M. A. Kadan

OSGI SERVICE PLATFORM

The OSGi specification describes a small layer that allows multiple Java based components to efficiently cooperate in a single Java Virtual Machine. It provides an extensive security model so that components can run in a shielded environment. However, with the proper permissions, components can reuse and cooperate, unlike other Java application environments. The OSGi Framework provides an extensive array of mechanisms to make this cooperation possible and secure.

Introduction

Today, software development largely consists of adapting existing functionality to perform in a new environment. In the last 20 years, a large number of standard building blocks have become available and they are heavily used in today's products; a prime example is the success of open software. However, the use of these libraries is not without problems. Integrating many different libraries can be daunting because many libraries have become complex and require their own libraries to function – even if that functionality is never needed for the product. This trend requires monolithic software products to undergo a heavy testing cycle. Add unsynchronized evolution of the different libraries and it becomes clearer why software development is so costly today.

OSGi technology [1, 2] is the dynamic module system for Java. The OSGi Service Platform provides functionality to Java that makes Java the premier environment for software integration and thus for development. Java provides the portability that is required to support products on many different platforms. The OSGi technology provides the standardized primitives that allow applications to be constructed from small, reusable and collaborative components. These components can be composed into an application and deployed.

The purpose of this article is to describe such a new trend in software development as composing products from relatively small and often unrelated modules and allowing them to communicate via provided services. OSGi specification fully covers this requirements thus providing a possibility to implement modular service-oriented architecture to be used in modern world of rapid and fail-safe software.

OSGi Alliance

The OSGi Alliance [3] is an open standards organization founded in March 1999. Its mission is to create open specifications for the network delivery of managed services to local networks and devices. The OSGi Service Platform specification delivers an open, common architecture for service providers, developers, software vendors, gateway operators and equipment vendors to develop, deploy and manage services in a coordinated fashion. It enables an entirely new category of smart devices due to its flexible and managed deployment of services.

The Alliance and its members have specified a Java-based service platform that can be remotely managed. The core part of the specifications is a framework that defines an application life cycle management model, a service registry, an Execution environment and Modules. Based on this framework, a large number of OSGi Layers, APIs, and Services have been defined. Briefly, OSGi technology provides a service-oriented plug-in-based platform for application development.

OSGi Framework Overview

The Framework forms the core of the OSGi Service Platform Specifications. It provides a general-purpose, secure, and managed Java framework that supports the deployment of extensible and downloadable applications known as bundles. Each bundle is a tightly-coupled, dynamically loadable collection of classes, jars, and configuration files that explicitly declare their external dependencies (if any).

OSGi-compliant devices can download and install OSGi bundles, and remove them when they are no longer required. The Framework manages the installation and update of bundles in an OSGi environment in a dynamic and scalable fashion. To achieve this, it manages the dependencies between bundles and services in detail. It provides the bundle developer with the resources necessary to take advantage of Java's platform independence and dynamic code-loading capability in order to easily develop services for small-memory devices that can be deployed on a large scale.

The OSGi Framework implements a complete and dynamic component model, something that is missing in standalone Java/VM environments. Applications or components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot; management of Java packages/classes is specified in great detail. Life cycle management is done via APIs which allow for remote downloading of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly.

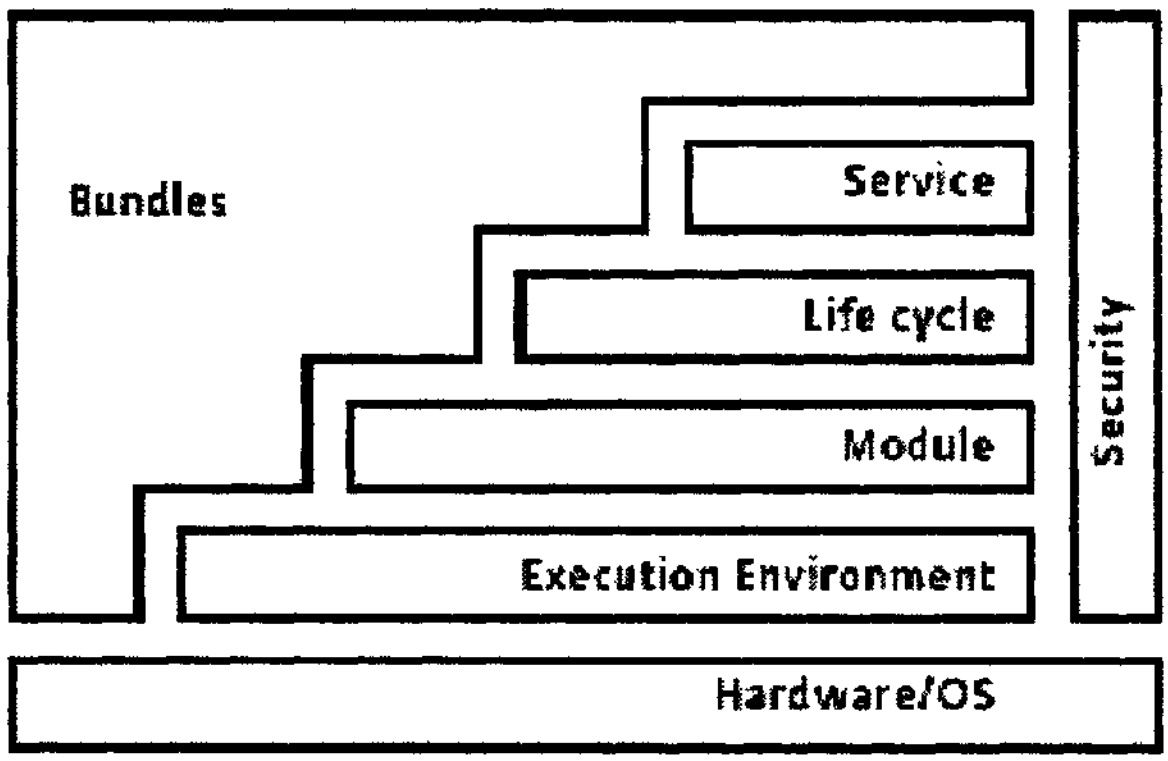

Figure 1. OSGi Framework Structure

The framework is conceptually divided into the following areas (Figure 1):

- Bundles – Bundles are normal jar components with extra manifest headers.
- Services – The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects (POJOs). Also it contains the API for management services (ServiceRegistration, ServiceTracker and ServiceReference).
- Life-Cycle – The API for life cycle management (install, start, stop, update, and uninstall bundles).
- Modules – The layer that defines encapsulation and declaration of dependencies (how a bundle can import and export code).
- Security – The layer that handles the security aspects by limit bundle functionality to pre-defined capabilities. The Security Layer is based on Java 2 security [4] but adds a number of constraints and fills in some of the blanks that standard Java leaves open. It defines a secure packaging format as well as the runtime interaction with the Java 2 security layer.
- Execution Environment – Defines what methods and classes are available in a specific platform. There is no fixed list of execution environments, since it is subject to change as the Java Community Process creates new versions and editions of Java.

A consistent programming model helps bundle developers cope with scalability issues in many different dimensions – critical because the Framework is intended to run on a variety of devices whose differing hardware characteristics may affect many aspects of a service implementation. Consistent interfaces insure that the software components can be mixed and matched and still result in stable systems.

Module Layer

The Module Layer defines a modularization model for Java. It addresses some of the shortcomings of Java's deployment model. The modularization layer has strict rules for sharing Java packages between bundles or hiding packages from other bundles. The standard Java platform provides only limited support for packaging, deploying, and validating Java-based applications and components. Because of this, many Java-based projects, such as JBoss and NetBeans, have resorted to creating custom module-oriented layers with specialized class loaders for packaging, deploying, and validating applications and components. The OSGi Framework provides a generic and standardized solution for Java modularization.

The Framework defines a unit of modularization, called a bundle. A bundle is comprised of Java classes and other resources, which together can provide functions to end users. Bundles can share Java packages among an exporter bundle and an importer bundle in a well-defined way. In the OSGi Service Platform, bundles are the only entities for deploying Java-based applications.

A bundle can carry descriptive information about itself in the manifest file that is contained in its JAR file under the name of META-INF/MANIFEST.MF. The Framework defines OSGi manifest headers such as Export-Package and Bundle-Classpath, which bundle developers use to supply descriptive information about a bundle.

Class Loading Architecture

Many bundles can share a single virtual machine (VM). Within this VM, bundles can hide packages and classes from other bundles, as well as share packages with other bundles. The key mechanism to hide and share packages is the Java class loader that loads classes from a sub-set of the bundle-space using well-defined rules. Each bundle has a single class loader. That class loader forms a class loading delegation network with other bundles.

A *class space* (Figure 2) is then all classes reachable from a given bundle's class loader. A class space must be consistent, such that it never contains two classes with the same fully qualified name (to prevent Class Cast Exceptions). However, separate class spaces in an OSGi Platform may contain classes with the same fully qualified name. The modularization layer supports a model where multiple versions of the same class are loaded in the same VM.

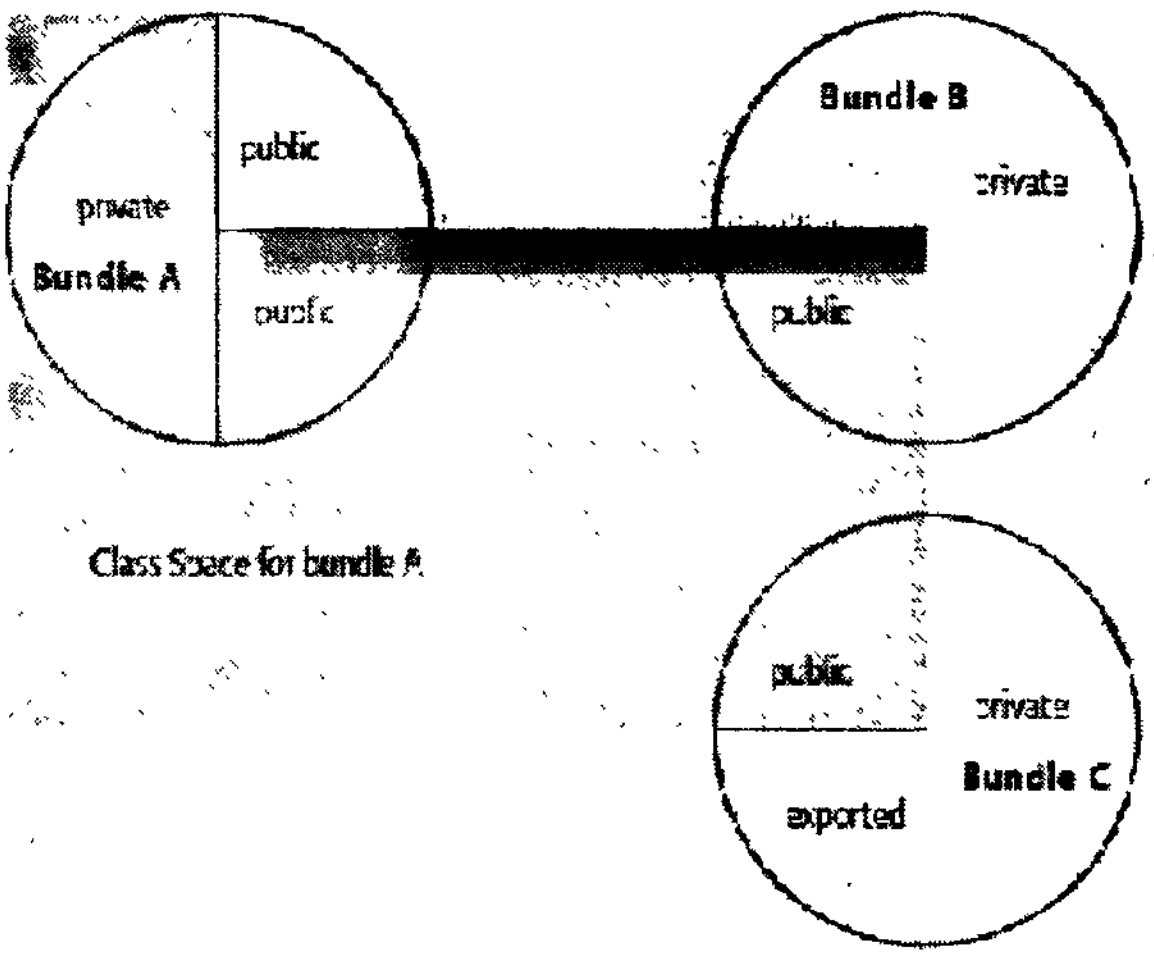

Figure 2. Class Space

The Framework therefore has a number of responsibilities related to class loading. Before a bundle is used, it must resolve the constraints that a set of bundles place on the sharing of packages. Then select the best possibilities to create a wiring.

Resolving is the process where importers are wired to exporters. Resolving is a process of satisfying constraints. This process must take place before any code from a bundle can be loaded or executed. A wire is an actual connection between an exporter and an importer, which are both bundles. A wire is associated with a number of constraints that are defined by its importer's and exporter's manifest headers. A valid wire is a wire that has satisfied all its constraints. The class structure of the wiring model is presented on Figure 3.

Life Cycle Layer

The Life Cycle Layer provides a life cycle API to bundles. This API provides a runtime model for bundles. It defines how bundles are started and stopped as well as how bundles are installed, updated and uninstalled. Additionally, it provides a comprehensive event API to allow a management bundle to control the operations of the service platform. The Life Cycle Layer requires the Module Layer but the Security Layer is optional.

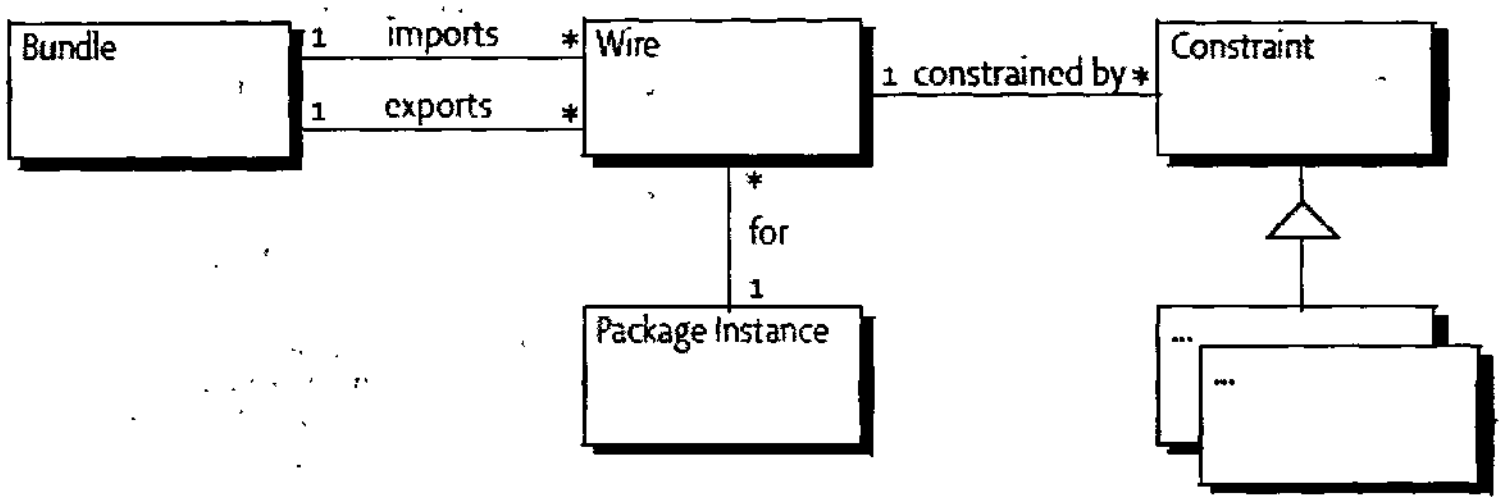

Figure 3. Class structure of wiring

A bundle represents a JAR file that is executed in an OSGi Framework. The class loading aspects of this concept were specified in the Module Layer. However, the Module Layer does not define how a bundle is installed, updated, and uninstalled. These life cycle operations are defined here. The installation of a bundle can only be performed by another bundle or through implementation specific means (for example as a command line parameter of the Framework implementation).

A Bundle is started through its Bundle Activator. This interface has a start and stop method that is used by the bundle programmer to register itself as listener and start any necessary threads. The stop method must clean up and stop any running threads. Upon the activation of a bundle, it receives a Bundle Context.

For each bundle installed in the OSGi Service Platform, there is an associated Bundle object. The Bundle object for a bundle can be used to manage the bundle's life cycle. This is usually done with a Management Agent, which is also a Bundle.

A bundle is identified by a number of names that vary in their scope:

- Bundle identifier – A long that is a Framework assigned unique identifier for the full lifetime of a bundle, even if it is updated or the Framework is restarted. Its purpose is to distinguish bundles in a Framework. Bundle identifiers are assigned in ascending order to bundles when they are installed.
- Bundle location – A name assigned by the management agent (Operator) to a bundle during the installation. This string is normally interpreted as a URL [5, 6] to the JAR file but this is not mandatory. Within a particular Framework, a location must be unique. A location string uniquely identifies a bundle and must not change when a bundle is updated.
- Bundle Symbolic Name – A name assigned by the developer. The combination of Bundle Version and Bundle Symbolic Name is a globally unique identifier for a bundle.

A bundle can be in one of the following states (Figure 4):

- INSTALLED – The bundle has been successfully installed.

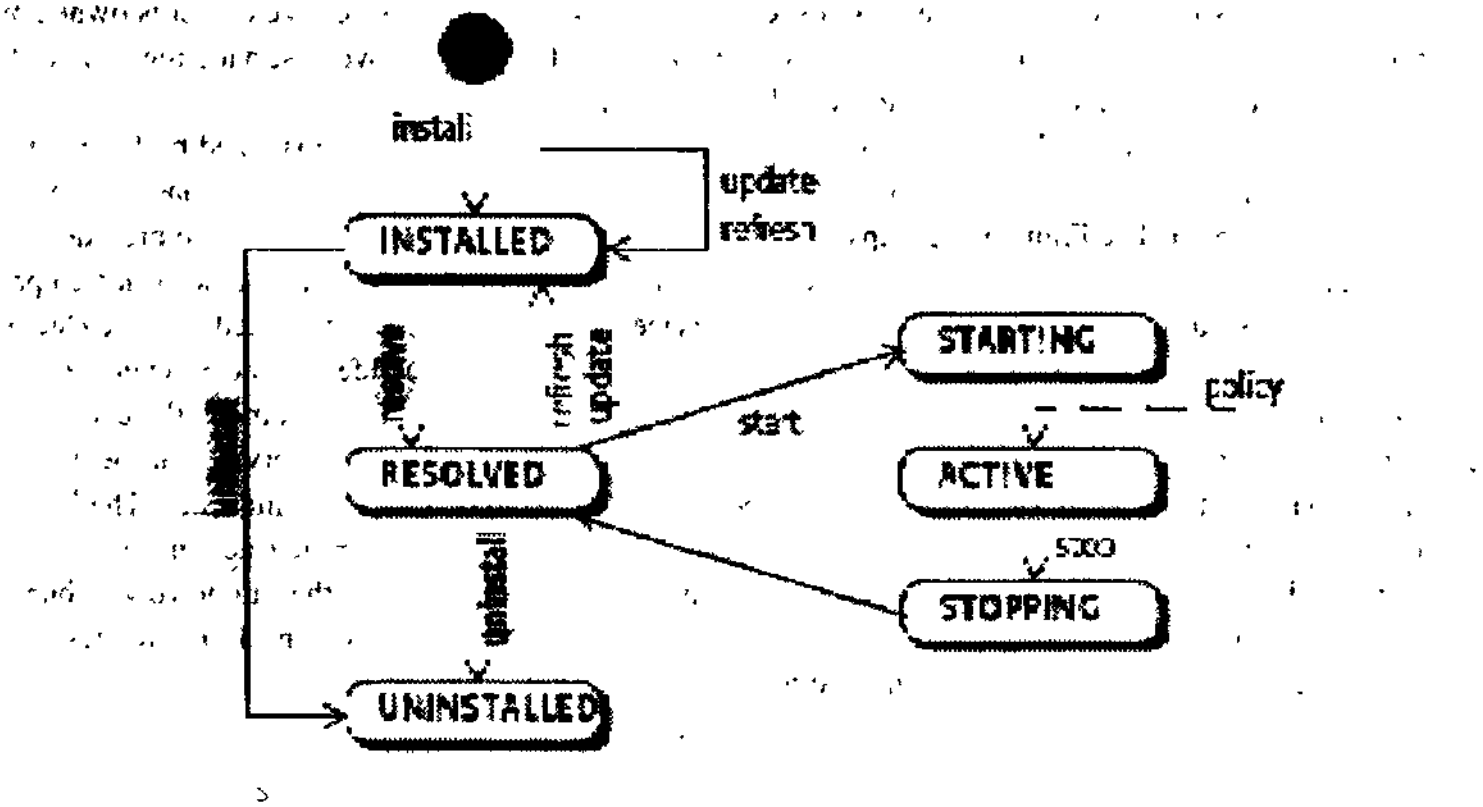

Figure 4. Bundle States

- RESOLVED – All Java classes that the bundle needs are available. This state indicates that the bundle is either ready to be started or has stopped.
- STARTING – The bundle is being started, the BundleActivator's start method will be called, and this method has not yet returned.
- 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's stop method has been called but the stop method has not yet returned.
- UNINSTALLED – The bundle has been uninstalled. It cannot move into another state.

When a bundle is installed, it is stored in the persistent storage of the Framework and remains there until it is explicitly uninstalled. Whether a bundle has been started or stopped must be recorded in the persistent storage of the Framework. A bundle that has been persistently recorded as started must be started whenever the Framework starts until the bundle is explicitly stopped.

In addition to normal bundles, the Framework itself is represented as a bundle. The bundle representing the Framework is referred to as the system bundle. Through the system bundle, the Framework may register services that can be used by other bundles.

Service Layer

The Service Layer provides a dynamic, concise and consistent programming model for Java bundle developers, simplifying the development and deployment of service bundles by de-coupling the service's specification (Java interface) from its implementations. This model allows bundle developers to bind to services only using their interface specifications. The selection of a specific implementation, optimized for a specific need or from a specific vendor, can thus be deferred to run-time through the service registry. Bundles register new services, receive notifications about the state of services, or look up existing services to adapt to the current capabilities of the device. This aspect of the Framework makes an installed bundle extensible after deployment: new bundles can be installed for added features or existing bundles can be modified and updated without requiring the system to be restarted.

The OSGi Service Layer defines a dynamic collaborative model that is highly integrated with the Life Cycle Layer. The service model is a publish, find and bind model. A service is a normal Java object that is registered under one or more Java interfaces with the service registry. Bundles can register services, search for them, or receive notifications when their registration state changes.

In the OSGi Service Platform, bundles are built around a set of cooperating services available from a shared service registry. Such an OSGi service is defined semantically by its service interface and implemented as a service object. The service interface should be specified with as few implementation details as possible. OSGi has specified

many service interfaces for common needs and will specify more in the future. The service object is owned by, and runs within, a bundle. This bundle must register the service object with the Framework service registry so that the service's functionality is available to other bundles under control of the Framework.

Dependencies between the bundle owning the service and the bundles using it are managed by the Framework. For example, when a bundle is stopped, all the services registered with the Framework by that bundle must be automatically unregistered. The Framework maps services to their underlying service objects, and provides a simple but powerful query mechanism that enables a bundle to request the services it needs. The Framework also provides an event mechanism so that bundles can receive events of services that are registered, modified, or unregistered.

A service interface is the specification of the service's public methods. In practice, a bundle developer creates a service object by implementing its service interface and registers the service with the Framework service registry. Once a bundle has registered a service object under an interface name, the associated service can be acquired by bundles under that interface name, and its methods can be accessed by way of its service interface. The Framework also supports registering service objects under a class name, so references to service interface in this specification can be interpreted to be an interface or class. When requesting a service object from the Framework, a bundle can specify the name of the service interface that the requested service object must implement. In the request, the bundle may also specify a filter string to narrow the search.

Summary

The OSGi specifications are so widely applicable because the platform is a small layer that allows multiple Java based components to efficiently cooperate in a single Java Virtual Machine. It provides an extensive security model so that components can run in a shielded environment. However, with the proper permissions, components can reuse and cooperate, unlike other Java application environments. The OSGi Framework provides an extensive array of mechanisms to make this cooperation possible and secure.

OSGi technology adopters benefit from improved time-to-market and reduced development costs because OSGi technology provides for the integration of pre-built and pre-tested component subsystems. The technology also reduces maintenance costs and enables unique new aftermarket opportunities because components can be dynamically delivered to devices in the field.

The presence of OSGi technology-based middleware in many different industries creates a large software market for OSGi software components. The rigid definition of the OSGi Service Platform enables components that can run on a variety of devices, from very small to very big. Adoption of the OSGi specifications can therefore reduce software development costs as well as provide new business opportunities.

References

1. OSGi Service Gateway Specification R4, October 2005 [Electronic resource] – OSGi Alliance, 2005. – Mode of access: http://www.osgi.org/resources/spec_download.asp. – Date of access: 12.02.2009.

2. OSGi Service Gateway Specification R4.1, May 2007 [Electronic resource] – OSGi Alliance, 2007. – Mode of access: http://www.osgi.org/resources/spec_download.asp. – Date of access: 12.02.2009.

3. About / OSGi Technology [Electronic resource] – OSGi Alliance. – Mode of access: http://www.osgi.org/About/Technology. – Date of access: 15.02.2009.

4. Java 2 Security Architecture, Version 1.2 [Electronic resource] - Sun Microsystems. – Mode of access: http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html. – Date of access: 16.02.2009.

5. Uniform Resource Identifiers URI: Generic Syntax [Electronic resource] – The Internet Society, 1998. - Mode of access: http://www.ietf.org/rfc/rfc2396.txt. - Date of access: 17.02.2009.

6. Uniform Resource Locators [Electronic resource] – The Internet Society, 1994. – Mode of access: http://www.ietf.org/rfc/rfc1738.txt. – Date of access: 17.02.2009.

Кадан Михаил Александрович, компания «InstinctTools», Гродно, магистр технических наук, mkadan@gmail.com