterschrift Verfasser) (Unterschrift Betreuer)

Eidesstattliche Erklärung

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder im Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quellen als Entlehnung kenntlich gemacht habe.“

Wien, 28. Oktober, 2008 Julio César Vergara-Heinrroth

Kurzfassung

Modul Management Systeme, wie z.B. *OSGi (Open System Gateway Initiative) Service Platform*, haben sich als Bausteine in der Architektur eigenständiger Anwendungen hervorragend etabliert.

Die OSGi Service Platform wird einen Meilenstein in der gesamten Java-Softwareentwicklung darstellen, so auch in der *Java Enterprise Edition*.

Bisher wurde der OSGi Service Platform nur geringe Aufmerksamkeit bei der Entwicklung von Unternehmenssoftware geschenkt. Der zentrale Inhalt dieser Arbeit behandelt die Zusammenführung von Unternehmenssoftware mit einem Modul Management System anhand der Java Enterprise Edition und der OSGi Service Platform.

Um die Rolle des OSGi in der Entwicklung von Java-EE-Anwendungen zu untersuchen, wurde eine Beispiel-Anwendung mit der OSGi Service Platform und Java-EE-Komponenten bzw. -Diensten erstellt. Durch den Einsatz in einer Anwendungsumgebung für Unternehmenssoftware, wie der Java Enterprise Edition, zwingt die OSGi Service Platform die/den Entwickler/in aufgrund ihrer Modul/Dienst-Architektur zu einer strikten Einhaltung einer modularen Softwarestruktur.

Die OSGi Service Platform kann schon jetzt eine grundlegende Rolle in der Entwicklung von Java-EE-Anwendungen in Bezug auf Versionierung, Bibliotheks/Modul-Abhängigkeitsverwaltung und Anwendungsverteilung einnehmen.

Abstract

Module Management Systems, like *the OSGi (Open System Gateway Initiative) Service Platform*, have been proven to be formidable architectural components in stand alone application.

The OSGi Service Platform will be a major keystone in the development of software across the board in the Java world, including the *Java Enterprise Edition*.

Little focus has been given to the use of the the OSGi Service Platform in the enterprise application development domain. The focus of this work is to study the integration of enterprise applications and a Module Management System using the Java Enterprise Edition and the OSGi Service Platform.

A sample application was developed using the OSGi Service Platform and Java EE components and services to investigate the role of OSGi with respect to Java EE applications.

The use of the OSGi Service Platform in an enterprise application environment, like the *Java Enterprise Edition*, forces developers to be discipline about modularization by introducing a *module/service-oriented* application development model.

The OSGi Service Platform can already play a fundamental role in Java EE applications with regards to application versioning, library/module dependency management and deployment.

Acknowledgments

I would like to express my heartfelt gratitude to my lovely wife, Rebecca, for being the keystone of my life and for being supportive and lovely to me at all times, good and bad.

To my daughters, Ally and Vivi, for putting a smile at all times and cheering and enriching my life in so many ways only they can.

To my parents, biological and in law, for supporting me at all times and their encouragement, which has kept me going.

I would also like to express my endless gratitude to my supervisor, Prof. Futschek, for believing in me and for always pointing me to the right direction.

Finally, to all my friends for showing me the Paraguayan, Austrian and American way of life.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Many efforts have been made in the software engineering field regarding the modularization and componentization[1] of software.

What are *components*? According to [ASKOXF] a component is defined as "a part or element of a larger whole," derived from the Latin word "componere," which stands for "put together." What are *modules*? According to [ASKOXF] a module is defined as "each of a set of parts or units that can be used to construct a more complex structure."

In this master thesis we will, however, rely on a more technical definition of the term. We will refer to a module as "a program unit that is discrete and identifiable with respect to compiling, combining with other units and loading." [IEEEGL] Moreover, according to [IEEEGL], the terms *module*, *component* and *unit* are often used interchangeably or are defined to be sub-elements of one another depending upon the context. In fact, the assumption of a specific context becomes even more important as different software and hardware architectures and frameworks tend to use the notion of a component/module in different ways. An example of such is the Java Enterprise Edition architecture which defines a module as a collection of components. In this work, we will examine the use of the word component and module in different ways in the contexts of software application. Regardless of context, one definition seems to be common: *a discrete program unit that is individually identifiable.*

Having defined these words formally, software developers and software architects have understood early on that partitioning applications plays a very important role in software development. In fact, modularization and componentization efforts can be found in all areas of software development, like programming languages, application design (use of design patterns) and even architecture. Having said that, in the context of application development, the notion of modules and components is almost indistinct and pervasively used. However,

1 Componentization is not an English word but is used in this context as a term for component-based development, breaking software application.

the meaning of the terms varies based on the context. In fact, [GRÖNE] raises awareness of this problema and provides a definition for:

- *System component*: “a component is an active part of the (abstract) system which exists at runtime; a component provides a defined functionality and communicates with other parts of the system. All parts of the system may have their own state.” This definition based on a look at a system with its behavior and runtime structures, which he calls the *system view.*

- *Software component*: “a component is a deployable software unit which is relevant at build–time (for example a library), or which may be loaded into memory at runtime and be processed by a processor or a virtual machine.” This definition is based on a software development point of view.

Furthermore, [GRÖNE] concludes that clear separation and definition of component terms is key to resolving the ambiguous use of the terms. In this work we use the later definition of component, namely *software component.*

In fact, the term *Component-Based Software Development* exists as an approach to software development, where a software component - according to the above-mentioned definition – advertises the services it provides and may be organized into repositories. These software components can be assembled together to create large systems. These systems' functionality is based on the overall functionality of each of these components and their cooperative interaction.

Having said that, a common understanding regarding modularization is that breaking an application into manageable pieces is key because of the following benefits:

- *Easier application management* is a benefit, because each individual piece can be managed independently; bug fixes, extension, improvements and modifications to modules can be done separately.

- *Redundancy avoidance* can be achieved by defining clear-cut modules with specific areas of concerns and cohesive responsibilities.
- *Easier testing* of each individual module, which can be tested independently, hence, creating more robust software. Early studies have already suggested that overall highly cohesive modules have a lower fault-rate than low-cohesive modules [CARD]
- *Software reuse:* individual modules can be reused in the assembly of many applications. Third party modules can be integrated.
- *Shorter development life-cycles*: because one could just buy a module or integrate freely available ones into an application. This plays a particularly important part in the success of new programming languages like Perl, Java, C#, etc. Moreover, it has allowed these languages to grow tremendously in past years in terms of capabilities, even beyond the boundaries of their standardization bodies. An example of such is Java which originally was almost solely standardized by the Java Community Process [JCP]. However, new very powerful initiatives exist from the Apache Software Foundation [APACHE], Eclipse [ECLIPSE], Source forge [SFORGE], Spring [SPRING], and others providing an additional array of products and libraries for the Java language that are extremely popular. Some of them have become widely accepted and are fundamental pieces of software used in many organizations.
- *Project Management flexibility*: since the applications are broken down into pieces, individual sub-projects can be assigned to different developers and/or development groups. An additional benefit also becomes clear in terms of human-resource management and knowledge-based human-resource project distribution. For instance, if an application uses many different technologies, modules can be assigned to developers and/or development groups with the knowledge and expertise in that specific technology.
- *Simplicity*: developers using modules and components really only need to know about the functionality of the services provided by these modules without any specific knowledge of the internals of these modules.

The above mentioned list of benefits is not intended to be a comprehensive list but it should rather point to some of the benefits of modularization and componentization.

A good example of an effort in modularization is the Java Enterprise Edition (Java EE), which provides architectural modularization and componentization of enterprise applications in the Java domain. One can also use it as an exemplification of the ambiguous use of the term component: web component, business component.

Moreover, the software engineering discipline has been so focused on these modularization efforts that it has produced a new kind of emerging system, namely *Module Management Systems.* Many modularization/componentization systems have been developed, but there is the emergence of a dominant standard in the Java domain for this kind of system, the *OSGi Service Platform* standardized by the OSGi Alliance - Open Services Gateway Initiative[2] (OSGi). This emerging module management system for Java introduces its own definition of a module different from the Java EE module definition, but also provides a different paradigm for software development, namely module-oriented/service-oriented development combination, where the complete software development effort is based around the notion of modules and the services they provide.

Also, in the component-based and module-based software engineering domain, most of the effort has been placed on the creation, definition and specification of components, modules and their architectures.

Additionally, in the research community most of the research regarding OSGi has been used in the context of home automation and embedded systems environments. Little to no attention has been provided on the use of OSGi in the enterprise as a platform for developing enterprise software.

2 The full text definition of OSGi is considered irrelevant [OSGIAL].

Consequently, there is the need for more investigation regarding the use of a module management system in the enterprise and the benefits of a module management system like OSGi bring to the enterprise application development problem domain. This work aims to shed some light on the use of OSGi in enterprise application programming and the benefits of it.

2 Motivation

There are so many programming languages and application programming frameworks for each different programming language that selecting one nowadays can be perceived as a matter of preference.

Why enterprise Java?

Java, although relatively new compared to C, C++ and other programming languages, has become a very powerful language with many capabilities, including many Application Programming Interfaces (API) and libraries to facilitate the development of many different types of applications.

In the Java domain there are three main distributions:

- the **Standard Edition (SE)** providing APIs for the development of stand-alone application and Applets – Java applications that run inside a browser,
- the **Micro Edition (ME)** providing APIs for the development of applications in embedded devices such as Portable Data Assistants (PDAs), phones, beepers and other devices with limited capabilities,
- the **Enterprise Edition (EE)** providing APIs for the development of server-side applications – Java applications that are deployed and run inside a server environment, where the server provides life-cycle management and services to the applications deployed in it.

The Java Enterprise Edition distribution has established itself as a major platform for business applications since the introduction of component based development models in the late 1990s. These component-based models for software development assume the presence of a middle ware server (application server) that provides the runtime environment and services to these components. Java EE is a specification by SUN that aims at standardizing these component development models as well as the services they provide. Their paramount goal is to release the developer from the intrinsic complexity of these services, thus releasing developers from unneeded complexity and enabling a much easier and efficient development process.

Why OSGi?

The origins of OSGi trace back to the home automation market as a means to:

- standardize the integration of devices and applications made for end users and provided by operators and service providers. In other words, it standardized the integration of application modules to allow different devices to work in a home. These standardizations included the cooperation and integration from devices and application modules to manage these devices under the assumption that applications, as well as devices, could be provided by different vendors.

- provide an execution environment for the defined application modules, where the execution environment is in full charge of the life-cycle and management of these application modules and devices.

Having said this, one of the key factors stressed by OSGi is the dynamic integration and collaboration of application modules in a service-oriented fashion, where modules(components) utilize other modules' services and publish their own services that can in turn be used by other modules.

Motivation

OSGi gained traction in the Java SE world with the adoption of the Eclipse Foundation of OSGi to provide module management to the Eclipse Platform. This contributed to the management of Eclipse plug-ins, namely modules for the Eclipse Platform. Eclipse is one of the major development tools for applications in Java and many enterprise application developers use it on a day-to-day basis. The inclusion of OSGi in Eclipse exposed a lot of developers to the capabilities offered by OSGi; however, until only recently OSGi was largely ignored in the enterprise application domain. The author believes that OSGi will be a major keystone in the development of software across the board in the Java world not only in Eclipse but also in the enterprise application programming domain.

Having said that, substantial research and knowledge exists regarding the use of OSGi in stand alone applications, in particular in combination with Eclipse. Additionally, there are studies that emphasize the use of OSGi in embedded system[LEGOOSGI]. However, the use of OSGi in the Enterprise Edition environment has been lacking/neglected due to the fact that many enterprises that use Java EE wait for their server vendor to dictate the use of specific technologies. Server vendor compliance to Java EE standards is extremely difficult due to an extensive list of APIs needed to be fulfilled [JEETECH]. In fact, there are just a few vendors that are fully compliant with the latest version of the Java EE [SSAPPMTX].

Java EE developers have understood since early on that although the Java EE distribution provides a developer with many features, it is not sufficient, as requirements and complexity for enterprise applications are high. Consequently, almost every Java EE development project will resort to using features outside the Java SE and Java EE specification. Examples of such are Web Frameworks, which are not considered part of Java EE but are a substantial part of many Java EE development efforts.

Java EE has provided for many years now a reliable software platform for enterprise applications. However, since the specification does not state any specifics regarding the implementation of the platform, many different application servers exist, which lead to several implementations where each server has a different approach to application management and deployment.

Despite some of the deficiencies in the Java EE specification, the Enterprise Edition of Java has established itself as one of the most widely used component-based enterprise application development environments with thousands of deployments available at the time this work was written [EVANSDC].

Although originally the Java EE platform was viewed as a development platform for large scale enterprises, in recent years due to the introduction of easy to use features, Java EE has made substantial inroads in the small businesses sector as suggested by this industry study [EVANSSM].

Consequently, Java EE and OSGi both provide a component based development model to application development. While OSGi development model is generic, Java EE development model can be considered specific to a technology, for instance: a *Servlet* – server-side dynamic web application - is built around the notion of the HTTP protocol and as an extension to HTTP servers.

The integration of Java EE and OSGi will introduce changes in the overall architecture of enterprise applications and in particular in their programming models. As a paradigm shift from a technology-oriented approach used in Java EE to a service-oriented approach use in OSGi.

Integrating OSGi and Java EE presents not only server vendors with new challenges and benefits, but also to application developers.

This work sheds some light on these challenges and benefits from the application developer's point of view.

3 Goal and Structure

The author believes the OSGi Framework will be a major keystone in the development of software across the board in the Java world. The introduction of the OSGi Framework in the Java EE domain will introduce changes in the development process as well as the architecture of enterprise applications.

This thesis studies the realization of service-oriented, component-based development model and process for Java EE applications with the use of existing technologies and tools.

The aim of this thesis is to try to evaluate the integration of Java EE and OSGi from an application developer's point of view rather than a server vendor's point of view. In particular, this work analyzes how these two technologies complement each other to create a coherent enterprise application. Additionally, this thesis introduces a possible and practical way to achieve the integration of OSGi in enterprise applications to achieve a simpler application with the decomposition of enterprise services to achieve a more lightweight application.

Further, the evaluation of the integration of Java EE and OSGi is done with respect to the following criteria:

- Simplicity and efficiency
    - Programming model: how to bring into agreement the different programming models
    - Packaging: evaluated different packaging structures
    - Deployments: test different deployments
    - Management: administration of components
- *Testability* of components and integration test
- *Extensibility* of components and the application framework

Additionally, the following aspects regarding the integration of Java EE and OSGi will be considered:

- *Benefits* of the integration of OSGi with Java EE
- *Limitations* of Java EE as well as OSGi

Chapter 4 provides readers with sufficient background information regarding the relevant technologies involved in the integration. In particular, Section 4.1 provides background information on the modularization units in the Java programming language. Section 4.2 provides a overview of the class loading mechanism in Java. Section 4.3 introduces the Java EE architecture, the componentization and modularization units in the Java EE architecture. Section 4.3 also outline the services available in a Java EE platform.

Section 4.4 provides a thorough introduction of the OSGi Service Platform and the composition of the platform. Additionally, all features in the OSGi Platform are explained. As the goal is to provide enough detail to allow understanding the integration of OSGi with Java EE

Chapter 5 introduces the sample application used to study the integration capabilities. As well as the motivation behind the architectural structure selected the sample application in this study. Finally, the lessons learned during the application elaboration of the sample application is presented.

4 Technology Background

The intention of this module is to provide the reader with sufficient background information regarding the major technologies involved in this paper.

This chapter is divided into four sections:

1. Section 4.1 explains the modularization units of Java
2. Section 4.2 explains the class loading features of Java
3. Section 4.3 provides an overview of the Java EE distribution in terms of architecture, modularization/componentization, development model
4. Section 4.4 provides a thorough review of the OSGi Service Platform

4.1 Modularization Units in Java

The Java programming language like many other programming languages comes with some notions of modularization.

Classes and Objects

Provides the object-oriented modularization units, enclose attributes (fields) that describe classes(objects) and behavior (methods) that provides action on the classes(objects).

Java Archive (JAR) File

A JAR file [JAROVER] is a platform-independent file format that aggregates many files in one. A Jar file is the primary unit of deployment for applications and libraries – classes and resources within them .

The file format for JAR files is a standard ZIP format, which supports compression to reduce size.

A JAR file contains a `META-INF/` folder that can contain additional information regarding the classes and resources within it by means of a `MANIFEST.MF` [JARSPEC] file, which provides meta information regarding the files within the JAR file.

Individual entries in the JAR file can also be digitally signed to authenticate their origins. Signature files and blocks are also placed in the `META-INF/` folder.

The mechanism for extending the Java platform uses the JAR file format to package extension classes[3]. Manifest attributes are available to support the extension mechanism and related features such as package sealing and package versioning.

Java Packages

Every class in Java exist in a specific namespace. This namespace mechanism in Java is provided by packages. Since packages provide namespaces for classes, packages are used to resolve potential name conflicts[4].

Many classes can belong to the same package. Packages can be considered a modularization units in Java.

Also, packages allow access to non-private fields and methods for classes; this allows related classes to be grouped together, without having to show details over the internals of the package. This privilege is not extended to other packages or sub-packages, because Java does not provide cross-package privileged access.

3 a group of packages housed in one or more JAR files that implement an API that extends the Java platform

4 The naming convention for packages is the reverse Internet domain name [JAVACONV].

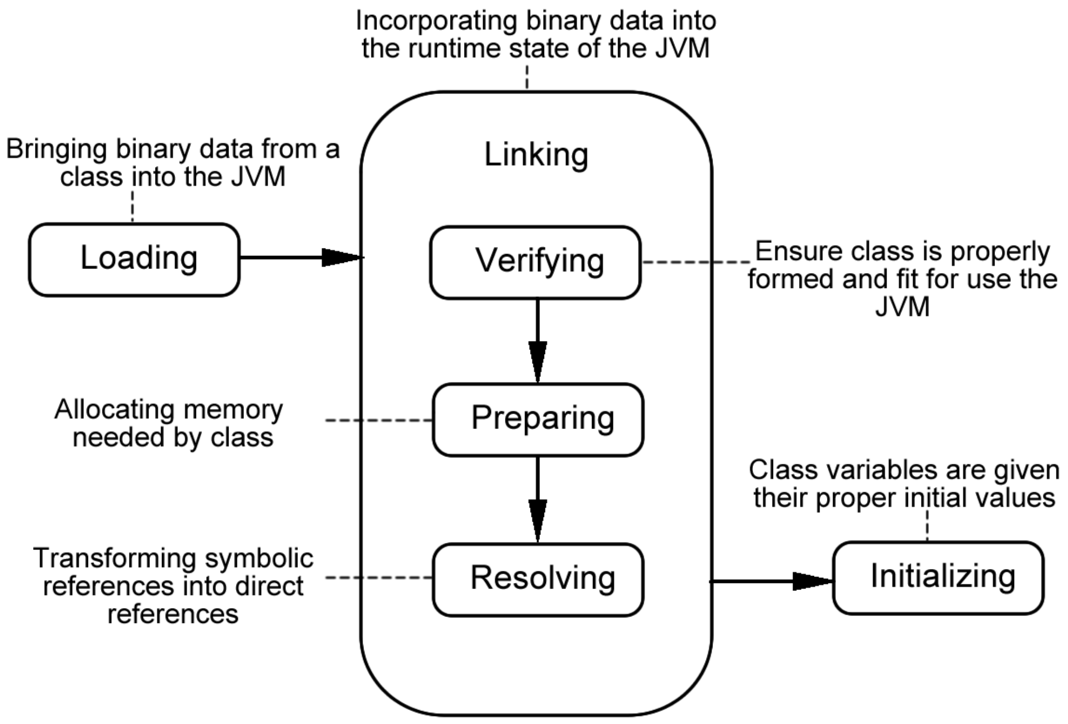

Incorporating binary data into the runtime state of the JVM
Linking
Bringing binary data from a class into the JVM
Loading
Verifying
Ensure class is properly formed and fit for use the JVM
Allocating memory needed by class
Preparing
Class variables are given their proper initial values
Transforming symbolic references into direct references
Resolving
Initializing

Figure 1: The Class Loading Process in Java

Class Loaders

In Java, class loaders are the components responsible for the dynamic linking and loading of byte-code class definitions to create runtime Class objects. See next section for a detailed discussion on class loading.

4.2 Class Loading in Java

In programming languages such as C/C++ the compilation process creates machine binaries. These machine binaries need to be assembled altogether into a final executable program or library in their own right. This final step required in the application assembly is the linking process, which merges codes from separately compiled sources along with other (shared) libraries into an executable application or library of its own.

In Java, although the steps to application assembly are similar, the approach is different. For every class declaration in a Java compilation unit - `.java` file, a class file - `.class` file - is created containing the Virtual Machine instructions called the bytecode. Hence, a class is the unit of software distribution in Java. There is no need to link these class files at compilation. However, during the compilation process the dependent classes must be available in order for the application to compile. Unlike the aforementioned programming languages, the linking process in Java is performed dynamically at runtime.

Having said that, c*lass loading* refers to the steps a JVM takes to make a class available at runtime. Figure 1[5] outlines the 3 steps of class loading: *physical loading, linking and initializing* [HOLBREICH].

1. *Physical loading* refers to the process of physically locating the class files, reading it and loading the contained bytecode. This is a process performed by class loaders.

2. *Linking in Java* refers to the process of:

    1. *Verifying the bytecode:* it ensures the binary representation of a class[6] is structurally correct, i. e. valid operation codes, instruction branches and others according to [GOSLING].

    2. *Class preparation:* allocating memory needed for building the necessary data structures to accommodate the class definition, members, methods, implemented interfaces. Building the necessary data structures.

    3. *Resolving the remaining symbolic references used within the class*, i.e. super classes where applied, types of fields, types of method signatures, types of method parameters, types of constructor parameters and types of variables used in methods and constructors. At this point, additional class loading is performed to resolved those sym-

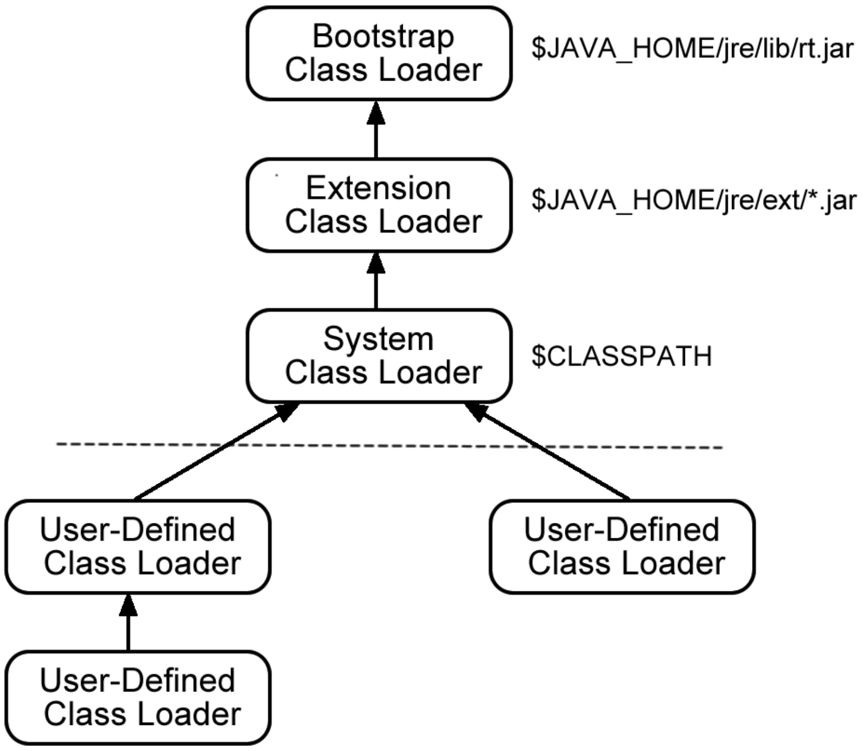
5 This figure is a reproduction of [HOLBREICH].
6 The term class is used generically to refer to either a class or an interface.

bolic names.

The whole linking process is performed by the JVM and upon resolution of the linking process, two classes are finally linked to each other.

3. During the *initialization phase* any static initializers - `static { /* ....*/ }` - are executed and static fields are initialized to their default values. Immediately after the initialization phase, the class becomes available for use.

As stated before, the process of loading classes is performed by *class loaders*. And as their name states, they are responsible for loading classes into the JVM. Every class loader has a policy for looking for classes as different class loaders may implement their own policies. Class loaders are organized in a tree like structure, see Figure 2 [HOLBREICH].


Bootstrap Class Loader
$JAVA_HOME/jre/lib/rt.jar
Extension Class Loader
$JAVA_HOME/jre/ext/*.jar
System Class Loader
$CLASSPATH
User-Defined Class Loader
User-Defined Class Loader
User-Defined Class Loader
Figure 2: Hierarchical Class Loading in Java

Upon request, a class loader checks to see if the class is already in its cache. If so, it returns the class object. If not, it delegates to its parent to load it or tries to load the class itself. Hence, the class lookup process goes in the following order: cache, parent and self.

According to [LIANG] the major advantages of the dynamic class loading system in Java are:

1. *Lazy loading*. Classes are loaded on demand, which allows the system to delay the loading of classes as much as possible, reducing memory footprint as much as possible.

2. *Type-safe linkage*. Dynamic class loading must violate the type-safety of the Java Virtual Machine. Dynamic loading must not require additional runtime checks. Link-time checks are acceptable, because they are performed only once.

3. *User-definable class loading policies*. Developers have complete control over the class loading process. Hence, a developer is able to define his own class loaders to load up classes from wherever he wants, locally, remotely, from a stream, etc.

4. *Multiple namespaces*. Class loaders provide separate namespaces for different components. Hence, two completely different classes with the same fully qualified name can be loaded in parallel using two different class loaders. Furthermore, the class type is uniquely identified by the combination of fully qualified class name and class loader used to load the class.

Multiple class loaders can be used within a single JVM and these class loaders can cooperate with each other. For instance, class loader CL1 can ask CL2 to load class C1. This relationship between class loader is called the *delegation relationship [LIANG]*.

Hence, some value-added features of class loaders are *code reloading*: which allows someone to update existing components/classes at runtime without a restart; *bytecode instrumentation*: which allows a class to be modified prior to being available for usage at runtime[7].

In short, class loaders can be considered a powerful tool for developers as well as administrators to managing software components such as server-side application components, applets and others.

7 Bootstrap classes cannot be instrumented as they are loaded by the *System Class Loader* and class loaders can only instrument classes loaded by itself.

4.3 Java EE Overview

This section aims to provide sufficient background information regarding the Java EE componentization and modularization model.

Java EE is the enterprise distribution of Java. Its major goal is to simplify the development of enterprise applications by providing a multi-tier architecture as well as modularization and componentization of software units. Furthermore, Java EE aims at standardizing these enterprise software components and the services these software components may use.

In particular, Java EE emphasizes on the development model of these components by providing specific APIs for the creation of components and the use of Java EE services

Java EE assumes the presence of a Java EE server, which provides an implementation for the different APIs in the distribution.

These Java EE APIs have two parts:

1. the *Service Provider Interface (SPI)*, which mandates what a Java EE product vendor should support, and

2. a *Client API*, which is what Java EE developers use to create components and use services.

This two-side approach allows developers to develop components and use Java EE services without knowledge of the real implementation, that is running their components, or providing the Java EE services. This approach also aims at minimizing dependencies with specific implementations of Java EE services and so keeping great degree of platform independence regarding not just the hardware architecture and the operating system but also independence regarding the Java EE server implementation.

All Java EE components must be assembled into modules. For instance, web components and web resources must be assembled into a web module. The module type used for deploying components depends on the type of Java EE components to be deployed. For instance, web components can only be assembled into a web module and never into an application client.

Architecture

The Java EE architecture is a multi-tier logical architecture; hence, the Java EE specification [JAVAEESPEC] does no imply any physical partition of architectural elements in specific physical machines, process, virtual machines or address space. The physical partitioning of the application is left out to the software architect/designer, which assumes at the same time the presence of built-in distribution in case components are distributed across many physical machines. This distribution mechanism must be completely transparent to all components.

The specific implementation of a distribution mechanism for Java EE applications/components is left out to the Java EE product vendor who also provides the implementation of Java EE APIs in an application server[8]. The Java EE specification does not mandate any specific architectural organization of an application server. However, the specific set of functionality as well as the runtime behavior for all components and modules that contain them are standardized. Additionally, the Java EE specification does not mandate a specific implementation for Java EE services either; again, leaving the implementation to application server vendors.

The Java EE architecture introduces the notion of containers. A *container* refers to a Java EE server implementation without dictating a specific architectural reference or guideline to server vendors. Containers are provided by Java EE servers.

8 Application server is a term for Java EE server implementation.

In essence, containers provide two fundamental features to Java EE components:

1. *Runtime Environment* – by adhering to a specific component's life-cycle
2. *Services* to Java EE components such as database access through Java Database Connectivity (JDBC)

Important is to note that different component types use different types of containers, Contracts also vary based on the the type of containers used.

Figure 3 shows a simplified view of the Java EE Architecture after [JAVAEESPEC], as well as the different types of components and their corresponding containers.

It is important to know that nobody can directly access components. If an application wants to access a component, it must do so through the corresponding communication protocol and the container.

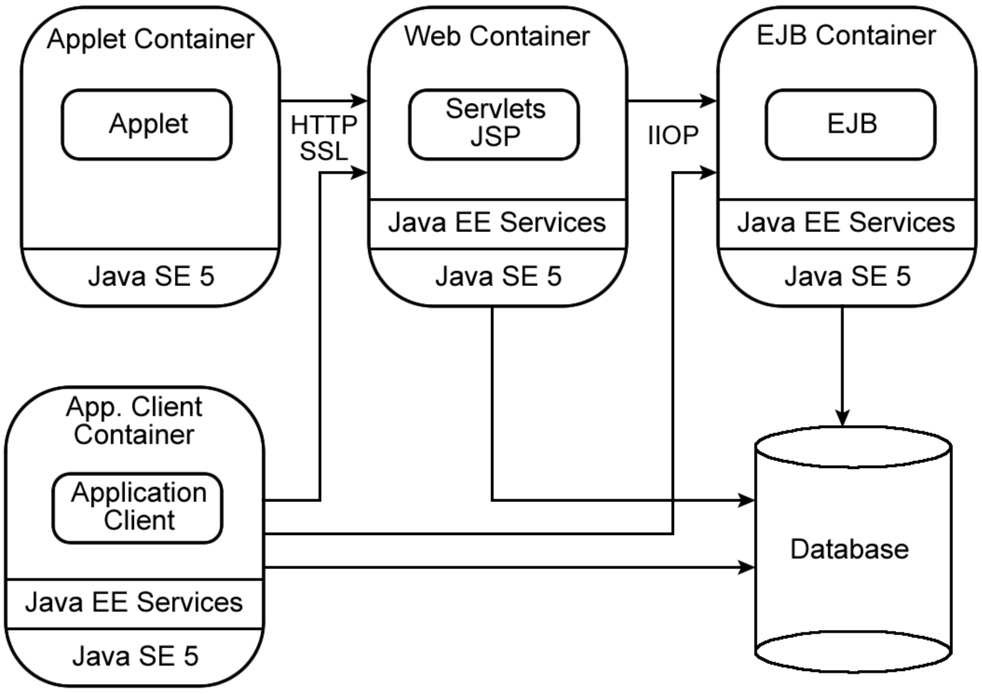


Figure 3: The Java EE Architecture

Components and Modules in Java EE

Figure 3 outlines four different types of components in the Java EE architecture.

1. *Application Clients* – are regular Java applications, either GUI-based or console-based – that run inside an application client container that provides access to Java EE services to the application client. The collection of all classes and resources that encompassed the application client is referred to as an A*pplication Client Module.*

2. *Applets* – are GUI applications that run inside a browser, where the browser provides the runtime environment to the applet. Browsers do not provide any Java EE service to applets, and, therefore, should not be considered a module in Java EE. However, applets can communicate to the Java EE environment using standardized interoperability protocols See section on interoperability.

3. *Web Components* – are components that respond to Hyper Text Transfer Protocol (HTTP) [RFC2068] requests from clients. Servlet, Java Server Page (JSP), filters, Web event listeners are examples of web components in the Java EE platform. These components execute inside a web container, which assumes the presence of a web server. The web server receives HTTP requests and dispatches these requests to the web container, which in turn dispatches to the corresponding web component. The collection of all web components – Servlet, JSP, filters, web event listeners – as well as web resources such as Hypertext Markup Language (HTML) pages, Cascading Style Sheet (CSS) files, Java Script (JS) files and others are all collectively called a W*eb Module.*

4. *Enterprise Java Beans (EJB)* – are server-side Java objects responsible for business logic. EJBs execute inside EJB containers, which in turn provides a transactional context to EJB components. EJBs are remotely

accessible using Internet Inter-Object Request Broker Protocol (IIOP) [IIOP]. IIOP is a distributed object technology, that allows method invocations across networks. A discussion on IIOP is beyond the scope of this work and is subject to further research and development. The collection of all EJBs as well as classes and resources needed by them is to be considered an *EJB module – Business Logic Module*.

5. *Resource Adapters* – are software components that provide connectivity to external resources - systems outside the Java EE environment. They can be used for two purposes: 1) to extend the functionality of a Java EE service, such as JDBC, or 2) to provide connectivity to an external system, for instance a mainframe environment. All classes that comprise the resource adapter as well as the resources they need are considered a *Resource Adapter Module*.

6. *Databases* – a database accessible through JDBC is required by the Java EE platform for storing business data. This database is accessible to all components. The Java EE specification does not mandate a specific DB. It solely mandates that the database must be accessible through Java Database Connectivity (JDBC) API. The database is not considered a module in the Java EE sense. However, it is viewed here as a key architectural component.

Modules in Java EE are determined by the deployment structure (packaging structure). Also, as mentioned earlier, specific types of components can only be assembled in specific types of modules, for instance web components can only be assembled into web modules. See section on deployment structure.

Java EE Services

The Java EE Architecture mandates the presence of the following standard services[9].

9 Some of the Java EE services are part of the Java SE distribution.

- *HTTP* – Servlet and JSP provide a server-side implementation for handling HTTP requests. Additionally, the client-side API in the `java.net` package can be used for client-side applications. Java EE assumes the presence of a web server.

- *HTTPS (HTTP over Secure Socket layer (SSL))* – is supported in the same fashion as the HTTP service. Java EE assumes the presence of HTTPS access to the web server.

- *Java Transaction API* (JTA)– enables transaction[10] support. The Client API is used by developers to demarcate (begin/end) transactions. Java EE assumes the presence of a transaction manager. The SPI interfaces are provided to server vendors to regulate the binding of resource managers from transactional systems (such as database managers) to the transaction manager steered through the Client API by developers.

- *Remote Method Invocation (RMI) over IIOP (RMI-IIOP)* – enables Java EE applications to work completely within the Java programming language using the Java Remote Method Protocol (JRMP) as the transport, or work with other CORBA-compliant programming languages using the IIOP. EJBs are RMI-IIOP server objects and, therefore, they can be accessed from Java/Non Java applications using IIOP to ensure interoperability with existing CORBAR environment. For a full reference, please see [RMI-IIOP].

- *Java Interface Definition Language (IDL)* – allows Java components to access CORBA objects using IIOP, which can be written using any other language supporting CORBA.

- *Persistence Service – Java Persistence API (JPA)* provides a standardization of *Object Relational Mapping(ORM)* tools.

  ORM tools:

10 A transaction is defined as an logical unit of work with ACID characteristics: Atomicity, Consistence, Isolation and Durability [RAMA].

  - enable the mapping of Java classes to tables in a relational database
  - provide basic Create, Read, Update and Delete (CRUD) operations
  - provide a querying facility to search the database

  Again, the Client API is what developers use to develop persistent applications. Vendors provide implementations of JPA. These implementations of JPA are called *Persistent Managers.*

- *Messaging service – Java Message Service (JMS)* enables access to messaging servers using Point-To-Point semantic or Publish-Subscribe semantics. Messaging servers are transactional systems that enable the submission/reception of messages in an asynchronous fashion, thus, providing developers with an asynchronous programming model. Java EE assumes the presence of a messaging server and provides a SPI to integrate it with the Java EE server. Developers use the Client API to develop JMS based applications to access the messaging server.

- *Naming / Directory service – Java Naming and Directory Interface (JNDI)* enables access to a naming/directory servers. *Naming servers* provide translation of human readable names to objects in Java. *Directory servers* provide naming services as well as the ability to attach properties to the stored names. Java EE assumes the presence of a naming server and provides the SPI to integrate it with the Java EE server. Developers use the Client API in JNDI to access or publish objects stored in the naming server.

- *Electronic mail service – JavaMail* enables the submission of email messages from within a Java EE environment. Java EE does not assume the presence of either a Single Mail Transport Protocol (SMTP), Post Office Protocol or Internet Mail Access Protocol (IMAP) server. It merely, provides the interfaces to bind one of those servers into a Java EE server environment and a Client API for accessing the email service.

Java Activation Framework (JAF) provides a framework for handling data in different Multi-Purpose Mail Extension (MIME) types. It's required by Java Mail.

- *Security service – Java Authentication and Authorization Service (JAAS)* allows the authentication of users within a Java EE server and allows control user access to Java EE components and services. It is a Java realization of the *Pluggable Authentication Module (PAM)*, which allows the use of authentication and authorization services independently from the real implementation of these services. In other words, it enables developers to write secure applications without knowledge of the authentication/authorization mechanism in use.

- *Web Services* – allows web access of XML-based services as well as the deployment of XML-based services from web components and EJB components.

- *Management* – Java Management Extensions (JMX) enables the writing of *Management Beans (Mbeans)*, which can be used to manage and monitor resources such as applications, devices, services, and the Java virtual machine. In Java EE JMX can be used to manage components, such as EJBs, and the server infrastructure, such as JDBC data sources.

Deployment Structure and Application Assembly

Java EE provides a portable deployment structure for enterprise applications. This Deployment structure is also the packaging structure. Components are assembled into different deployment structures based on their type.

The basic unit of deployment in Java EE is the Module. One module can be comprised of one or many components. Additionally, a Java EE module may have a deployment descriptor that lists all the components in the module. These deployment descriptors are optional as deployment information may be annotated into the corresponding classes since Java 5.

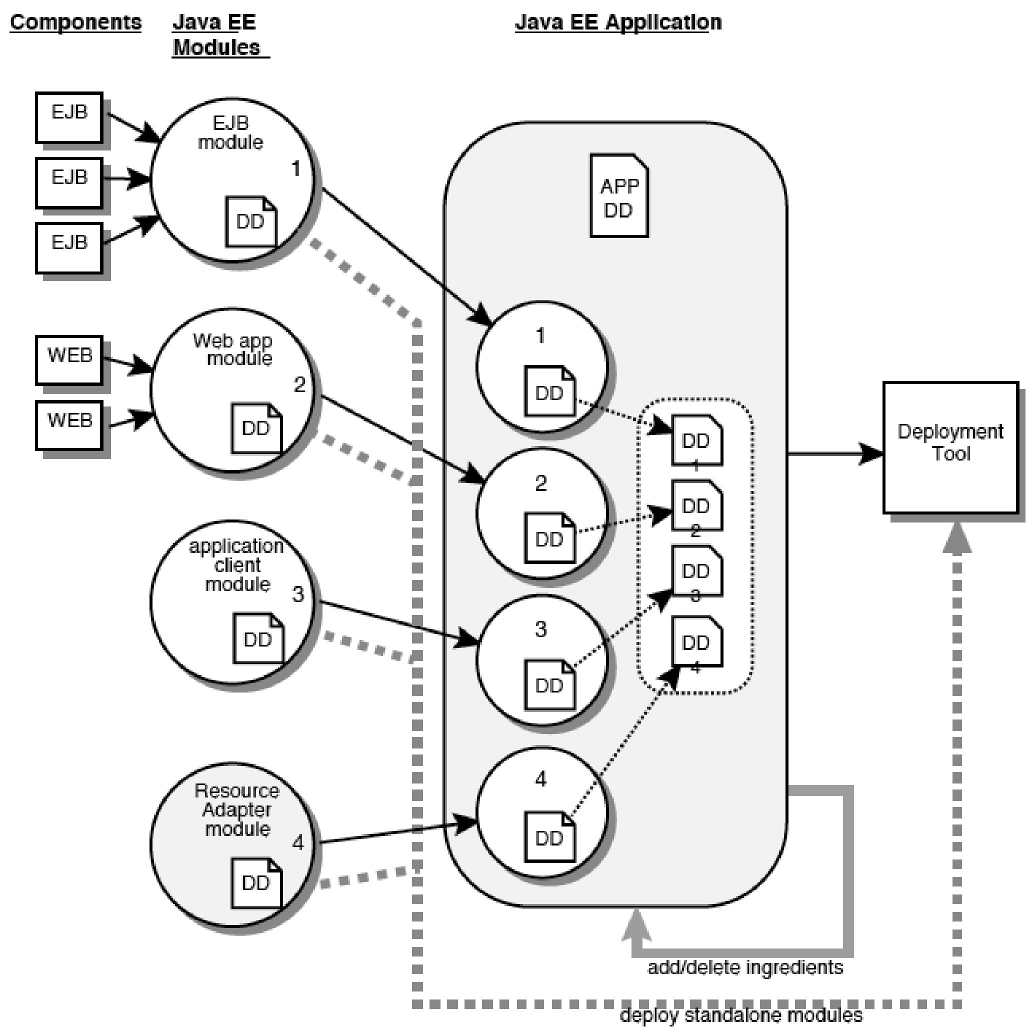


Figure 4: Java EE Deployment Structure

Figure 4 from [JAVAEESPEC] outlines the four different module types, where each module type contains one or many components of the same type. Different component types cannot be deployed in the same module:

1. *EJB Module* – containing EJB components as well as classes and resources they used. EJB modules are deployed (packed) as EJB-JAR files. EJB-JAR files are JAR files with the following structure is as follows:

```
META-INF/MANIFEST.MF - Manifest
META-INF/ejb-jar.xml - EJB Deployment Descriptor
```

```
*.class – all classes that made up or are used by the
components.
*.jar – referenced JAR files must be included in the
manifest.
```

2. *Web Module* – containing web components as well as classes, resources they used. Web modules are deployed (packed) using Web Archives (WAR) files. WAR files are JAR files with the following structure:

```
META-INF/MANIFEST.MF – Manifest
WEB-INF/web.xml – Web Deployment Descriptor
WEB-INF/classes/*.class – all classes that made up or
are used by the web components.
WEB-INF/lib/*.jar – all libraries needed by the web
components.
*.* – all accessible web resources and folders web
components except the above mentioned.
```

3. *Application Client Module* – containing all classes and resources. Application client modules are deployed using JAR files with the addition of an application client deployment descriptor like the other modules. The structure is as follows:

```
META-INF/MANIFEST.MF – Manifest
META-INF/application-client.xml – App. Client Deploy-
ment Descriptor
*.jar – referenced JAR files must be included in the
manifest.
*.* – all classes and resources that made up the ap-
plication client.
```

4. *Resource Adapter Module* – containing all classes that comprise the resource adapter.

```
META-INF/MANIFEST.MF – Manifest
META-INF/ra.xml – Resource Adapter Deployment De-
scriptor
*.jar – referenced JAR files must be included in the
manifest.
```

```
*.dll/*.so - native libraries.
*.* - all classes and resources that made up the re-
source adapter
```

Figure 4 from [JAVAEESPEC] also defines the structure of a Java EE application, which can be comprised of one or many modules regardless of the type of module. A Java EE application also has an optional deployment descriptor `META-INF/application.xml` , but if omitted, default naming rules for modules apply, because annotations are only used at the component level.

Figure 4 from [JAVAEESPEC] also shows, that modules can be deployed individually or that they can be assembled into a Java EE application.

Also, Java EE assumes the presence of a deployment tool to enable the deployment of the different modules or Java EE application into the final Java EE server. The structure or architecture of the deployment tool varies for different server implementations as each vendor provides its own.

All modules as well as the Java EE application are packed using JAR Files. However, the internal structure of these files varies depending on the type of module. The internal structure of a Java EE application is also different from the structure used by the modules.

During the deployment, the modules and components are configured and integrated into the existing infrastructure. Each module type must be installed into the corresponding container type and configured according to their annotations, or deployment descriptor if present.

Interoperability

The Java EE Specification mandates support of communication protocols for the different type of containers.

- Applet containers must be able to issue the following request types: JRMP, IIOP, HTTP and SSL. No mandatory requirements exist for responses.

- Application client containers must be able to issue the following request types: JRMP, IIOP, HTTP, SSL and SOAP over HTTP. No mandatory requirements exist for responses.
- Web containers must be able to issue the following request types: JRMP, IIOP, HTTP, SSL and SOAP over HTTP. Additionally, web containers must be able to serve HTTP, SSL and SOAP over HTTP requests.
- EJB containers must be able to issue the following request types: JRMP, IIOP, HTTP, SSL and SOAP over HTTP. Additionally, EJB containers must be able to serve IIOP, IIOP over SSL, SOAP over HTTP requests.

4.4 OSGi Overview

4.4.1 Introduction

The Open Services Gateway initiative (OSGi) Service Platform backed by the OSGi Alliance [OSGIAL] provides a Java-based software development platform specification, which aims at providing an open and common architecture for the coordinated development, deployment and management of software modules and (optionally) services provided by these modules.

The OSGi approach on application development is module and service oriented, where the overall application is comprised of the sum of all modules involved (cooperating) in the application. Hence, the term *Module System for Java.*

The current state of the OSGi Service Platform at the time this paper was written is Release 4. Having said that, this work uses the release 4 of the OSGi Platform and versions are not considered. Also, it is noted here that many portions of this section are based on [OSGISPEC] and [OSGISERV]

4.4.2 Overview

The OSGi Service Platform can be divided into two parts: 1) the OSGi Framework, and 2) Services, which in turn can be sub-classified into a) core services and b) optional services. The OSGi Framework forms the core of the OSGi Platform.

The specification [OSGISPEC] provides a definition and interfaces. Vendors provide an implementation of the framework and the services. Additionally, vendors may provide extensions to the framework. However, their use make for less portable software.

The OSGi Framework allows developers to create application modules known as *bundles*. Hence, a bundle is the modularization unit in OSGi. A bundle is an extensible and downloadable application module that can in turn be deployed in the OSGi Framework. Bundles can provide zero to many services that can be used by other bundles, which provides a service-oriented programming model for modules. The OSGi Framework, in turn, provides a general-purpose secure management environment for these bundles.

The OSGi Framework delineates a management environment for bundles and services, i.e. installation, removal and update of bundles as well as starting, registration, deregistration, stopping of services.

As stated before, the OSGi Framework is Java-based. In particular, it makes extensive use of Java's platform independence and dynamic class loading (see section on class loading). Java's class loading mechanism is central to the dynamic nature of Java as it enables the ability to install components at run time. More importantly, class loaders play a critical role in providing security in Java because the class loader is responsible for locating and fetching the class file, consulting the security policy and defining the class object with the appropriate permissions according to the security policies in place.

The service platform functionality is divided into layers, and each layer is responsible for a specific feature. Figure 5 after [OSGISPEC] depicts these layers.

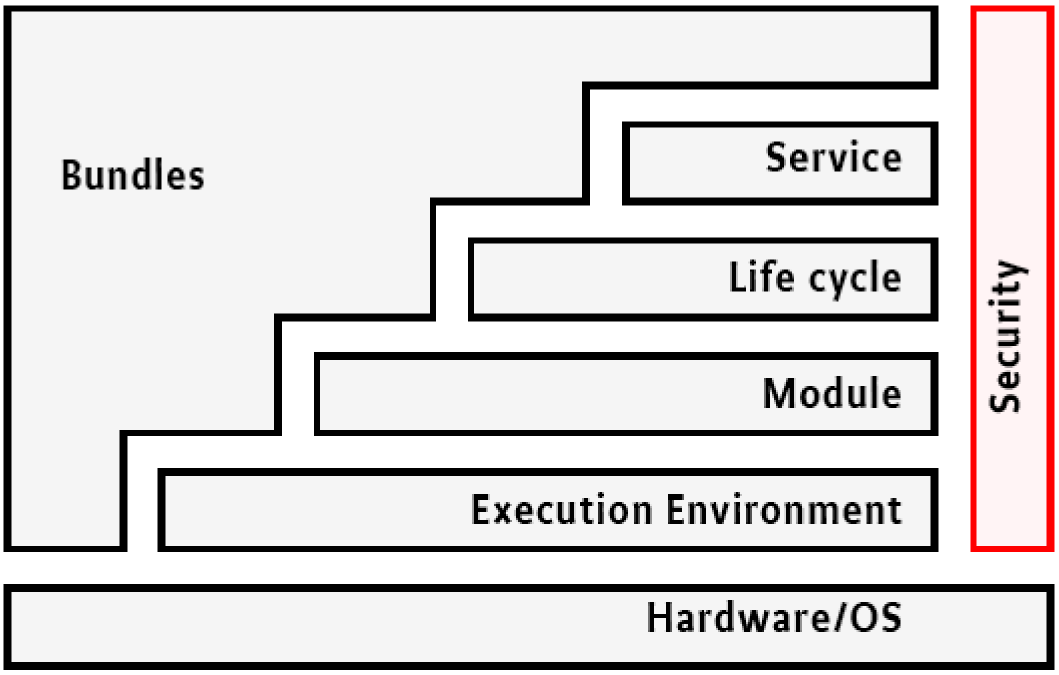


Figure 5: OSGi Layers

- *The Security Layer* – is based on Java 2 Policy System, see vertical section on security, with some extensions specifically to the management of OSGi modules. Additionally, this layer defines a secure packaging format and delineates the runtime interaction of this packaging format with the Java 2 security layer. One can see from Figure 5 that security is regarded as a cross cutting concern across all other layers. Also, it is to note that the security layer is optional.

- *The Module Layer* – defines the a generic and standardized solution for Java modularization. In particular, it enhanced the Java SE deployment model – JAR files – by adding additional rules for sharing/hiding Java packages between bundles.

- *The Life Cycle Layer* – provides a means (API) to manage the bundles in the Module Layer. More importantly, it provides a runtime model for bundles. In essence, how bundles are installed, updated, started, stopped and uninstall. It also delineates the state in which a bundle is and how to transition from one state into another state.

- *The Service Layer* – provides a concise and consistent programming model for bundle developers. It aims at simplifying the development and deployment of service bundles. In particular, it does so by decoupling the service's specification from the service's implementation, advertise the service using an interface and publish an implementation of the interface. Allowing bundles to select different service implementations at runtime through the framework's Service Registry, which allows bundles to: 1) register new services, 2) receive notifications about the state of services, 3) lookup existing services. Thus, allowing bundles to adjust to changes in the environment

- *Execution Environment* refers to the Java environment in which the bundles will be executed. For instance, J2SE-1.5 for applications running in a regular desktop or CDC-1.1 if OSGi is to run in a J2ME Foundation 1.1 complaint device.

Figure 6 after [OSGISPEC] depicts the interaction between the layers and the bundles. Top to bottom,

- bundles use services

- bundles are started, which initiates the bundle's life cycle. Upon startup of the life cycle, the bundle is installed as a module, and services are published by the bundle and managed by the framework.

- classes are loaded from the bundles based on their dependencies

- bundles are executed in a specific execution environment.

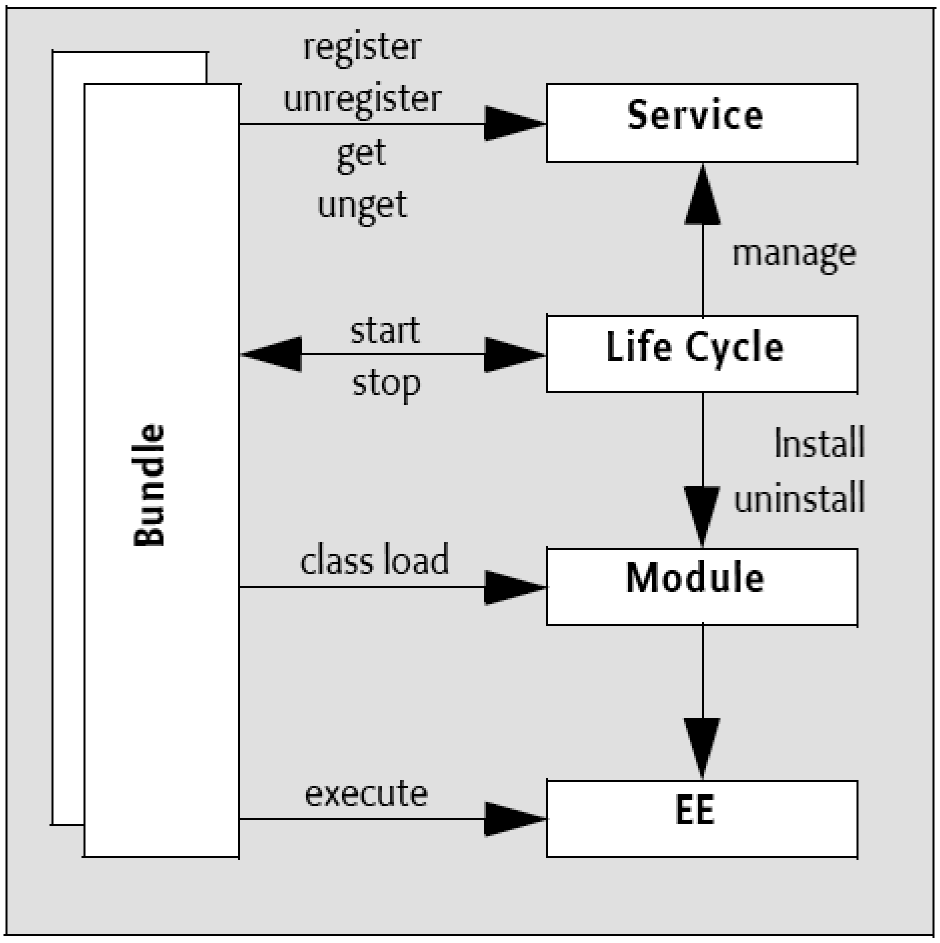


Figure 6: OSGi Layer Interactions

4.4.3 Module Layer

The Module Layer provides a standardized and generic modularization model for Java from a language perspective regardless of the type of Java application, e.g. Java ME, Java SE or Java EE. The modularization model in OSGi extends and enhances the regular packaging and deployment format of Java SE applications, namely the organization of classes and resources in packages and JAR files.

The modularization unit in the OSGi Framework is called a *bundle*, which consists of classes and other resources, which together provide functionality to users and other bundles. Bundles can share Java packages with other bundles, where the bundle containing the package explicitly exports the package is called an *exporter bundle*. Another bundle importing explicitly the package in an exporter bundle is called an *importer bundle*. Bundles are the only entity of modularization in OSGi and, therefore, the only application units.

A bundle is deployed as a JAR file, see section on JAR files. Also, upon being installed into the OSGi Framework, a bundle gets assigned a unique *bundle id* that can be used to refer to that bundle from within the OSGi Framework at runtime.

A bundle is a JAR file that:

- Contains all Java classes and other resources such as graphics, sound, help files, etc. to provide some functionality.

- May contain other (embedded) JAR Files that are available as resources and classes to the containing bundle. The contained JAR file, however, cannot contain other JAR files, thus this structure is non recursive.

- Contains a manifest file describing the contents of the JAR File according to the JAR File specification. The `META-INF/MANIFEST.MF` file contains additional header information regarding the bundle such as dependencies to other bundles, imported/exported packages and so on. The manifest headers must strictly follow the manifest format as of [JARSPEC]

- May contain an `OSGI-OPT/` folder that contains optional documentation regarding the bundle. OSGi Framework implementations may ignore or entirely remove this folder.

- Every bundle must have a unique symbolic name within the OSGi Framework. However, there may be multiple versions of the same bundle. Hence, bundle uniqueness is resolved by the bundle's symbolic name and version number.

A bundle must first be installed and started in that order. At that point:

1. the bundle's functionality is exposed and published services become available to other bundles installed in the OSGi Platform.

2. The bundle itself can be triggered to use services exposed by other bundles.

See section on life cycle layer.

Execution Environment

Bundles can be restricted to executed only in specific *execution environments*. *Execution Environment* is the term use in the OSGi Framework to outline a specific set of capabilities that a bundle can assume the Java Runtime Environment will provide, for instance J2SE, CDC, etc. Table 1 after [OSGISPEC] outlines the different execution environments specified in the OSGi Framework.

EE Name	***Description***
CDC-1.0/Foundation-1.0	Equal to J2ME Foundation Profile
OSGi/Minimum-1.1	OSGi EE minimal set that allows the implementation of the OSGi Framework
JRE-1.1	Java 1.1.x
J2SE-1.2	Java 2 SE 1.2.x
J2SE-1.3	Java 2 SE 1.3.x
J2SE-1.4	Java 2 SE 1.4.x
J2SE-1.5	Java 2 SE 1.5.x
JavaSE-1.6	Java SE 1.6.x
PersonalJava-1.1	Personal Java 1.1
PersonalJava-1.2	Personal Java 1.2
CDC-1.0/PersonalBasis-1.0	J2ME Personal Basis Profile
CDC-1.0/PersonalJava-1.0	J2ME Personal Java Profile

Table 1: Standardized Execution Environments in OSGi

When an Execution Environment is in the bundle header, it is the responsibility of the bundle developer to use only features provided in the stated environment.

For instance, `Bundle-RequiredExecutionEnvironment: J2SE-1.4, J2SE-1.5` - with the previous header the bundle is allowed to use only APIs existing in both Java environments. Thus, new Java 5 features cannot be included.

Conversely, a bundle requires that a specific Execution Environment is present before it is installed. The OSGi Framework is also obliged to provide the names of the distinct execution environment it provides. This feature is to be considered volatile as another bundle may extend the Execution Environment at runtime and thus change or add new features in the execution environments.

Bundle Cooperation

Many bundles can be installed and started into a single *Java Virtual Machine* (JVM).

The mechanism for bundle co-existence and co-operation used by the OSGi Framework is the *class loading* feature provided by Java Programming Language. In fact, the OSGi Framework exploits extensively the class loading feature of Java and at the same time it enhances it by providing strict and well-defined rules for bundle cooperation.

Bundles can hide/share packages and classes with other bundles. For example, bundle B1 exports a package P1 and bundle B2 imports package P1 from B1. B1 is called the *exporter* and B2 the *importer*. Additionally, the exporter can explicitly exclude classes within the exported package, this feature is known in OSGi as *class filtering.*

A bundle can also be directly wired to another bundle. This is, however, not recommended as it tightly couples bundles to each other. It also allows the existence of *split packages* – package contents come from different sources (bundles). That in itself may lead to [OSGISPEC]:

- Completeness issues: no guarantee where the package ends. No means to figure out if all classes may have been included.

- Ordering: bundles must be required in the right order.
- Performance: when searching from classes in multiple bundles, an increase number of `ClassNotFoundException` are thrown until the class is found.
- Confusion: as classes might come from different bundles and thus loaded by different class loaders.
- Mutable Exports: the export signature of the requiring bundle can suddenly change.
- Shadowing: classes in the requiring bundle are shadowed by those in a required bundle.

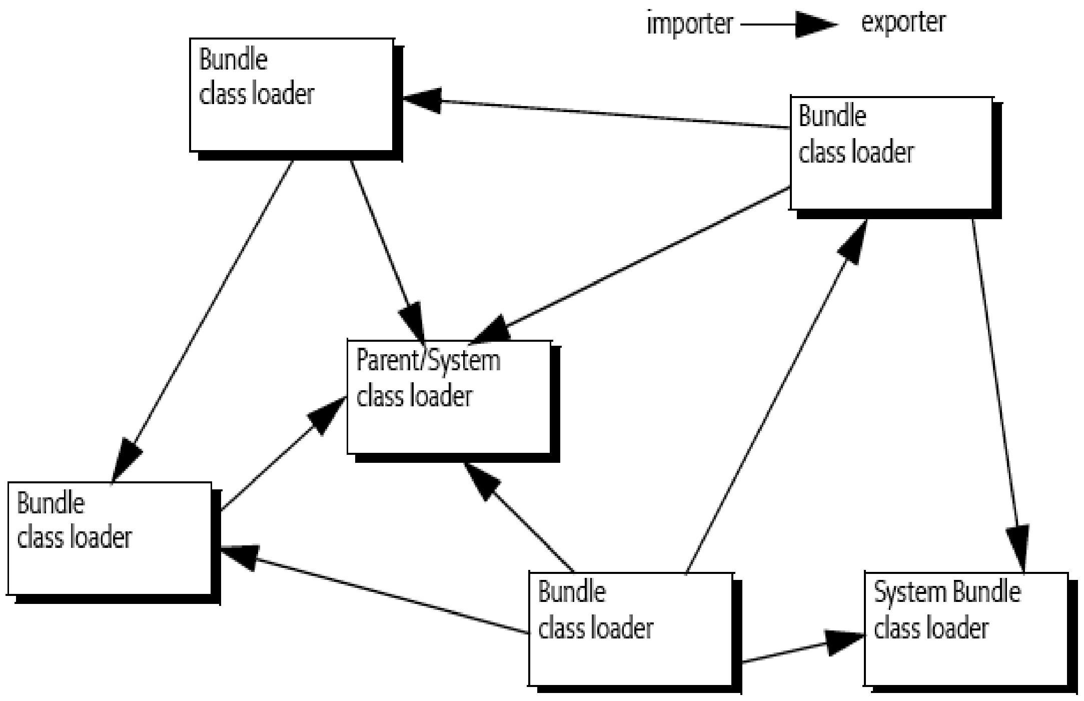


Figure 7: Class Loading Structure in OSGi

Each bundle that is not a fragment[11] is loaded in its own class loader. The bundle class loader provides the bundle with its own namespace. This avoid name conflicts with other bundles as well as ensure resource sharing with other bundles.

A single bundle class loader as well as the class loaders from other bundles create a delegation network of class loaders. This class loader delegation model exploits the ability of a class loader to delegate the loading of a class to another class loader. Figure 7 depicts the class loading delegation model in OSGi.

The OSGi Framework defines 3 distinct types of class loaders:

- *System Class Loader:* loads classes and resources provided by the execution environment; i.e. classes in the `java.*` packages.
- *Framework Class Loader:* loads classes and resources provided by the OSGi Framework, as well as classes and resources provided by core services, classes in the `org.osgi.*` packages and their implementation. This class loader is used by the system bundle, which provides the implementation of the OSGi Framework. See section on system bundle.
- *Bundle Class Loader:* loads classes and resources included in the bundle (JAR file) as well as classes and resources included in enclosed JAR files or extensions provided by fragment bundles. Every bundle has it's own class loader and share the parent class loader.

The OSGi Framework also defines the bundle's *class space*, which includes classes and resources loaded from:

- *boot class path* – classes and resources loaded by the System Class Loader.

11 Fragments are bundles that extends other bundles, this bundles are called a *host bundles*. Therefore, fragments are treated as part of the host bundle and become part of the host bundle class space. This feature is typically used to support translation files for different locales. Which allows the different localization to be shipped independently. Fragment bundles can also be used to provide optional extensions to a specific bundle.

- *framework's class path* – classes and resources loaded by the Framework Class Loader.
- *bundle class path* – classes and resources loaded by the Bundle Class Loader.
- *required bundles path* – classes and resources loaded by other bundles.
- *imported packages* – classes and resources imported from other bundles.

As with regular class loaders, the class space must be consistent, in other words, every class must be resolved exactly once. However, the OSGi Framework supports versioning, which allows the loading multiple versions of the same class.

Having said that, the OSGi Framework is responsible for instantiating and maintaining all class loaders except for the System Class Loader, which is the primordial class loader provided by the execution environment.

The OSGi Framework is not only responsible for providing a class loader to each bundle but it is also responsible for *resolving bundles. Resolving* is the process of *wiring two bundles,* where one bundle is the importer and another is the exporter. In other words, the OSGi Framework is responsible for creating the bundle's class space, creating a class loader after all wires are resolved and finally, loading and executing each class in the corresponding bundle's class loader.

Although the OSGi Framework is responsible for resolving bundles, bundle developers have the means to declaratively state all bundle dependencies (imported/exported packages) and their constraints (version) in the `META-INF/MANIFEST.MF` file.

According to [LIANG], class loaders must exist in strictly hierarchical structure with the primordial class loaders at root of the inheritance tree. However, the maintenance of this hierarchical structure has to be ensured by the class developer, which may lead to errors at runtime if the class loader does not follow the implementation guidelines. Having said that, the OSGi Framework mandates the loading of classes in the `java.*`[12] packages by the System Class Loader, but it also allows other classes to be loaded by the System Class Loader, thus the OSGi framework allows the expansion of the bootstrap class space. This is done via the system property `org.osgi.framework.bootdelegation`. For instance when running on a SUN JVM the following must be necessary. `org.osgi.framework.bootdelegation=sun.*, com.sun.*`. Which states all class from the `sun` as well as the `com.sun` packages should be loaded by the System Class loader.

Bundles are not allowed to import any of the `java.*` packages. However, the OSGi Framework must explicitly export relevant non `java.*` packages such as Java extension packages, `javax.*`. This is done via the system property `org.osgi.framework.system.packages`. For instance, `org.osgi.framework.system.packages=javax.cryto.*`. Careful attention must be paid here, as these exports can collide with some to the other exported packages by bundles. This is a deviation of Java as everything the JRE has to offer is available to an application running in it automatically.

Bundle exports are accessible immediately after the bundle has been resolved. Hence, an importer may use an exported package before the exporter bundle is started.

Bundle Constraints

Constraints are conditions on a wire a bundle provider can state in the manifest to match imports to exports. Constraints also state explicit conditions on an export that importers must follow in order to use the bundle.

12 The wild card means recursive matching; all classes in the `java` package as well as other subpackages within it.

A bundle is able to constrain an import from another bundle by a) absolute version, b) version list, or c) version ranges. An OSGi bundle versioning is used as: *Major* – an incompatible update, *Minor* – a backward compatible update, *Micro* – a change that does not affect the interfaces. For instance, `Import-Package: p; version="[1.5.0,1.5.3)"` will import packages p of version 1.5.0 through 1.5.3, excluding version 1.5.3.

Imports can be declared *optional or mandatory,* optional imports means the imported package is not required for the bundle to resolve correctly. This can be specified using the dynamic import header: look for exported package when needed (class loading); or the resolution-directive: import definition with the resolution set to optional.

Inter-package dependencies can be depicted by means of the *uses* directive in the Export-Package header. This means: if an importer A wants to use the package p1 in bundle B that uses package p2 in bundle C, importer A must also import p2 in bundle C for the bundle to be resolved.

The complete set of constraints constructed from recursively traversing the wirings is called *implied package constraints.* These implied constraints are not automatic imports, but must be taken into consideration when importing in order to preserve class space consistency.

The OSGi Framework also allows the importer/exporter to influence the wiring process in a declarative way, called *declarative constraints.* This allows semantics like, import accounting module from company “MyCompany”.

Although the wiring of an imported package is typically done implicitly, an importer has a means to state specifically from which bundle it wants a package to be imported. This includes a specific bundle name and bundle version or version ranges. This feature is also called (bundle) *Provider Selection.*

The OSGi Framework also provides a facility for bundles to provide classes and resources to other bundles, via the `Fragment-Host` header. This feature allows a bundle to provide optional features to existing bundles. This feature is commonly used for localization.

Resource Loading

Resources from a bundle may come from the bundle's JAR file, fragments, imported packages or the bundle classpath. The OSGi Framework states all resources must be loaded using the class loader for that bundle. This should be done with the class loader's methods `getResource(String)` and `getResources(String)`, which return an URL or an enumeration of URLs respectively. These URLs are bundle relative and are called *bundle entry URLs.* API classes provided by the OSGi Framework must load resources using the bundle's class loader.

Loading Native Code

Another key Java feature that the OSGi Framework is based on is the Java Native Interface (JNI). JNI allows the integration of native code[13] into the bundles. All native libraries must be loaded using the class loader of the bundle attempting to load the library. Native code libraries can be specified in the MANIFEST.MF file in a header to enhance the wiring process. Bundle providers can include the following native library descriptions:

- *Operating System (OS)* name to indicate the name of the OS for which the native library runs, for instance Windows2000, WindowsXP, Linux, etc. For a comprehensive list of OS names, refer to [OSGISPEC] [OSGIAL]

- *OS version* specifies the version range under which the native library runs.

- *Processor* specifies the hardware architecture under which the native library runs (Mips, Alpha, x86, etc.). For a comprehensive list of proces-

13 Native code is code compile exclusively for a specific hardware architecture and operating system.

sors refer to [OSGISPEC] [OSGIAL]

- *Language* to indicate the language for which the bundle has been localized.

The OSGi allows only one entry in the native library header per platform. If multiple libraries need to be loaded for one platform all libraries need to appear in the same header as a list.

A native library can only be loaded by single class loaders in order to preserve namespace separation. Failure to do so will result in a linkage error.

Bundle Localization

The manifest file includes a substantial amount of human readable OSGi headers. The OSGi Framework allows a bundle provider to extract all that information and to place it into Java Resource Bundle[14] for each supported Locale[15]. The localized information is placed in properties resources `OSGI-INF/l10n/bundle.properties` resource bundle for the default locale. Each supported locale follows with its own file with 2-letter language code in lower case and 2-letter country code as specified by [reference to Java API for Locale], for instance for German and Austria `OSGI-INF/l10n/bundle_de_AT.properties`. The content of these properties files are key value pairs, where the key is the same in all properties files, but the values are in the corresponding language they support.

Extension Bundle

Optional parts of the OSGi Framework as well as extensions to the Framework and the execution environment can be done using *extension bundles*. These

14 Resource bundles contain locale specific objects. It allows a developer to isolate a Java application from most if not all the locale specific information. Although Java supports many different kinds of resource bundles, the OSGi Framework uses specifically the `PropetyResourceBundle` class.

15 A Locale object represents a specific geographical, political, or cultural region. More specifically Java uses ISO-639-2 for language codes and ISO-3166 for country codes. Locales enable tailoring of information according to an users location.

bundles are different from regular bundles as they can provide packages that must reside in the boot class path or framework's class path. Therefore, their packages cannot be imported/exported. Extension bundles will not be loaded in their own class loader either, as extension bundles will be appended to the framework's class path and boot class path extensions are installed in the boot class path.

Furthermore, extension bundles must have `AllPermission` granted to them as they are part of the Execution Environment or the OSGi Framework. Both of these Protection Domains have `AllPermission` granted to them. See section on Security Layer.

4.4.4 Life Cycle Layer

The Module layer describes the static characteristics of bundles. The Life-Cycle Layer specifies the dynamic characteristics and behavior of this bundles. In other words, the runtime state of a bundle and the framework itself as well as events triggered by these state transitions.

Furthermore, the OSGi Framework provides a well-defined API to control the life-cycle of bundles as well as the state of the framework and the installed bundles.

Although there are many classes involved in the life cycle layer, some of the most relevant classes in the API are [OSGISPEC]:

- `Bundle` – Represents an installed bundle in the Framework.
- `Bundle Context` – A bundle's execution context within the Framework. The Framework passes this to a Bundle Activator when a bundle is started or stopped.
- `Bundle Activator` – An interface implemented by a class in a bundle that is used to start and stop that bundle.

- `Bundle Event` – An event that signals a life cycle operation on a bundle. This event is received via a (Synchronous) Bundle Listener.
- `Framework Event` – An event that signals an error or Framework state change. The event is received via a Framework Listener.
- `Bundle Listener` – A listener to Bundle Events. Should be implemented by developers willing to listen to bundle events.
- `Synchronous Bundle Listener` – A listener to synchronously delivered bundle events. Should be implemented by developers willing to listen to bundle events.
- `Framework Listener` – A listener to Framework events. Should be implemented by developers willing to listen to framework events.
- `Bundle Exception` – An Exception thrown when Framework operations fail.
- `System Bundle` – A bundle that represents the Framework.

Bundle

Every `Bundle` installed in the framework is represented by a `Bundle` object, which can be used to manage the bundle in the OSGi Framework.

Every bundle has an *identifier* (`long`) assigned by the OSGi Framework and is valid for the full life time of the bundle. Bundle identifiers are assigned in ascending order corresponding to the installation order.

Every bundle object must have also a *symbolic name* retrieved from the header in the manifest file; keep in mind that the combination of symbolic name and bundle version is what makes a bundle globally unique.

Each bundle object has a *bundle location* representing the URL to the JAR file used to install the bundle from. This location must be unique and cannot be changed.

A bundle can be in one of the following states during its life cycle:

- *INSTALLED* – The bundle has been successfully installed. A bundle's structure must be valid and its location must be unique before it can be installed on the framework . At this point the bundle gets a unique bundle identifier (higher than an existing one) and the bundle object is created. This is persistent and it must be managed by the framework until the bundle is uninstalled. If the bundle for an existing location already exists, the framework must just update the existing object. Finally, all remaining life cycle operations must be performed on the instantiated bundle object.

- *RESOLVED* – A bundle enters the resolved state after all bundle dependencies, fragments, imports, exports, etc., are resolved. Thus, in this state all Java classes that the bundle needs are available. This state indicates that the bundle is either ready to be started or that it has stopped.

- *STARTING* – The bundle is being started. Immediately after and based on the activation policy declared in the manifest the bundle will be activated or not. *Activation* is the process of starting up[16] the bundle; conversely, d*eactivation* is the process of shutting down[17] the bundle. Activation is optional, for instance, a library bundle. There are two types of activation policies: a) *eagerly* – where the bundle is immediately activated (default), and b) *lazily* – where the bundle remains in the STARTING state until the first class in a specified package(s) from bundle is loaded. The activation depends on the activation policy (eager/lazy), but the activation process is done using a `BundleActivator` object a bun-

16 During the startup process bundles can create threads, start servers, register services, etc.

17 All previously started threads must be stopped at this point. At this point no framework object must be reached by the bundle. There is no need to deregister services or listeners as they will be cleanup automatically by the framework.

dle developer provides in the manifest. The Activation is always triggered by the framework, who creates a `BundleActivator` and invokes its `BundleActivator.start(BundleContext)` method.

- *ACTIVE* – The bundle has been successfully activated and is running; its Bundle Activator start method has been called and returned.
- *STOPPING* – The bundle is being stopped. The `BundleActivator.stop (BundleContext)` method has been called but the stop method has not yet returned. In general, the stop method should undo the work done by the start method. Upon successful execution of the stop method, the bundle transitions into the resolved state.
- UNINSTALLED – The bundle has been uninstalled. It cannot move into another state.

Figure 8 after [OSGISPEC] depicts the states of a bundle as well as the state transitions from one state to another.

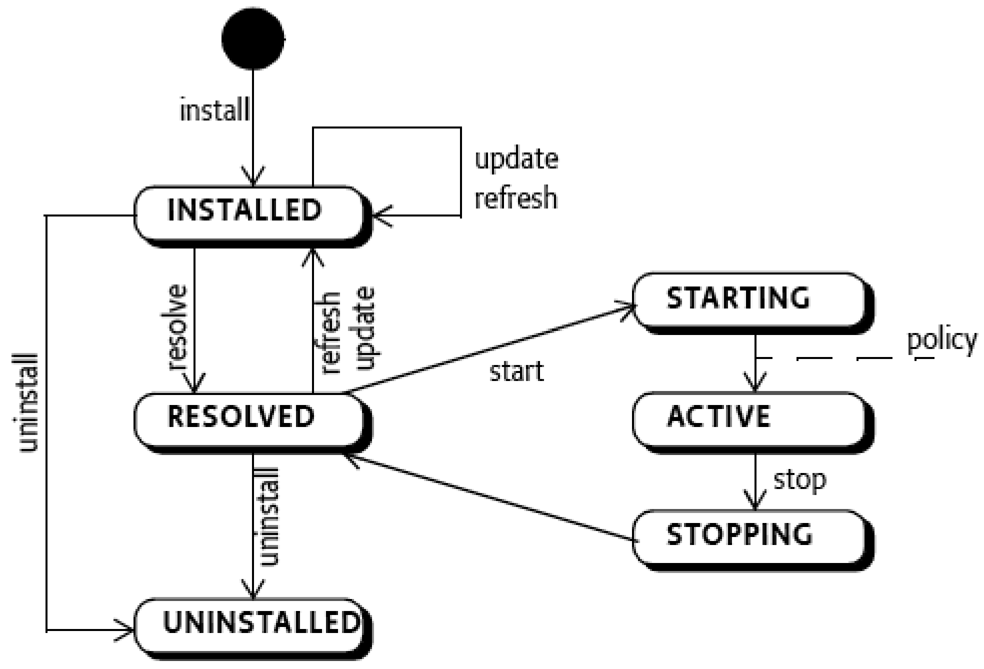


Figure 8: Bundle State Diagram and Transitions

System Bundle

The OSGi Framework itself is implemented as a bundle. This bundle is referred to as the *system bundle*. Built-in services offered by the Framework are registered by the system bundle.

Although the system bundle shows as a regular bundle, it is different to other bundles in the following ways.

- It always has the identifier 0 (zero)
- The location shows typically “System Bundle”
- Although it has it's own symbolic name, it should also respond to the `system.bundle` alias.
- It's life cycle is different:
    - start does nothing, because the bundle is already started
    - stop returns immediately and shut downs the framework

    - update returns immediately and shut downs and restarts the framework in a new thread.

- It cannot be unistalled.

Bundle Context

The Bundle Context object realizes the relationship between the a bundle and the OSGi Framework. Hence, this object represents the execution context of a bundle and provides access to the framework's capabilities to the bundle. This object is provided by the framework to the Bundle Activator.

Some of the capabilities provided by the bundle context are:

- Installing new bundles
- Retrieving and interrogating other bundles installed in the OSGi Framework. This capability is not restricted. Any bundle should get access to any other bundle in the framework.
- Providing environment information to the bundle, such as framework version, framework vendor, framework language, execution environment, processor, OS version, OS name and other.
- Obtaining persistent storage area. The OSGi Framework defines optional support for a private storage area in form of file system (`java.io.File`). Bundle Context allows the use of this private storage area
- Registering/retrieving services
- Subscribing/unsubscribing to event broadcast

Events

The OSGi Framework supports two types of events:

- *Bundle Event* (`BundleEvent`)to report changes in the life cycle [reference to figure on bundle state] of bundles.

- *Framework Events* (`FrameworkEvent`)to report information, warning and error messages as well as to report when the framework has been started, updated and the start level has changed.

A bundle developer interested in listening to bundle events must implement a `BundleListener` or a `SynchornousBundleListener` to receive notification of changes in the bundle's life cycle. `SynchronousBundleListener`s are called during the processing of the event before `BundleListener`s. Additionally, `SynchronousBundleListener`s can listen to lazy activation, starting and stopping events, which are state transition events. Other wise all events are triggered after the bundle has changed its state.

A bundle developer interested in listening to framework events must implement a `FrameworkListener`.

All event listeners are registered using the `BundleContext` object. The framework ensures all events are delivered only to listeners registered at the time the event was published and to listeners in active bundles - bundles in the active state.

4.4.5 Service Layer

The Service Layer of the OSGi Service Platform delineates a collaboration model for bundles that is tightly integrated with the bundle's life cycle. This collaboration model is built around *services*. A *service* is to be regarded as piece of functionality provided by a bundle, for instance, data service, connection pool service, naming service, etc.

The service model use in the OSGi Framework is of publish, find, bind. The framework provides a facility for bundles to publish, find and bind to each other's services. This service facility is called the *Service Registry.* Additionally, the OSGi framework provides a means to restrict access and operations on the services.

The service model in the OSGi Framework is persistent, because the framework allows bundles to track services even across framework restarts.

Also, this service model accounts for the evolution of services over time, which allows a bundle to be bound to a new version of the service it uses.

Furthermore, the OSGi Framework comes with a comprehensive API to control operations on services – publishing, finding and binding – as well as an API to track service events.

A service is a Java object registered under one or many interfaces with the service registry, which is provided and fully managed by the framework.

Although the are many classes involved in service layer. Some of the most relevant classes in the API are [OSGISPEC]:

- `Service` – An object registered with the service registry under one or more interfaces together with properties. This object can be discovered and used by bundles.

- `Service Registry` – Holds the service registrations. It provides th facility that the OSGi Framework provides for bundles to publish, find services.

- `Service Reference` – A reference to a service. Provides access to the service's properties but not the actual service object. It allows a bundle to inquire about a service without directly using the service and thus creating a dependency. It can be used by the bundle to select the most suitable service based on the service's properties. It is used in combination with the bundle context to acquire the the service object.

- `Service Registration` – The receipt provided by the service registry when a service is registered successfully. The service registration can be used to update the service properties. It can also be used to deregister the service. The service can only be unregistered by the holder of the service registration.

- `Service Permission` – A permission to delineate the use of a service or to preform the registration of a service.

- `Service Factory` – A facility to let the registering bundle customize the service object for each using bundle.

- `Service Event` – An event fired when a service is registered, unregistered or when the service properties are updated.

- `Service Listener` – A listener to Service Events. Should be implemented by developers willing to listen to service events.

- `Filter` – An object that implements a simple but powerful filter language[18] that can be used to filter the selection of a service based on it's service properties.

Services

Bundles provide functionality to other bundles by means of a service, which represents the unit of bundle cooperation in the OSGi Framework.

The *OSGi Service* consist of: a) a *service interface*[19] that defines the service, and b) a *service implementation* that provides the service as defined in the service interface.

An OSGi service is provided by, registered by and owned by a bundle. The service also runs within the bundle. The bundle registers the service with the

18 The filter expressions are based on Lightweight Directory Access Protocol (LDAP) . Please refer to RFC 2254.
19 Note that regular Java classes (abstract, final or concrete) can be advertised as services too

OSGi Framework's service registry. Upon successful registration the registered service becomes available to other bundles. It's important to note that, if a bundle B1 publishes a service S1 and a bundle B2 uses the service S1 in bundle B1, the S1 still runs within the bundle B1 (using the bundle B1 class loader).

A service may contain service properties (key/value pairs - `String`) which describe the service. The service properties are intended to provide information about the service and should not be used to alter the functionality of it. These service properties can be used to filter services available in the framework.

The framework predefines some of these service properties.

- `objectClass`: provides a set of interface names under which the service implementation was registered. This property is set automatically.
- `service.description`: can be use for documenting the service. This property is optional.
- `service.id`: the framework assigns a unique service id to each registered service.
- `service.pid`: is a unique persistent id that can be used to identify a service across framework restarts.
- `service.ranking`: is numeric value that can be used to rank services that are registered under the same service interface. The framework returns the service with the highest rank order if present. If not present, the framework returns the service with the lowest service id;

Service Management

An important aspect of service management in the OSGi Framework is that all dependencies are managed by the framework.

- When a bundle is stopped, all registered services for that bundle are automatically deregistered by the framework.
- The framework provides an API for a bundle to manage it's services and the services it uses.
- The framework provides an infrastructure to notify bundles about service registration, modification and deregistration events.
- Every bundle has means to retrieve all services it has published and all services it s using.
- The framework must ensure that during the registration all service implementations adhere to the interface(s) it has been publish for.
- Services can be registered and unregistered by a bundle dynamically while the bundle is in the STARTING, ACTIVE or STOPPING states.
- The framework must ensure that a bundle requesting a service will be able to safely cast the service object to any of the associated interfaces under which the service has been registered. Having said that, the framework's service registry must ensure that bundles see only compatible services to avoid a `ClassCastException`.

Service Events

As mentioned earlier the framework provides an API for handle event in the service life cycle: registration, unregistration and changes in the service properties.

A `ServiceListener` allows a bundle to listen for service events for a specific compatible services only. A bundle developer using a service should implement a `ServiceListener` and register it with the framework. This `ServiceListener` is used to track the availability of a service; thus, it allows the bundle developer to take appropriate actions when the service state changes.

In particular, bundle developers must ensure that all references to another bundle's class are deleted to minimize the effect of stale references[20]. More importantly, the OSGi Framework does not specifies the behavior of a service when it becomes unregistered. Such services may continue working or they may throw an exception at the discretion of the service implementation.

Some bundles would like to listen to all events even from incompatible services. This can be accomplished by means of the `AllServiceListener` interface, which allows a bundle to listen to all service events regardless of their compatibility.

Service Factory

Service factories as their name say are classes that can be use to produce service object, based on the Service Factory design pattern [GAMMA].

In the OSGi Framework a service factory can be used to customize the service objects returned by a bundle to the requesting bundle. Therefore, it allows to create a unique service object for each bundle that gets the service. At the same time, a service factory also helps manage dependencies not managed by the framework. For instance, a service can be notified when a bundle no longer uses the service.

Service factories can also be used to minimize stale references if coded using *indirection*. Indirection simply said that the service objects return by the service factory do not return the service object itself, but a reference to it. Hence, when the service object becomes invalid, the reference can be nullified, thus removing the service object.

Service Definition

Organizations and bundle developers define and create services. However, the OSGi Framework defines a list of built in services.

20 A stale reference is an object reference from a class in a stopped bundle.

In fact, the OSGi Framework classifies these services in two major categories:

1. *Core Framework Services* defined [OSGISPEC] are considered part of the framework, but they are optional, and

    1. *Permission Admin* – The permissions of current or future bundles can be manipulated through this service. Permissions are activated immediately once they are set.

    2. *Package Admin* – Bundles share packages with classes and resources. The update of bundles might require the system to re-calculate the dependencies. This Package Admin service provides information about the actual package sharing state of the system and can also refresh shared packages. i.e. break the dependencies and recalculate the dependencies.

    3. *Start Level* – Start Levels are a set of bundles that should run together or should be initialized before others are started. The Start Level Service sets the current start level, assigns a bundle to a start level and interrogates the current settings.

    4. *URL Handler* – The Java environment supports a provider model for URL handlers. However, this is a singleton making it impossible to use this in a collaborative environment like OSGi that potentially has many different providers. This service specification enables any component to provide additional URL handlers.

2. *Add-on Services* defined in [OSGISERV]:

    1. *Log Service* – The logging of information, warnings, debug information or errors is handled through the Log Service. It receives log entries and then dispatches these entries to other bundles that have subscribed to this information.

2. *Configuration Admin Service* – This service provides a flexible and dynamic model to set and get configuration information.

3. *Device Access Service* – Device Access is the OSGi mechanism to match a driver to a new device and automatically download a bundle implementing this driver. This is used for Plug and Play scenarios.

4. *User Admin Service* – This service uses a database with user information (private and public) for authentication and authorization purposes.

5. *IO Connector Service* – The IO Connector Service implements the CDC/ CLDC javax. microedition.io package as a service. This service allows bundles to provide new and alternative protocol schemes.

6. *Preferences Service* – This service provides access to hierarchical database of properties, similar to the Windows Registry or the Java Preferences class.

7. *Component Runtime* – The dynamic nature of services -- they can come and go at any time -- makes writing software harder. The Component Runtime specification can simplify handling these dynamic aspects by providing an XML based declaration of the dependencies.

8. *Deployment Admin* – The primary deployment format for OSGi is the bundle, which is a JAR/ZIP file. The Deployment Admin provides a secondary format: the deployment package. Deployment Packages can combine bundles with arbitrary resources into a single deliverable that can be installed and uninstalled. A comprehensive model of resource processors allows user code to extend the resource types.

9. *Event Admin* – Many OSGi events have specific typed interfaces, making it hard to receive and filter events generically. The Event Admin provides such a generic, topic-based event mechanism. The

specification includes mapping for all existing framework and service events.

10. *Application Admin* – The OSGi bundle model is different from the typical desktop or mobile phone application model that relies on starting and stopping applications. The Application Admin prescribes such a traditional application model and its required management infrastructure.

11. *Http Service* – The Http Service is, among other things, a servlet runner. Bundles can provide servlets, which becomes available over the Http protocol. The dynamic update facility of the OSGi Service Platform makes the Http Service a very attractive web server that can be updated with new servlets, remotely if necessary, without requiring a restart.

12. *UPnP Service* – Universal Plug and Play (UPnP) is an emerging standard for consumer electronics. The OSGi UPnP Service maps devices on a UPnP network to the Service Registry. Alternatively, it can map OSGi services to the UPnP network. This is a recommended Release 3 specification.

13. *DMT Admin* – The Open Mobile Alliance (OMA) provides a comprehensive specification for mobile device management on the concept of a Device Management Tree (DMT). The DMT Admin service defines how this tree can be accessed and/or extended in an OSGi Service Platform.

14. *Wire Admin Service* – Normally bundles establish the rules to find services that they want to work with. However, in many cases this should be a deployment decision. The Wire Admin service therefore connects different services together as defined in a configuration file. The Wire Admin service uses the concept of a Consumer and Producer service that interchange objects over a wire.

15. *XML Parser Service* – The XML Parser service allows a bundle to locate a parser with desired properties and compatibility with JAXP.

16. *Initial Provisioning* – defines how a Management Agent can make its way to the OSGi Service Platform. A management Agent can be use to remotely administer the Service Platform, specifically, enabling the management of a Service Platform by an Operator.

17. *Foreign Application Access* – The OSGi service architecture is not natively supported by foreign application models like MIDP, Applets, other Java application models. The purpose of this service is to enable these foreign applications to participate in the OSGi service oriented architecture.

5 Java EE and OSGi Integration

As mentioned earlier, the purpose of this chapter is to shed some light over the integration of the Java EE and the OSGi Framework. A sample application was created to experiment with a service-oriented development model for Java EE applications using OSGi.

The service-oriented architecture is outlines here and the results are presented based on the experience gather during the creation of the sample application.

As mentioned earlier the aim of this work is to analyze the integration of OSGi and Java EE application from the application developers point of view.

5.1 The Sample Application

The implementation of an address book was selected for the feasibility study of the service-oriented Java EE application.

The ideas behind selecting the address book as the sample application was:

1. to select a simple application in terms of business logic, because the emphasis of this work is not a specific business logic. The emphasis of this work is rather to study the composition of a modularized Java EE application, and

2. to select only the most commonly use features of a Java EE application to keep the example still relevant. In particular, the author puts an emphasis on the most commonly used features in a Java EE application. This features are:

    1. database access service[21] using JDBC,

    2. persistence management service using JPA,

21 Data source in terms of Java EE.

3. transaction management service JTA
4. web components using Servlets and JSP.

   Please note that not all Java EE services were studied, because additional features require more careful study. Because of this, the author suggests topics that require further study and are beyond the scope of this work.

Finally, this sample application studies the integration of non Java EE features in the new service oriented Java EE environment.

Application Server Decision

For the sample application a lightweight architecture was selected. When it comes down to Java EE development, developers can

1. either "prune" the application server features they need,
2. or include individual Java EE features needed.

Although, the second approach represents a small deviation in terms of Java EE, which assumes the presence of an application server, the author believe that in the near future many application developers will not rely in an application server as much, but much rather rely in the features a Java EE server provides.

For this work, the author selected to the second approach to keep the application weight to a minimum but also to fully realize a service-oriented Java EE application.

Architecture

Modules and services created for the sample were:

1. *Start Database bundle* – starts and stop a Hypersonic SQL[22] database in a separate process. This bundle checks if the Database is started, if so, it does nothing; if not, it starts the Database at a specific location. This bundle uses OSGi file system based persistence capability to start the database at a specific location. It also introduces additional meta information in bundles. Please note, that in large scale enterprise applications this approach will not work, because of the complexity of large database management systems. However, the realization of a similar service to check the presence of the database is possible.

2. *Data Source bundle* – this bundles creates JDBC connection pool and publishes it as OSGi service. This bundle depends on the

3. *Transaction Manager bundle* – this bundles depends explicitly on the *Data Source* service published by the *Data Source* bundle and attaches a transaction manager to the *Data Source* to manage user transactions when accessing the database.

4. *Address Book Data Service bundle* – provides data service to the address book database. For instance, adding a new contact, removing a new contact, adding a new phone, etc.

   First, this bundle checks if database tables exists, if not, it generates the address book data model.

   A persistence manager (JPA) was used to ease the development process of this data service. The persistence manager uses the *Transaction Manager* service published by the *Transaction Manager* bundle and the *Data Source* service published by the *Data Source* bundle. The whole address data service was programmed using the Data Access Object [GAMMA]design pattern in order to abstract access to the database.

   Finally, this service publishes the *Address Book Service* as an OSGi service, which is realized as a Service Facade [GAMMA]. The purpose of

22 Hypersonic SQL is a small foot print, file system based database. [HSQLDB] .

this service facade is to completely encapsulate the realization of the address book data service. This facilitates the change or update of the service without breaking the overall application.

5. *Vcard Bluetooth bundle* – detects Bluetooth[23] capabilities in the operating system where the enterprise application resides. If so, the operating system starts a server that waits for incomming vCards[24] over Bluetooth. Upon reception of a vCard the server stores it in operating system's temporary directory and publishes the vCard as a file handle (`java.io.File`).

6. *Vcard Receiver bundle* – uses the “Whiteboard” pattern [KRIENS] to receive vCards published by the *Vcard Bluetooth* bundle. Upon detection of the vCard, the server publishes them as `Vcard` objects to the web application as a service.

7. *Address Book Web bundle* – provides a web front end to the address book. It depends on the *Address Book Data Service* published by the *Address Book Data Service* bundle. Additionally, it also uses the “Whiteboard” pattern [KRIENS] to receive `vCards` objects published by the *Vcard Receiver* bundle. Upon reception of these `vCard` objects this bundle shows the client the content of the received business card, so the user can add them to the address book.

5.2 Evaluation Points

In this section the lessons learned during the creation of the sample are outline. The relevant evaluation topics are presented.

23 Bluetooth is a wireless personal area network for connecting devices in near proximity [BLUETOOTH].
24 Vcard is a file format standard for electronic business cards.

Bundle Repositories

The OSGi Framework does no provided a centralized bundle repository. The OSGi Framework keeps track only of installed bundles and it has no notion of bundles that can be potentially installed.

However, some vendors are taking the initiative to provide a Maven-based[25] and Web-Based[26] repositories for OSGi bundles.

A bundle repository will be particularly useful for existing Java libraries, which can be retrofitted to work in an OSGi environment. The term *OSGi-ed library* is introduced here as a regular JAR file library with the appropriate import/export definitions in the manifest and without a bundle activator.

Conversely, many of these freely available libraries contain errors, which can lead to very difficult to debug errors at runtime. When errors occurs in libraries without visible dependencies in the manifest, this problem is even more exacerbated, and have to be debugged on a per needed ad-hoc basis.

In most cases, the absence of a bundle repository will lead to major confusions for developers as developers must either find existing bundles themselves or they may OSGi-ed JAR files themselves. Enabling a library for use in the OSGi framework is not difficult, but a tedious exercise.

The addition of a bundle repository will also enable a component based development process as outline by [CHA], which suggests the realization of a component repository to enable a component based development process.

25 Maven[MAVEN] is a software project management and comprehension tool. Maven can manage a project's build, reporting and documentation from a central piece of information.

26 Web-based access is given to users to browse through the bundles. However, no updating or uploading is allowed.

Libraries

The Java EE specification relies on the presence of libraries in the application server. Many of these libraries are distributed as JAR files, but assumed to be already presence in the server; examples of such libraries are the API libraries for servlets, JSP, JPA, JTA and other Java EE libraries. The OSGi definition of a bundle is a JAR file with meta information in the `META-INF/MANIFEST.MF` file. OSGi also assumes that all packages and their contents are private to the bundle, which mean using a library in the usual Java way is not possible in the OSGi Framework. Therefore, a comprehensive repackaging of all libraries was needed to include the appropriate meta information to make them work in an OSGi implementation. Luckily many vendors are starting to provide this "OSGi-ed" libraries and making them accessible in repositories, see the section on repositories for details regarding repositories.

For instance, an extract of the manifest of the Java Persistence API looks as:

```
...
Import-Package: javax.sql
...
Implementation-Vendor-Id: javax.persistence
Export-Package:
javax.persistence;version="1.0.0",javax.persistence.sp
 i;version="1.0.0";uses:="javax.persistence,javax.sql"
Bundle-Version: 1.0.0
...
```

Since some of the freely available bundle libraries contain errors. During the creation of the sample application some libraries had to be newly packed. Additionally, not all libraries can be found in those bundle repositories, here again a repacking of the library was needed, an example of such was to bluecove [BLUECOVE] library that provides an implementation of the *JSR-82 - Java API for Bluetooth* to use Bluetooth from within a Java application.

Development Environment

For the sample application the author select Eclipse for the development environment. Eclipse itself comes with very good support for OSGi bundle creation. However, the Eclipse tool set for OSGi development has some drawbacks:

- One can only test again a single target platform. If something in the target platform changes, one must reconfigure the whole development environment.
- No support for any other features but Java, not Java EE features at all.

Additionally, some of the bundles could not be developed efficiently as the tool set provided for bundle development does not include the editors for web resources like JSP pages, HTML pages and others.

Having said that, as of now the development tools available for OSGi development in the enterprise are considered to be in their infancy. However, new tools with better OSGi and Java EE integration will appear as OSGi becomes more mature in the Java EE problem domain.

Development Model

The OSGi development model is intrusive, because one has to adhere to the provides a comprehensive set of APIs to use OSGi within the application, e.g. `BundleContext`, `BundleActivator`, etc.

There is no clear development model for Java EE applications that use OSGi. This work was done using the native OSGi programming model. The advantage of using the OSGi development model is that developers are forced to thing in terms of modules and module dependencies. This is beneficial, because the modularization effort is not just in the design, but also in the programming model.

Nonetheless, there is an emerging programming model for OSGi application development, namely Spring-OSGi that allows the transparent – no need to know about OSGi – use of OSGi in Java. The Spring-OSGi programming model isolates developers from all OSGi APIs. Access to OSGi features can be keep transparent but, if needed, developers still have access to OSGi specific objects.

Spring-OSGi programming model can be used as a means to integrate OSGi into an existing Java EE environment in a transparent form.

More importantly, the *dependency injection* capabilities provided in the Spring Framework can be use to inject OSGi services into any Java class. In fact, these was done in the sample application in the web module. These represents a new paradigm in programming, namely *Service Injection*.

Web Application Support

OSGi provides an HTTP Service but this HTTP Service cannot be compared to powerful web containers provided by application servers. The use of OSGi's HTTP service is done programmatically and not declaratively. However, OSGi provides a declarative service that can be adjusted to support a declarative approach as known in Java EE applications.

A much better approach to web application support in OSGi is the realization of an extension of OSGi that allows the deployment of WAR file in it's current form. This extension can be easily be done using the extension capabilities in OSGi and the Extender [KRIENS2] design pattern. Thus, integrating WAR files in an OSGi environment.

Java EE Services Support

Java EE service can be easily modularized using OSGi as shown in the sample application.

The author found no major problems in the modularization of the JDBC service, JTA service, or JPA service. All services could be exposed assembled as bundles and published as OSGi services.

However, all implementations were done without distribution in mind. Services that rely on distribution present new challenges to the modularization effort as OSGi is single JVM module management system.

In particular, the integration of OSGi environment using a Enterprise Service Bus (ESB) would be interesting and requires more research. Such an integration will enable the realization of a federated OSGi environment.

Unified Packaging

Java EE supports 4 different packages structure, namely WAR, JAR (Application clients and EJB), RAR and EAR. All these packaging structures are nothing else than an extension to the JAR file format. Therefore, all this packages could be deployed as OSGi bundles.

In the sample application outline here, the single deployment structure use is a JAR file. Therefore, OSGi simplifies the packaging of Java Application by using a standardize format.

Different application servers use different deployment strategies and tools, because of the differences in their architectures. OSGi can solve, this issue if it is adopted as the deployment strategy in enterprise application. Why? Because the OSGi Framework standardizes the installation of modules and all of them are handled equally.

Management

Dynamics refers to the fact that services may attached and detached at any point in time. This must be addressed specifically by the developers using the White board patterns.

Ambiguity occurs when there are multiple candidates that may provide a specific service. The issue of ambiguity can be solved using versioning or using service properties, which was use in the vCard bundle to retrieve the file handles containing the business cards.

Architectural issues

The only architectural view provided by OSGi is a modular view that emphasizes on the dependencies between bundles.

This dependency view can be use to clarify deployment issues of Java EE applications.

OSGI Binding Models in Java EE Environment:

The author found three ways to integrate enable OSGi in a Java EE Environment. These are:

1. Use OSGi as a mean to modularized your Java EE application running OSGi inside a Java EE system. In this deployment model, the OSGi is contained within the WAR, JAR module. This deployment model is impossible for EJB as the EJB specification imposes strict class loading restriction.

2. Use OSGi at the bottom as a mean to provide the infrastructure to your entire Java EE Serve. In this approach the whole server is build on top of the OSGi Framework. This deployment structure provides the best architectural solution, as all Java EE components will be able to profit from bundles installed in the application server. It also supports a strict service oriented development model. This was also the solution used in this work. For instance, [JONAS] provide such an architecture.

3. Use OSGi at the top as a means to provide additional services to your Java EE component. In this architecture, the application server does not provide an OSGi Platform to Java EE applications, but allows the use of bundles within server.

In accordance to the goals of the thesis, these different binding models shows that the OSGi Platform is a very flexible platform.

Versioning

The OSGi Service Platform comes with strict wiring rules for modules base on modules. These versioning strategies allows the coexistence of multiple versions of the same modules. Hence, allowing the deployment of applications that rely on different versions of an API for instance.

Having said that the versioning control provided by OSGi takes completely care of the isolation of Java classes. In an OSGi environment two independently configured services that required one another and use different versions of the same API can run concurrently and therefore cooperate with each other seamlessly without any classpath class loading conflicts.

Testing

Testing of OSGi application can be done using existing test toolkits, such as Junit, Hamcrest, etc. During the elaboration of the sample application the only minor issue encountered was that in order to use these toolkits, more precisely Junit and Hamcrest,they need to be adapted to be able to run in an OSGi environment.

A positive finding of this work is that OSGi opens a new frontier in application testing,. OSGI allows the creation of modularized test, that can be dynamically installed and uninstall in an OSGi Platform to perform life tests on a system.

Extensibility

The OSGi Platform can be easily extended as the core of the platform provides the necessary mechanism to do so. As an example, one can use the extender example used in the realization of the web application to extends the capabilities of the framework to accept WAR files as a bundle.

6 Conclusion and Outlook

OSGi enables the development of applications that are so much more intertwined to the extend that one can develop functionality that can be either deployed at the client or at the server seamlessly providing a) the client comes with support for OSGi, like some newer cellular phones, or b) the client can be provisioned with an OSGi environment at runtime. In fact, it opens the possibility to just develop OSGi bundles without the need to determine a priori where the bundle will be deployed either on the server-side or on the client-side, thus allowing for a more fluid application architecture were a functionality can be deployed to best suited tier.

The current state of OSGi already allows the development of modularized, service-oriented enterprise applications in the Java EE domain. This modularized development model provides many advantages in terms of project management as many teams can be organized for the parallel development of modules in an application. Additionally, it provides for more efficient application development as new applications can easily be assembled with previously built modules into the new application.

OSGi forces developers to think about modularization due to its programming model. This represents very valuable because the modularization effort is not only in the development process, but also in the developed code itself.

OSGi opens a new frontier in application testing. It allows the creation of modularized test, that can be dynamically installed and uninstall in an OSGi Platform to perform life tests on a system.

Currently, Java EE application updates can be preform on a life system [JAVAEESPEC]. However, these module update in a Java EE application has to be performed in full. A Java EE application modularized with OSGi enables also partial updates of applications.

Versioning of Java EE applications is one the most important issues addressed by the OSGi platform, in particular making sure that there are no conflicts between different versions of the same software running on the same system. Thus preventing clashes between the different version of modules in the system, or different versions of the same libraries. The issue of versioning becomes more and more relevant as application server providers make extensive use of libraries also needed by the components they run. Versioning becomes more relevant as different application servers allow users to customize their servers by means of class loading manipulations, which is completely dependent on the type of server, which makes for very complicated custom deployment environments in different servers. This fact is so dramatic to the extend that application software providers must enforce the use of specific application servers in order to support their products, which has economic implications when it comes down to the end user/consumer of these applications footing the bill for it. OSGi solves the class loading chaos or discrepancies in different application servers.

OSGi lacks a component model. Therefore, Java EE components running in a OSGi environment provides developers with a strong service-oriented module oriented with a component architecture.

There is still work that needs to be done in the Integration of OSGi in the enterprise environment. These work areas relates to the standardization of component repository for bundles in application servers.

The Integration of an Enterprise Service Bus in an OSGi environment needs to be research to enable a more distributed architecture.

Another interesting direction regarding OSGi is the realization of OSGi based testing framework for life tests.

The use of Web Services (XML-based services) in an OSGi environment to expose OSGi services as Web services can also be studied further.

The Modularization of remoting capabilities (see section on interoperability) already present in application servers needs to be studied further. For instance, remoting using IIOP, HTTP, SSL, etc.

Finally, there is still much work to be done in terms of OSGi within - or in combination of - an enterprise application development. However, the OSGi service platform, in it's current state, can play a fundamental role in the simplification of Java EE applications in terms of application versioning, library/module dependency management and deployment of concurrent versions of the same application. This stresses the importance of a module management system for enterprise application.

References

Please note that Internet links are transitional. At the time of writing of this theses all links were navigable!

[APACHE]: , The Apache Software Foundation, http://www.apache.org

[ASKOXF]: Compact Oxford English Dictionary, http://www.askoxford.com

[BLUECOVE]: BlueCove (JSR-82 implementation), http://www.bluecove.org/

[BLUETOOTH]: Bluetooth Special Interest Group, http://www.bluetooth.org

[CARD]: David N. Card, Gerald T. Page, Frank E. McGarry, Criteria for Software Modularization, 1985

[CHA]: Jung-Eun Cha, Young-Jung Yang, Mun-Sub Song, Hang-Gon Kim. Design and implementation of component repository for supporting the component based development process

[ECLIPSE]: , Eclipse Foundation, Inc., http://www.eclipse.org

[EVANSDC]: Evans Data Corp - Market Research, http://www.evansdata.com/press/listReleases.php?view=archive

[EVANSSM]: Janet Hendrickson-Dalys, Enterprise Java Sweeps Into Small Businesses, New Evans Data Survey, http://www.evansdata.com/press/viewRelease.php?pressID=90

[GAMMA]: Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1994

[GOSLING]: James Gosling, Bill Joy, Guy Steele, Gilad Bracha, The Java Language Specification, 2005

[GRÖNE]: Bernhard Gröne, Andreas Knöpfel, Peter Tabeling, Component vs. component: Why We Need More Than One Definition, 2005

[HOLBREICH]: Alexander Holbreich, Java Class Loading, 2008

[HSQLDB]: Hypersonic SQL, http://www.hsqldb.org/

[IEEEGL]: The Institute of Electrical and Electronics Engineers, IEEE Standard Glossary of Software Engineering Terminology, 1990

[IIOP]: CORBA™/IIOP™ Specification

[JAVACONV]: Code Conventions for the Java Platform, http://java.sun.com/docs/codeconv/index.html

[JAROVER]: Java Archives (JAR) Files, http://java.sun.com/javase/6/docs/technotes/guides/jar/index.html

[JARSPEC]: JAR File Specification, http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html

[JAVAEESPEC]: Java Platform, Enterprise Edition (Java EE) Specification, v5 - JSR 244, 2006

[JCP]: The Java Community Process, http://www.jcp.org

[JEETECH]: Java EE Technologies, http://java.sun.com/javaee/technologies/

[JONAS]: JonAS - Java Open Application Server, http://jonas.objectweb.org

[KRIENS]: Peter Kriens, BJ Hargrave, Listeners Considered Harmful:The “Whiteboard” Patterns, 2004

[KRIENS2]: Peter Kriens, Extender Pattern with Automatic Servlet Registration

[LEGOOSGI]: Lego Mindstorms Robots http://r-osgi.sourceforge.net/mindstorms.html

[LIANG]: Sheng Liang, Gilad Bracha, Dynamic Class Loading in the Java Virtual Machine, 1988

[MAVEN]: Maven, http://maven.apache.org/

[OSGIAL]: OSGI Alliance, http://www.osgi.org

[OSGISERV]: The OSGi Service Platform, Service Compendum, 2007

[OSGISPEC]: The OSGi Service Platform, Core Specification, 2007

[RAMA]: Raghu Ramakrishnan, Johannes Gehrke, Database Management Systems, 2002

[RMI-IIOP]: Java RMI over IIOP, http://java.sun.com/products/rmi-iiop/

[RFC2068]: R. Fielding, U.C. Irvine, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, Hypertext Transfer Protocol -- HTTP/1.1, 1997

[SFORGE]: SourceForge, Inc. , http://www.sf.net

[SPRING]: SpringSource, Inc. , http://www.springframework.org

[SSAPPMTX]: The Server Side Application Server Matrix, http://www.theserverside.com/tt/articles/article.tss?l=ServerMatrix