Modularization of Existing Product Using OSGi Framework

Prachee Kane
Computer Science Engineering
MIT College of Engineering
Aurangabad,India
prachi.godbole@gmail.com

Dr. Radhakrishna Naik
Computer Science Engineering
MIT College of Engineering
Aurangabad,India
naikradhakrishna@gmail.com

***Abstract*—** this paper can act as a guideline to convert existing components or application to OSGi bundles. The prime motive for this conversion is to reorganize the existing monolithic, non modular code to component based architecture. OSGi is an open source framework that allows components to be easily integrated dynamically. OSGi provides several advantages such as modularity, isolation of components, sharing of components, easy induction of a component in the application at run time and easy removal of a component at run time. Existing products or incrementally developed products can exploit these features of OSGi. This paper first introduces OSGi and then discusses the areas that should be looked into while developing OSGi bundles, deploying bundles and converting existing programs to OSGi bundles**.**

***Keywords*—**component-based, OSGi, modularity

*****

I. INTRODUCTION

The keyword behind this thought process is "Modularity". The basic idea of an underlying modular design is to organize a complex system as a set of distinct components that can be developed independently and then plugged together. Modularity presents advantages like reuse of components, and replacement of components. in a simple plug-in and plug-out fashion. Modularity leads to ease of component management eventually leading to easy product management. The modularization involves two aspects. One is to generate separate functional components out of existing application and second is to modify or wrap it as OSGi bundles. The current paper limits itself to the second aspect. The first aspect is application specific and hence generalized suggestions cannot be specified in that regard. The considerations that should be made while creating bundles out of normal components are described here. The application in consideration should be developed using Java programming language. The bundles and other considerations are made assuming the same programming language. This introduction introduces OSGi framework followed by introduction to Java Interface Standard and Manifest file.

A. OSGi (Open Service Gateway Initiative)

OSGi provides a convenient and loosely coupled mechanism to integrate components.

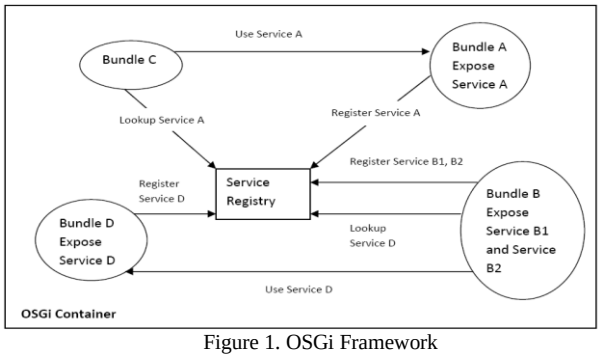

Figure 1. OSGi Framework

The OSGi framework is shown in Figure.1 above, it is a service platform for the Java programming language that implements a complete and dynamic component model. The entities involved in an OSGi Application are as follows –

- *Bundles*: Applications or components come in the form of bundles. Bundles are normal jar components with extra manifest headers. These bundles are dynamically loadable. The bundles can be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot. The life-cycle events of bundles are fired by the OSGi container.
- *Services:* The services layer connects bundles in a dynamic way by offering a publish-find-bind model for Plain Old Java Interfaces (POJI) or Plain Old Java Objects POJO. In other words, we can say that the bundles expose their functionalities in the form of services that can be availed by other bundles. Typically, a service includes an implementation (an instance of a class), one or more service interfaces under which the service is published, and a set of service properties. In the OSGi model, any Java class can be published as a service to be used by other bundles in the system.
- *Services Registry:* It provides the Application Program Interface (API) for management service. Services can register themselves in the service registry. Bundles can lookup services in this registry. This is a public registry indicating the use of whiteboard pattern in OSGi. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly.
- *OSGi Container/Framework*: This is the process that holds all the bundles and the service registry. It also manages the life cycle of the bundles. It isolates the bundles by a layer that defines encapsulation and declaration of dependencies (how a bundle can import and export code). Management of Java packages/classes is done by using different "ClassLoaders" for different bundles. The deployers can specify in the manifest file whether the classes in the bundles have to be shared

between bundles or should be encapsulated within the bundle.

The features of OSGi are described below

- Easy induction and removal of components – this is achieved through OSGi commands. The commands can be input from OSGi console. The component can be induced in the OSGi container using 'install" command. The component can be removed from OSGi container using "uninstall" command. The components can be activated or deactivated after installation using the "start" and "stop" command respectively. The commands can be input dynamically, even as the OSGi container is running and other components (bundles) are servicing the client. The newly added bundle makes itself available on activation. The clients can start using the services of the newly added bundle. The bundle can also start using the interface exposed by other existing bundles without a container shutdown.
- Isolation of components – The component *i.e* bundle is loaded using a separate class loader instance. In Java, the class loader is the mechanism that controls where class definitions come from and decides whether or not to allow a particular class definition to be used. So, when it comes to versioning and security, class loaders play a central role. Bundle class loader preserves the version of the classes used by the bundle and makes them available to the bundle without interfering with the class definition versions of other bundles.
- Sharing packages – OSGi container allows sharing of class definitions by providing the feature where one class loader (initial class loader) can delegate class loading to another class loader (effective class loader). The bundle attached to initial class loader can access the class definition present in the effective class loader.
- Modularity – Each bundle is considered to be a separate independent functional unit that can use services exposed by other bundles or expose services of its own. These modules can easily be plugged-in or plugged-out of the application.

OSGi containers can be advised about different configurations through a Manifest file. This file is included in the bundle jar installed in OSGi container.

B. Java Interface Standard

OSGi containers like Equinox, use Java language for developing the container as well as the components. The components *i.e* the bundles must follow the OSGi specification. The specification requires that the bundle should have certain callback methods as defined in the java interfaces of the specification. These methods will be invoked by the container on certain events.

C. Manifest File Standard

OSGi specification also mentions the structure and elements of the Manifest file. This file must be created by the component (bundle) developer. This file has all the dependency information related to the bundle. The developer must build this file according to the specification and include it in the bundle jar file. The OSGi container refers to this file when it deploys the bundle. This file is very crucial as it specifies the list of all jar files on which the current bundle depends. It also contains the classpath information. The services that must be activated before the current bundle is activated are also mentioned in this file. The information in this file enables the OSGi framework to confirm the availability of the services and bundles on which the current bundle is dependent. This file also contains information about which services and components are exposed by the current bundle.

II. DEVELOPING AND DEPLOYING OSGI BUNDLES

The following points should be considered while developing OSGi bundles.

A. Write Activator class

The specification provides a *BundleActivator interface*. The bundle developer must implement this interface. The OSGi container creates instances of a bundle's *BundleActivator* as required. If an instance's *BundleActivator.start* method executes successfully, it is guaranteed that the same instance's *BundleActivator.stop* method will be called when the bundle is to be stopped. *BundleActivator* is specified through the Bundle-Activator Manifest header. A bundle can only specify a single *BundleActivator* in the Manifest file. Fragment bundles must not have a *BundleActivator*. The specified *BundleActivator* class must have a public constructor that takes no parameters so that a *BundleActivator* object can be created by *Class.newInstance ()* when the "install" OSGi command is input to the OSGi container.

B. Write callback method for "start" command

The specification mentions that when the bundle is activated using the "start" command a callback method - void **start**(BundleContext context) - is invoked .

The developer must use this method to perform following actions –

- Register services.
- Lookup services and get the references.
- Initialize daemon threads.
- Handover the control to a main thread that will perform the functional aspect of the bundle.
- It is also recommended that all the threads should be added to a shared data structure for easy management.

C. Write callback method for "stop" command

The callback method invoked on deactivation of the OSGi bundle is the void stop (BundleContext context) method. This method should include the following things –

- Stop all the daemon and non-daemon threads created. Note that daemon threads must also be stopped explicitly as they will not terminate automatically

onEase of Use bundle deactivation. This is because even though the bundle is deactivated the process does not terminate.

- Give appropriate messages in log files.

D. Precautions while deployment

A meticulous documentation is required stating the packages imported and exported. This must include the version number of the packages. Appropriate versioning of packages is the key to exploit the OSGi framework without getting hassled with class loader exceptions.

It is suggested that for every OSGi container instance one manifest document should be maintained by the deployer. A thorough document of the services exposed must be created by the class designer. This should mention the service interface details, the details about the parameters passed. The functionality and pre and post conditions of the service interface methods.

The deployer should know about the parameter classes used by the service interface of the bundle. The deployer must export the packages of the parameter classes to avoid ClassCastException at runtime due to different class loader versions of the same class.

III. GUIDELINES TO CONVERT AN EXISTING APPLICATION OR A COMPONENT TO OSGI BUNDLE

A. Expose Interface Service

The functionality of the existing component or application that is used by other components or application should be identified. These functionalities are nothing but the services of the upcoming bundle. These functionalities must be added to one or more interfaces using the Interface Segregation Principle (ISP) .These interfaces should be registered as services using the "registerService" API of the BundleContext.

While deciding the parameters of the service interface methods, care should be taken to choose appropriate data type of the method parameters and return types. It should be noted that the parameters could either be primitive types or Serializable type. No error notification is given if Non-Serializable parameters are passed to the methods. However, when the service is accessed by other remote components (i.e from outside the framework) a runtime exception may occur

B. Expose Packages

OSGi container provides a separate classloader to every bundle. The bundle deployer can export the packages used by a bundle. This will enable other bundles to access the class instance loaded by the exporting bundle. As a guideline the bundle should export those packages that contain classes used as method-parameter-types in service interface. This will avoid the classloader mismatch errors at run time.

C. Rewrite thread design

The thread design of the component must accommodate the OSGi framework design that the bundle can be stopped and started multiple times within the same process. This implies that the threads spawned within the bundles must also be stopped and started with the bundle. Java threads do not provide inbuilt APIs for stopping the threads. Thus the bundle designer must take care of this requirement and provide a design that can keep track of all the threads created by the bundle. Following suggestions may be useful for the designer for the above scenario.

- Extend all threads from a base thread class called as ThreadBase. This class should maintain a list of all the threads created in the bundle. This list can be populated each time the constructor is invoked. This class should also have a stop method that resets a flag. All threads should refer to this flag as they iterate in a loop. The stop method mentioned above should be called for all the threads through the Activator stop callback described above.
- Suspend the threads on Activator stop callback and interrupt each thread on Activator start callback.

D. RMI consideration

The bundle can be accessed by other bundles in the same framework or from other java components outside the framework. The programmer can handle by making provision for access through OSGi framework and also through RMI.

Expose the bundle service as an OSGi service. This will enable the bundles internal to the framework to access the services without RMI overhead.

Register a remote object through RMI registry, such that the remote object exposes the same services as the bundle. This will enable the java components outside the framework to access the services through RMI.

IV. CONCLUSION

Though OSGi is introduced for many years now, its use in the small project market is not as wide. The current effort encourages projects to switch over to OSGi frameworks as they are open source, freely available and above all provides a much needed modularity to an application design. Small projects developed for small institutions and organizations internally are normally monolithic in design and are not suitable for incremental development. Shifting to OSGi frameworks can help such projects. Modules can be developed gradually and integrated into OSGi framework. New modules can be added over a period of time and easily integrated with the existing application by way of OSGi. Thus OSGi imparts design and easy integration of new modules to amateur projects.

REFERENCES

[1] R. S. Hall, "OSGi Implementation and Experience Report,", Consumer Communications and Networking Conference First IEEE , 2004.

[2] OSGi Alliance, "About the OSGi Service Platform", White Paper June, 2007.

[3] Lev Kozakov, Mirko Jahn, Yuradaer Doganata, "OSGi Enablement in UIMA Framework," IBM Research Report July, 2007.

[4] OSGi Alliance, "OSGi and Core Release 5", OSGi Alliance, 2012.

[5] "OSGi and Equinox Creating Highly Modular Java Systems", Jeff McAffer , Paul VanderLei , Simon Archer, eclipseRT the eclipse series, Second Edition.

[6] Chris Aniszczyk, Bern Kolb, Martin Lippert, "OSGi for Eclipse Developers", Slideshare, April , 2009

[7] "The Unified Modeling Language User Guide", Grady Booch, James Rumbaugh, and Ivar Jacobson, Addison Wesley, Second Edition.

[8] "System Analysis and Design", A Dennis, B.H. Wixom, R.M. Roth, Wiley Publication, Fourth Edition.

[9] Irina Astrova, Arne Koschel, Björn Siekmann, Mark Starrach, Christopher Tebbe, Stefan , "OSGi in Cloud Environments", World Academy of Science Engineering and Technology, 2013