RC24308 (W0707-101) July 16, 2007
Computer Science

IBM Research Report

OSGi Enablement in UIMA Framework

Lev Kozakov, Mirko Jahn, Yurdaer Doganata
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

OSGi enablement in UIMA Framework

Lev Kozakov, Mirko Jahn and Yurdaer Doganata

IBM T.J.Watson Research Center

{kozakov, mjahn, doganata}@us.ibm.com

Abstract

Migration of existing software systems to OSGi is gaining momentum with wide acceptance of the OSGi technology as a standard way of managing software life-cycle through a service-oriented component-based approach. Although the technical merits of the OSGi approach are broadly recognized, the migration of existing systems may require significant efforts, such as modularizing the system code and embracing service based design. An alternative approach is adapting existing software for component based OSGi environment by putting a thin layer between the OSGi Service Platform and existing software systems. Adaptation, as opposed to migration, eliminates the necessity of modularizing or redesigning existing systems, which is particularly important when the existing software platform and associated programming model is mature and has already penetrated into the market place. As a case study of such an adaptation approach, this report demonstrates an OSGi adaptation of a popular open source Java platform built by IBM Research for integration of software analytics, processing unstructured information. The major challenge is providing analytic and application developers with all the benefits of the component based OSGi platform without entailing substantial modifications of existing framework code base and/or programming model. This report presents a comprehensive solution for integrating analytics and other resources in OSGi environment. The results of this report are applicable not only to a particular framework, but also to a wider class of Java frameworks.

1. Introduction

The OSGi Platform is a widely adopted dynamic module system for Java, which is accepted today as a standard way of managing software life-cycle through a service-oriented component-based approach. The OSGi Technology is considered as the most recent and successful step towards software product lines and industrial reuse of software components. It is expected that through adoption of the OSGi Technology software development and maintenance costs will be significantly reduced. The list of mainstream software products already adopted the OSGi Technology include Eclipse IDE, Spring Framework, IBM WebSphere, Lotus Sametime, Tivoli and others (see [4]).

The main focus of the OSGi Technology lies in the integration of software components based on the OSGi Service Platform, which defines a unit of modularization, called a bundle. The OSGi Framework provides a container for managing bundles and supports a full set of bundle lifecycle management operations. The modularization concept in the OSGi Framework is supported by the module-based class loading policy. The OSGi Framework implements a powerful and rigidly specified class loading model that allows modules to declare a shared and private class space and controls linking between modules. In this model, each bundle has its own class loader, and forms a class loading delegation network with other bundles. The OSGi Service Platform also defines a dynamic collaborative 'find-and-bind' service model, where services are normal Java objects 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.

Although the technical merits of OSGi approach are broadly recognized, the migration of existing systems may require significant efforts, such as modularizing the system code and shifting toward a service based design. An alternative approach is adapting existing software for OSGi environment by putting a thin layer between the OSGi Service Platform and an existing software system. Adaptation, as opposed to migration, eliminates the necessity of modularizing or redesigning the existing system, which is particularly important when the existing software platform and associated programming model is mature and has already penetrated into the market place.

This report presents the case study of such an adaptation approach - OSGi adaptation of the Unstructured Information Management Architecture (UIMA) Framework. The framework is a popular open source Java platform built by IBM Research for integration of software analytics, processing unstructured information. The UIMA platform is designed as component software architecture, but the notion of components in the UIMA Framework is not compatible with that of OSGi. Analysis components in UIMA are logical entities referred by their declarative descriptors and usually deployed in a pre-specified *application class space.* In OSGi, however, class space is dynamically managed by the container and the components are physical entities, encapsulated in JAR files, called bundles.

OSGi adaptation enables the UIMA Framework to run analytics and other resources encapsulated in OSGi bundles, transforming the framework into an industrial standard component based development and integration platform, fully interoperable with other OSGi products. OSGi enablement improves asset management, adds dynamism, and facilitates embedding and process automation. At the same time, it preserves current UIMA application programming model and makes the underlying OSGi technology almost transparent to UIMA application developers. The major challenge is providing analytic and application developers with all the benefits of the OSGi Service Platform without entailing substantial modifications of existing framework code base and/or programming model. As a solution, this report introduces the OSGi Service Adaptor, which provides a thin enablement layer between the OSGi Service Platform and UIMA applications. The report presents a comprehensive solution for integrating UIMA components in OSGi environment. The results of this report are applicable to a wider class of Java integration frameworks.

The report is organized as follows:

- Section 2 highlights the features of the OSGi technology that are most relevant in the context of this report;
- Section 3 presents the main features of the UIMA technology;
- Section 4 describes the goals and elements of the OSGi enablement in UIMA;

- Section 5 contrasts two approaches to OSGi enablement – adaptation vs. migration;
- Section 6 anatomizes the OSGi Service Adaptor – thin adaptation layer that converts the UIMA Framework into an OSGi compatible product;
- Section 7 highlights the process of creating analytic and other component bundles;
- Section 8 highlights the features of application development in the OSGi enabled UIMA Framework;
- Section 9 presents related work;
- Section 10 concludes the report;
- Appendix contains the implementation diagram showing bundle dependencies.

2. OSGi technology highlights

The OSGi Alliance [1] makes the following statement: "OSGi Technology is the dynamic module system for Java". Today, this technology is widely adopted by both the Java community (see [2], [3]) and the major software vendors (see, for instance, [4]). The main focus of the OSGi Technology lies in the integration of software components based on the OSGi Service Platform. A comprehensive overview of the OSGi Service Platform is given by the OSGi Alliance in [5]. In this section we will briefly review several key features of the OSGi Service Platform that are most relevant in the context of this report.

Bundles and bundle management

In the OSGi Service Platform, a bundle is defined as a unit of modularization. OSGi bundles are JAR files comprised of Java classes and other resources without any specific structural constraints. Unlike other JAR files, OSGi bundles always include a manifest file, providing information about the bundle. This file consists of headers, which specify information that the OSGi Framework utilizes to correctly install, resolve and activate a bundle. For example, some manifest headers are used to state dependencies on other resources that must be available before a bundle is activated. In short, the OSGi Framework provides a container for managing bundles and supports a full set of bundle lifecycle management operations, such as installing, updating, resolving, starting, stopping and uninstalling.

Modules and class loading

The modularization concept in the OSGi Framework is supported by the module-based class loading policy. The OSGi Framework implements a powerful and rigidly specified class loading model that allows modules to declare a shared and private class space and controls linking between modules. In this model, each bundle has its own class loader, and forms a class loading delegation network with other bundles (see [5], section 3.4). A bundle class space includes imported packages, a shared class space of required bundles and private bundle classes. Note, that a consistent bundle class space cannot include two classes with the same fully qualified name. However, separate class spaces in the OSGi Framework may contain classes with the same fully qualified name. Therefore, the OSGi Framework supports a model where multiple versions of the same class are loaded in the same VM, which allows binding two versions of the same component within the same application. Another important OSGi concern is the case where a package comes from multiple sources a.k.a. *split package* case. To warn against using split packages, the OSGi Specifications added the constraint that a package must have a single source.

Services and service registry

The OSGi Service Platform defines a dynamic collaborative 'find-and-bind' service model (see [5], chapter 5). A service in OSGi Platform 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. The service registry provides a comprehensive model to share objects between bundles.

3. UIMA technology highlights

UIMA has been designed by IBM Research as integration platform for broad set of multi-modal software analytics, processing unstructured information in order to extract certain structured data or knowledge (see [6]-[10]). Conceptually, this platform can be characterized by the two major features: (1) declarative descriptors specifying metadata pertaining to each resource, such as an analysis engine or linguistic resource, and (2) common data structure, named Common Annotation Structure (CAS), which is used to exchange analysis results between analytics. UIMA is realized as an extensible and scalable framework that supports an application through all information processing stages from acquisition of original unstructured information to its analysis and, ultimately, to the utilization of results.

UIMA positions itself as “component software architecture” (see [7]), but the notion of component in the UIMA Framework has rather a declarative nature. The framework operates with declarative descriptors of analytical ‘components’, leaving all the operational environment issues, such as organizing the class space for Java resources or setting environment variables and system options, to the application. Declarative descriptors of UIMA analytics refer to other descriptors and resources that are expected to be found in the application class space. The UIMA Java Framework utilizes two basic mechanisms for resolving local references: (1) using the resource file path in the local file system and (2) using the application class space or custom context class loader to locate the resource. In the second case, the application is fully responsible for setting the class space boundaries, like adding certain folders or JAR files to the class path.

Analytics in the UIMA Framework are integrated at the level of declarative descriptors. All analytics, included in the analysis pipe, are automatically deployed and run by the framework, and the results of the analysis are automatically passed from one node to another in CAS objects. Technically, the integration of analytics in the UIMA Framework is done by constructing an aggregate declarative descriptor, which refers to declarative descriptors of other (delegate) analytics. Such an aggregate descriptor may, in its turn, be a subject for further integration in a more complicated aggregate descriptor. As we mentioned earlier in this section, the references are resolved in the local file system by using either the descriptor file path or the Java class space constructed by the application.

UIMA analytics produce the results of analysis in a form of typed objects (annotations) with valued attributes. The types of annotation objects are organized in a hierarchical annotation Type System (TS). The UIMA Framework defines few built-in primitive types, like Integer or String, and all annotation types are constructed from these primitive types. UIMA analytics can reuse or extend existing type systems or define their own types independently of any existing type system. Annotation type systems in UIMA Framework are declared by using TS descriptors.

The CAS system manages the organization and storage of all typed annotations in CAS objects. To facilitate the access to CAS objects for Java developers the UIMA Framework provides a Java based object-oriented programming interface to the CAS, named JCas (see [11]). The JCas mechanism generates JCas TS classes directly from the TS descriptors and allows developers to treat typed annotations as regular Java objects.

4. The goals of OSGi enablement in UIMA

The primary goal of the OSGi enablement is to transform the UIMA Framework into an industrial standard component based development and integration platform. The OSGi technology brings a variety of advantages to the UIMA Framework by extending the framework capabilities to share and run analytics encapsulated in OSGi bundles. It also improves asset management, adds dynamism, and facilitates embedding and process automation. The secondary, but not less important consideration of the OSGi enablement is preserving current UIMA application programming model and making the underlying OSGi technology transparent to UIMA application developers. Such an approach is particularly important for UIMA – a mature software platform, which has already penetrated into the market place and is characterized by a stable programming model.

In more details, the elements of OSGi enablement in UIMA are as follows:

- Encapsulation of UIMA analytics and other resources into OSGi component bundles
  - creating, distributing and managing UIMA components as reusable OSGi bundles;
  - component bundles can encapsulate UIMA analytic(s) or other reusable resources, like type systems, dictionaries, libraries, etc.
- Integration of UIMA analytics by reference to associated component bundles
  - aggregate analytic component can refer to the resources of its delegate component bundles rather then copying all required resources to the aggregate component bundle itself;
  - built-in version management and life cycle support for analytic components is provided by the OSGi container.
- Running UIMA analytics as OSGi bundles
  - UIMA Framework can deploy and run analytics, encapsulated in OSGi component bundles and managed by the OSGi container.
- Enabling dynamic deployment of UIMA component bundles
  - UIMA applications can discover and dynamically deploy UIMA analytics, encapsulated in OSGi component bundles.

OSGi components are more robust and reliable than plain Java objects, because they are effectively isolated from the application environment. The components are managed by the OSGi container, so that applications no longer need to be aware of the physical location of component bundles and of the component dependencies. Such isolation from the application environment facilitates embedding of UIMA component bundles into different products and application frameworks, like J2EE containers, RCP applications, etc.

The following example (see Figure 1) illustrates seamless embedding of OSGi bundles into the J2EE container, as described in the recently graduated Eclipse server side project [13]. Integration of 3rd party components into the Web application is done automatically as soon as the component bundles are resolved and activated by the OSGi container. The application servlet is also a bundle registered to the OSGi container by using HTTP service APIs.

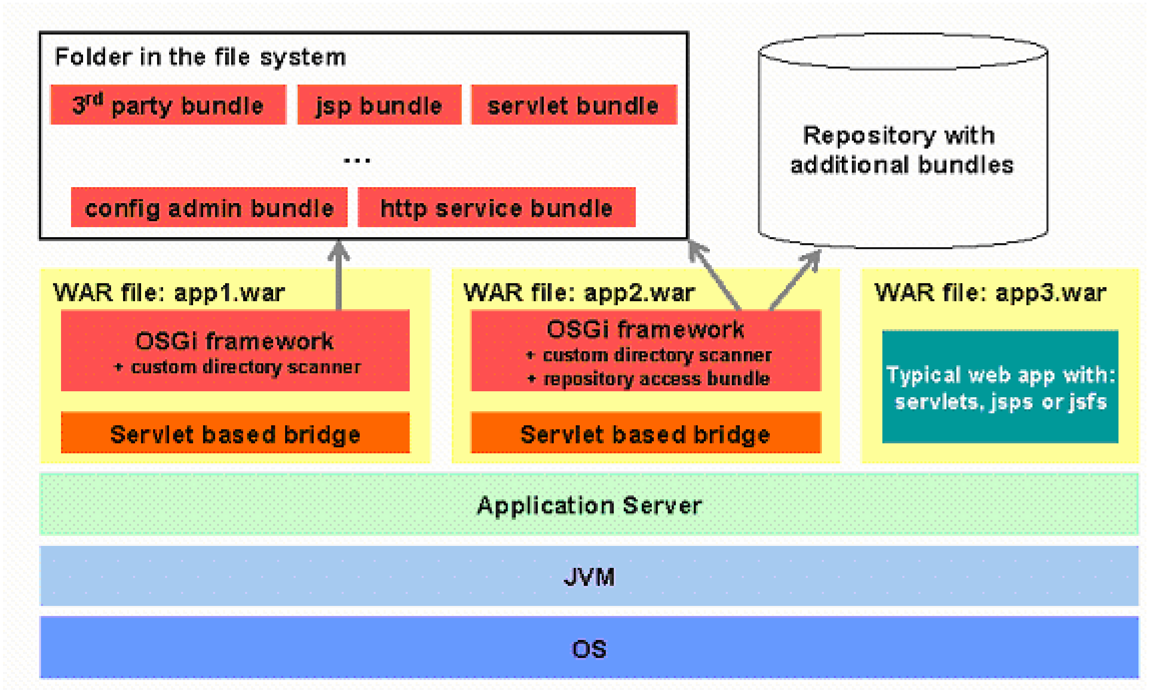

Figure 1 - OSGi embedded into an application server

As shown in Figure 1, a bridging servlet is used to establish the communication between the Application Server and the OSGi Container. In particular, the core application is developed according to the OSGi specification and uses the HTTP service defined in the specification to register its web resources. Such an application could be considered as self hosting of the HTTP service and doesn't require an external application server. Based on the isolation provided by OSGi, the application itself is independent from the HTTP service implementation, as long as the service fulfills the service contract defined in the specification. This is the key fact, the server side project is using. This project provides two important items. One is a bridging servlet and the other one is a custom implementation bundle of the HTTP service specified in OSGi. The bridge servlet, as depicted in Figure 1, is deployed on behalf of the OSGi application in the web container. Every request addressed to the application is now received by the servlet. With the first request, the servlet starts an OSGi container, which can either be part of the WAR file or be located elsewhere in the file system. Depending on the configuration, different bundles are installed during this process – either from a local or remote location, such as a repository. The HTTP service provided by the project is a proxy endpoint of the outer application service. Every request accepted by the servlet bridge bundle is forwarded to the proxy HTTP service and handled as if it is a self hosted service. With this approach it is possible to have multiple OSGi containers along with mainstream WAR based applications being deployed without conflicting with each other.

5. OSGi adaptation vs. OSGi migration

In this section we will compare and contrast two different practical ways of transforming UIMA into an OSGi enabled platform, namely OSGi migration and OSGi adaptation. Before making any comparison, we need to explain some of the core incompatibilities between OSGi and UIMA in order to understand the impact of each approach. These incompatibilities can be summarized as follows:

- The UIMA Framework uses the application class space to access all resources and Java classes of analytics and type systems. In the OSGi container, on the other hand, both the application and the framework are encapsulated in separate bundles and isolated from analytic component bundles and other resource modules. Thus, neither the application class space nor the framework class space includes Java classes and other resources that are necessary for deploying analytics in UIMA Framework.
- The UIMA Framework assumes that all Java classes and resources within a single analysis pipe are included in a single class space. In the OSGi container, each analytic component bundle has its own class space. Thus, the UIMA Framework is unable to run an aggregate component that integrates (by reference) several analytic component bundles.

Both the OSGi migration and adaptation approaches require that the UIMA Framework code, the code of analytics and other resources, as well as applications are encapsulated in OSGi bundles. Besides this similarity, the two approaches have significant differences as shown below.

A pure OSGi migration approach may be characterized by the following features:

- UIMA is transformed into service based architecture. Current API of UIMA Framework is transformed into a set of service interfaces, like UIMAFrameworkServices, UIMACasServices, etc.
- UIMA analytics are also transformed into services. All UIMA operations required to deploy and run an analytic are done inside the analytic component bundle class space.
- Integration of UIMA analytics into analysis pipes is done by using component service references instead of UIMA descriptor references.

While the OSGi migration approach can theoretically bring substantial benefits and remove certain UIMA limitations, like the requirement to use single class space for all resources within a single analysis pipe, in practice, this approach requires complete re-design and partial re-implementation of the framework API. It also implies significant changes in the application programming model and the analytic front-end code.

The OSGi adaptation approach, in contrast to migration, does not require any changes in the framework design or implementation. It also does not require any modifications in the code of UIMA analytics and UIMA descriptors. The main feature of the OSGi adaptation approach is the adaptation layer that allows applications to run UIMA analytics, encapsulated in OSGi bundles, without modifying the UIMA Framework code or changing the application programming model. Besides common benefits of OSGi enablement, this approach also simplifies transition to component based environment both for the UIMA Framework developers and for the community of UIMA analytic developers. The OSGi adaptation approach, however, does not completely eliminate the UIMA Framework limitations, such as the necessity to use single class space within a single analysis pipe. The application still is responsible for assigning a single class space to each analysis pipe. In response to these limitations, the OSGi adaptation layer provides the necessary class space established within the OSGi container, as described in the next chapter.

6. OSGi Service Adaptor for UIMA Framework

The OSGi Service Adaptor represents the adaptation layer that allows UIMA applications to run UIMA analytics and other resources encapsulated in OSGi bundles and deployed in the OSGi container. General architecture of the OSGi Service Adaptor layer is depicted in Figure 2.

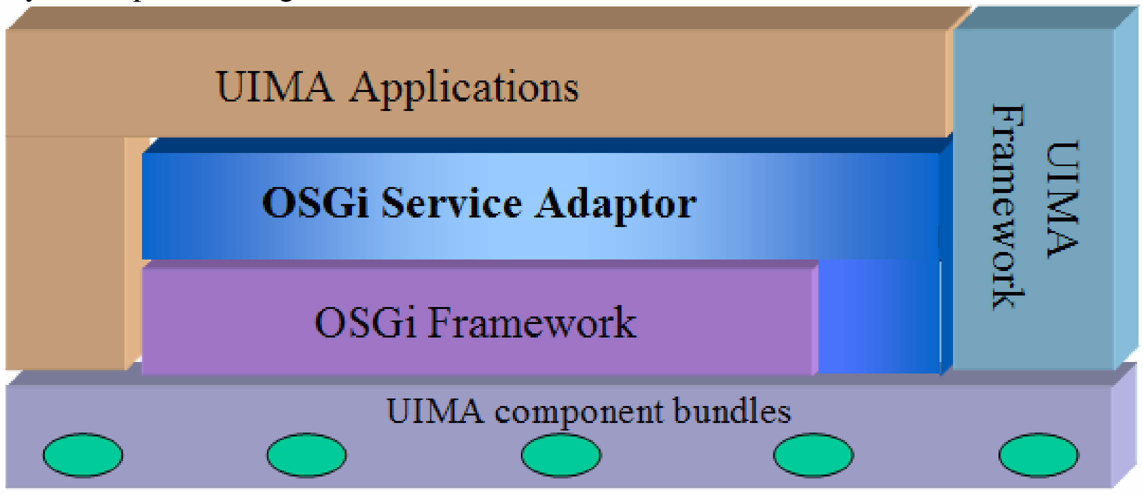

Figure 2. General architecture of the OSGi Service Adaptor layer.

The OSGi Service Adaptor is responsible for:

a) Allowing applications to manage OSGi bundles encapsulating UIMA analytics and other resources. The management capabilities include locating UIMA component bundles and loading them into the OSGi container to make them available for deployment by the UIMA Framework. This mechanism is based on the OSGi Framework API.
b) Providing applications with an easy access to shared resources of UIMA component bundles in order to deploy and use them in the UIMA Framework. The shared resources include component descriptors, component class space, metadata and others. This mechanism is based on the OSGi services registered by each UIMA analytic component bundle, as detailed further in this section.

Each UIMA analytic component bundle registers its OSGi services including the component reference service and one additional service for each of its public UIMA component descriptors. UIMA component bundles that register their OSGi services are called *active* component bundles, as opposed to *inactive* component bundles, like type systems or other resource bundles. Active component bundles can be independently discovered via OSGi Service Registry. Inactive component bundles need to be wired to other component bundles in order to share their resources. The OSGi Service Adaptor API provides dynamic access to registered UIMA component services through the OSGi Service Registry.

Logically, the OSGi Service Adaptor comprises the following functional elements:

A. The Adaptor Services Integrator – responsible for tracking all standard Adaptor services and notifying applications of the Adaptor service availability events. This module also provides convenient access to all standard Adaptor services.
B. The UIMA Configurator – responsible for managing UIMA component bundles and notifying applications of UIMA component life cycle events.
C. UIMA Component Services – responsible for providing access to shared resources of UIMA component bundles, loaded into the OSGi container.
D. UIMA Component Registry – responsible for looking-up UIMA component bundles loaded into the OSGi container.

The Adaptor Services Integrator monitors all standard Adaptor services by using the OSGi `ServiceTracker` capabilities and makes its own OSGi service available only when all other standard Adaptor services are available. The Integrator notifies subscribed applications of the Adaptor service availability events.

The UIMA Configurator combines several functions:

a) discovering new UIMA component bundle files in the designated directory;
b) installing UIMA component bundles in the OSGi container upon application request;
c) automatically activating UIMA components installed in the OSGi container;
d) monitoring UIMA component life cycle changes and notifying subscribed applications of UIMA component events.

The Configurator registers its own OSGi service that allows installing UIMA component bundles and subscribing for UIMA component events.

The UIMA Component Services define the API for accessing shared resources of UIMA component bundles, including:

- component bundle class loader – the bundle class loader established by the OSGi container;
- component UID – a unique identifier of a component bundle;
- component metadata, such as symbolic name, version and properties;
- component descriptor(s), a.k.a. specification(s); each component may have one or more public UIMA descriptors (specifications).

The Component Services also provide the API for *publishing* UIMA components, i.e. registering OSGi services that belong to UIMA component bundles. Each active UIMA component bundle registers two or more OSGi services at the start-up: (1) component reference service, providing access to component bundle resources, and (2) component specification service(s) for each of its public UIMA component descriptors. UIMA component publishing is done by the UIMA Component Publisher OSGi service, which is a part of the Component Services.

UIMA Component Registry defines the API for looking-up instances of UIMA component services in the OSGi Service Registry. The Component Registry allows to seek for both UIMA component references and specifications based on internal bundle ID, component UID and various metadata.

7. Creating and using UIMA component bundles

A typical UIMA analytic consists of its component descriptor, implementation code, JCas TS classes and, optionally, other resources, such as dictionary files. Each component descriptor, in its turn, may import other descriptors forming aggregate analytics. When an analytic is deployed in UIMA, the framework loads specified analytic implementation classes and JCas TS classes within the class space established by the client application. Usually, it means that the client application is responsible for adding all necessary Java classes and JARs to the default class path or configuring class path extension by using the integration framework API. The UIMA Framework allows applications to dynamically set a distinct class path extension or even use a distinct class loader for a certain analysis pipe, which may include one or more analytics.

In a modular system that consists of UIMA component bundles, the concept of application class space does not work any more. The UIMA libraries are encapsulated into one or more bundles and isolated from UIMA component bundles in the OSGi container as separate bundles with their own class space (class loaders). Applications themselves are also encapsulated into bundles and isolated from UIMA component bundles. Connections between the bundles show necessary dependencies, as illustrated in Figure 3:

- application bundle depends on the UIMA libraries bundle;
- primitive annotator bundles 1 and 2 depend on TS bundles encapsulating appropriate JCas TS classes, as well as on the UIMA libraries bundle;
- aggregate annotator bundle depends on its delegate bundles and the UIMA libraries.

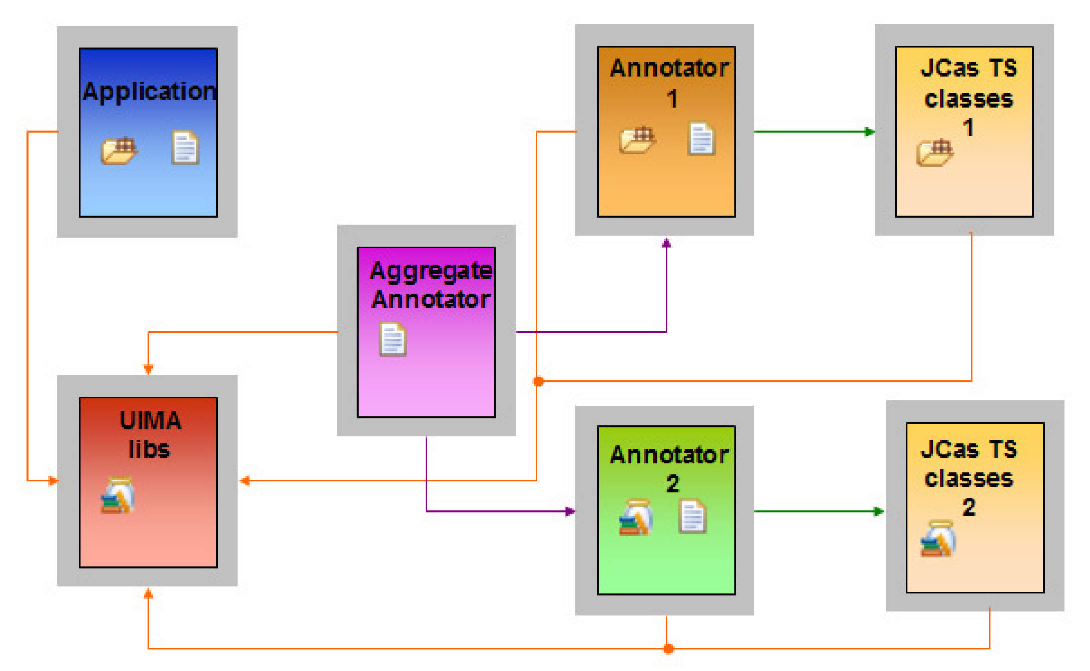

Figure 3. Dependencies between bundles in a modular UIMA application

The bundles with their dependencies in Figure 3 form the following class spaces within the OSGi container:

a) UIMA libraries bundle class space, which is limited to the bundle itself;
b) application bundle class space, including the application bundle itself and the UIMA libraries bundle;
c) annotator 1 bundle class space, including the annotator 1 bundle itself, as well as the JCas TS bundle 1 and the UIMA libraries bundle;
d) annotator 2 bundle class space, including the annotator 2 bundle itself, as well as the JCas TS bundle 2 and the UIMA libraries bundle;
e) aggregate annotator bundle class space, including the aggregate annotator bundle itself, both its delegate annotator bundles and the UIMA libraries bundle.

Note, that each class space may be associated with one or more bundle class loaders created by the OSGi container.

To deploy and run the aggregate annotator in the UIMA Framework, our application needs to locate the aggregate component bundle in the OSGi container and provide UIMA framework with access to the aggregate annotator class space. As we already mentioned in the section 6, this problem is resolved thru adding the adaptation layer that allows each UIMA analytic to register its services in the OSGi Service Registry. The OSGi Service Adaptor adds a thin service layer to each UIMA annotator bundle, as illustrated in Figure 4. UIMA annotator bundles register their references as OSGi services, and the application locates the aggregate component bundle service. Thru this service reference the application gains access to the bundle class loader and other resources of the aggregate component bundle and then deploys and runs the aggregate annotator in the UIMA Framework by setting the class loader for the aggregate analysis pipe.

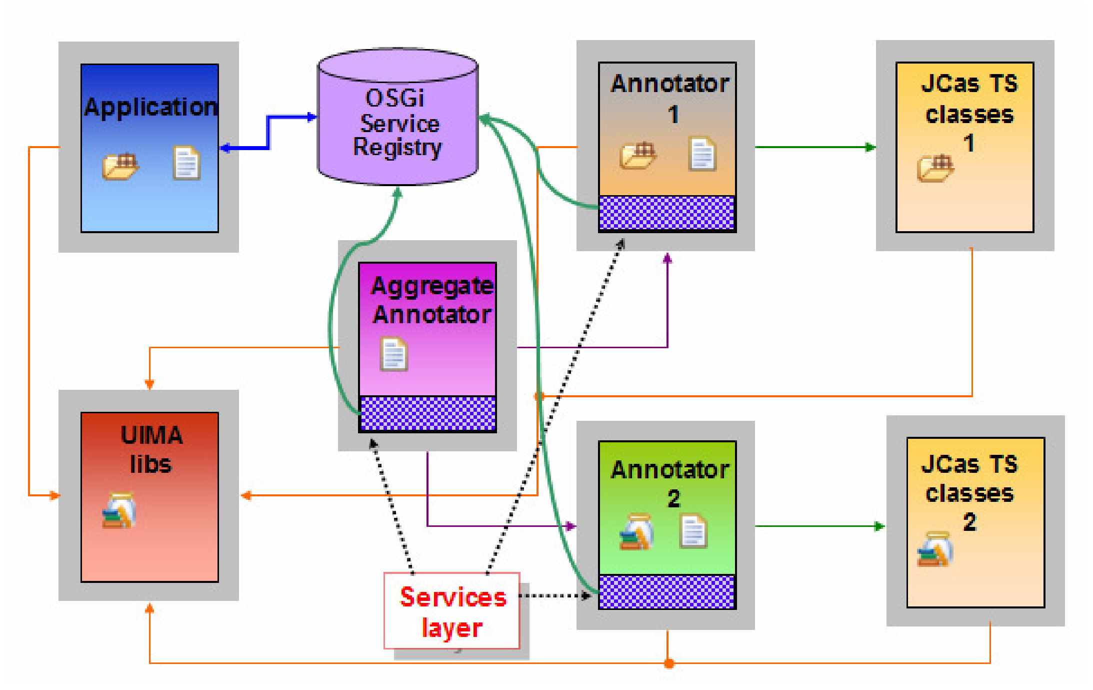

Figure 4. Registering UIMA components as OSGi services

All UIMA component bundles can be schematically split into 3 different categories:

I. **UIMA analytic component bundles**, encapsulating primitive or aggregate UIMA analytic components; each analytic component bundle should declare its UIMA component descriptor(s) in the OSGi manifest.

II. **UIMA type system bundles**, encapsulating JCas type system classes; each type system component bundle should declare its UIMA type system descriptor in the OSGi manifest.

III. **Inactive resource bundles**, encapsulating shareable linguistic and other resources, like dictionaries, etc. Inactive resource bundles may also declare their UIMA resource descriptors (if appropriate) in the OSGi manifest.

As we mentioned earlier, only UIMA analytic component bundles are *active*, i.e. register their OSGi services. Other categories of UIMA component bundles are connected to appropriate active bundle(s) through explicit OSGi level dependencies, a.k.a. OSGi *wiring*. Each UIMA component bundle, encapsulating primitive analytic components, should be wired to at least one type system bundle to resolve its JCas type system classes. The reason why the JCas type system classes are not directly encapsulated in analytic component bundles is explained in the following passage: the OSGi container uses different class loaders for loading classes from different bundles; therefore, the same JCas type system classes, encapsulated in different bundles, are loaded with different class loaders. However, the UIMA Framework requires that within a single analysis pipe the same JCas type system class is always loaded with the same class loader.

Further in this section we will describe the process of creating UIMA analytic component bundles in Eclipse PDE [12]. We will briefly describe the steps required to create a UIMA analytic component bundle from existing code. In general, the following points are important when creating UIMA component bundles:

a. UIMA component bundles should be created as standard OSGi bundles to ensure compatibility with non-Eclipse OSGi Framework implementations.
b. Different UIMA component bundles are not allowed to share the same symbolic name to avoid name collisions when integrated into aggregates.
c. Different primitive analytics need to use different namespace (package names) for their Java code to avoid split package cases when integrated into an aggregate class space.

Note that developers of UIMA component bundles should adhere to OSGi best practices (see [25]) to make sure their products are compatible with other UIMA component bundles and OSGi products, in general.

Step 1: Creating a type system bundle.

As we already mentioned, TS bundle encapsulates UIMA TS descriptor and JCas TS classes. Primitive analytics usually require some TS descriptions, so that appropriate TS bundle(s) should be ready prior to creating an analytic component bundle. The process of TS bundle creation in the Eclipse PDE includes (a) creating a new OSGi bundle project without Activator class, (b) adding bundle wiring to the UIMA runtime bundle, (c) copying the TS descriptor file into an unique Java package, (d) generating JCas TS classes, (e) exporting Java packages containing JCas classes as well as the TS descriptor and (f) specifying the location of the TS descriptor by using the `UIMA-Descriptor` manifest header.

Step 2: Creating a primitive analytic component bundle.

Typically, creating analytic component bundles from existing code does not require any code modifications. The process of creating a primitive analytic component bundle in the Eclipse PDE includes (a) creating a new OSGi bundle project with an Activator class, (b) adding bundle wiring to the UIMA runtime bundle, the OSGi Service Adaptor bundle and appropriate TS bundle, (c) replacing the generated Activator class with the standard UIMA component Activator template, (d) copying analytic implementation code into unique Java packages, (e) copying analytic descriptor files into a unique Java package, (f) exporting Java packages containing the implementation code as well as the analytic descriptor and (g) specifying the location of the analytic descriptor by using the `UIMA-Descriptor` manifest header.
Note 1: analytic component bundles need to re-export dependencies on TS bundles in order to make the TS packages visible to aggregates.
Note 2: analytic descriptors can import TS descriptors from wired TS bundles only ‘by name’, based on the analytic’s bundle class space.

Step 3: Creating an aggregate analytic component bundle.

Aggregate analytics do not contain their own code but import descriptors of their delegates. The process of creating aggregate analytic component bundle in the Eclipse PDE includes (a) creating a new OSGi bundle project with an Activator class, (b) adding bundle wiring to the UIMA runtime bundle, the OSGi Service Adaptor bundle and appropriate delegate bundles, (c) replacing the generated Activator class with the

standard UIMA component Activator template, (d) copying aggregate descriptor files into a unique Java package and importing delegate descriptors 'by name' from appropriate analytic component bundles, (e) exporting the Java package containing the aggregate descriptor and (g) specifying the location of the aggregate descriptor by using the `UIMA-Descriptor` manifest header.
Note that an aggregate component bundle needs to re-export its dependencies on delegate bundles in order to make the analytic packages visible to upper level aggregates.

The code of the standard UIMA component Activator template does not depend on the UIMA component itself. The `Activator.start()` method, which is invoked by the OSGi container after the component bundle is installed, uses the OSGi Service Adaptor API to *publish* the UIMA component bundle, in other words, to register the reference service of the component bundle and make the encapsulated analytic available for UIMA deployment. For more information on creating UIMA analytic component bundles in Eclipse PDE see [26].

8. Creating applications for OSGi enabled UIMA

Application development for OSGi enabled UIMA utilizes the following programming methods:

- using asynchronous event driven control flow;
- using the OSGi Service Adaptor API to manage UIMA component bundles and get access to available components in the OSGi container;
- using the UIMA Java Framework API to deploy and run UIMA components in the UIMA Framework.

This section provides comprehensive analysis of the most important features of a sample OSGi application, named `AETester`, which can discover, install and run UIMA analytic bundles.

The first module of each active OSGi application is its `Activator` class. The Activator for the `AETester` application extends the standard OSGi bundle activator class. First of all, the `Activator` class additionally implements the `IUIMAServiceListener` interface, defined by the OSGi Service Adaptor for applications that need to receive UIMA service events. The `Activator` class instance, created by the OSGi Framework during the application start-up, registers itself as the UIMA services listener in the `start()` method:

```
UIMAServices.addServiceListener( this, context );
```

Once the UIMA services listener is added, the OSGi Service Adaptor notifies it of all UIMA service events by calling the listener's `serviceChanged()` method. In this method the `Activator` class performs appropriate actions in response to certain UIMA service events – initializes or terminates the main `AETester` application module, implemented in the `AETesterApp` class. This class implements the

IUIMAComponentListener interface, defined by the OSGi Service Adaptor API for applications that need to receive UIMA component events. The AETesterApp class instance, created by the Activator class, registers itself as the UIMA component listener in the initialize() method:

```
UIMAServices.getConfigurator().
                    addComponentListener( this, context );
```

After the UIMA component listener is added, the OSGi Service Adaptor notifies it of all UIMA component events by calling the listener's componentChanged() method. In this method the AETesterApp class performs the following actions in response to certain UIMA component events:

- skips all events other then UIMAComponentEvent.REGISTERED (a new component is available for immediate use in the OSGi container) or UIMAComponentEvent.DISCOVERED (a new component bundle file has been found in the designated folder);
- installs into the OSGi container all new component bundle files, which have been discovered in the designated folder;
- once the new AE component becomes available in the OSGi container, the application runs it in a separate thread with sample input documents.

To deploy and run an available UIMA AE component the application produces the AE instance based on the IUIMAComponent reference, received with the UIMA component event. Producing AE instance involves both the OSGi Service Adaptor and the UIMA Framework calls as outlined below. The application gets the AE XML component descriptor URL from the given IUIMAComponent reference (uimaComponent):

```
URL aeDescriptorUrl = uimaComponent.
       getDefaultComponentSpec().getComponentDescriptor();
```

Note that in this case we use the default component descriptor, but it's possible to retrieve more that one descriptor, if specified in the component bundle manifest. Next, the application gets the bundle class loader of the given component:

```
ClassLoader bundleClassLoader =
             uimaComponent.getBundleClassLoader();
```

Note that the bundle class loader is defined by the OSGi container and represents the class space, which includes both the component bundle itself and the whole tree of required or wired component bundles. The next few steps are the same as for any typical UIMA application:

```
// get UIMA XMLParser
XMLParser xmlParser = UIMAFramework.getXMLParser();
// build XMLInputSource object for AE descriptor URL
XMLInputSource xmlSource =
           new XMLInputSource( aeDescriptorUrl );
// create AE resource specifier
ResourceSpecifier aeSpecifier =
           xmlParser.parseResourceSpecifier( xmlSource );
```

The next steps are necessary for any application using OSGi enabled UIMA – the component bundle class loader is used to set the class space of the UIMA analysis pipe:

```
// create new UIMA resource manager
ResourceManager resourceManager =
                UIMAFramework.newDefaultResourceManager();
// customize ResourceManager using bundle class loader
resourceManager.setExtensionClassPath(
                     bundleClassLoader, "", true );
// produce AE using customized resource manager
AnalysisEngine ae = UIMAFramework.produceAnalysisEngine(
                     aeSpecifier, resourceManager, null );
```

After the AE is deployed in the UIMA Framework (the AE instance is created), the applications runs the AE using the regular UIMA Java Framework API. For more information on application development for OSGi enabled UIMA see [26].

9. Related work

Although OSGi exists since 1999, the real popularity boost came with the release 3, especially after the Eclipse Project [14] decided to replace the core code base with their own OSGi implementation in Eclipse 3.1. Today, the derived Equinox Project [15] together with the Plug-in Development Environment [12] in Eclipse represent the de-facto standard for creating OSGi based applications. With the release of version 4 of the specification, the componentization capabilities draw more and more attention, as already mentioned in [4]. Currently, we see an ongoing growing number of OSGi adopters among several IT domains. The following paragraphs will point out some of the ongoing efforts done in several domains.

Application Server vendors

The largest movement towards OSGi among one industry domain can be found in Java EE. Almost any application server vendor is internally using OSGi or, at least, providing an adaptation approach:

- Object Web JOnAS 5 (in development) [16][17]
- IBM WebSphere Application Server V6.1 [18]
- BEA microService Architecture [19]
- Apache Geronimo (prototyped, but not official) [21]
- JBoss Application Server (discussed) [20]

Java Frameworks

The Spring framework developed by interface21[22] is probably the most popular enterprise Java framework today, with numerous citations in magazines and conferences. The use of “Plain Old Java Objects” (POJOs) instead of cluttering up your application with vendor dependant APIs, but just injecting dependencies on components/ frameworks/ mechanisms in a descriptive way can be considered as the key of its success. Despite of the valuable improvements, however, due to the lack of componentization in Java, Spring adopted the OSGi, as well. In Version 2.1, OSGi enablement is an important new feature, which leverages packaging, distribution and discovery of new components in the application container. Based on Java Beans, Spring is able to use pure OSGi mechanisms to manipulate the class path of the components by obtaining the ClassLoader through OSGi APIs, which is not possible for resource centric approaches like UIMA. Additionally Spring offers dependency injection build upon Aspects. This technique, no matter how valuable, is not a part of the OSGi specification and due to its specific technical nature can not be implemented based on the specification itself. There is an ongoing work on creating an aspect enabled OSGi container on the base of Equinox[15] together with the “Aspects and OSGi” Equinox Incubator project. The current experiences acquired by this expert group, lead to the latest RFP 0076, which is about to describe an enhanced component model in OSGi that addresses many of the found limitations.

Apache Harmony[24] is an open implementation of the Java Standard Platform Edition. Unlike other projects, here it is not possible to run OSGi at all, because the project defines the environment, in which OSGi cannot be executed. Instead, the developers of Harmony decided to use the rich meta-data provided by bundles to define dependencies of class libraries among each other. With this information a check for validity can be executed to ensure that all required resources are available and implementations can be easily exchanged.

10. Conclusion

This report discusses the concepts and technical approaches for OSGi enablement of the UIMA Framework. The adaptation approach, as opposed to migration, is introduced as the low cost solution that extends an existing framework with a modularization facade. The OSGi adaptation solution is not limited to a particular software system, such as the UIMA Framework, it can be applicable to certain classes of Java systems, especially integration frameworks.

OSGi enablement transforms the UIMA Framework into an industrial standard component based development and integration platform fully interoperable with other OSGi products. OSGi enablement also improves asset management, adds dynamism, and facilitates embedding and process automation. While extending the capabilities of UIMA, the enablement preserves the current UIMA application programming model and makes the underlying OSGi technology almost transparent to UIMA application developers.

We introduced the OSGi Service Adaptor – a thin layer between the framework and the OSGi container that allows running analytics encapsulated in OSGi bundles. The adaptor utilizes OSGi services to allow the framework dynamically obtain and integrate runtime resources of analytics without specifying static dependencies on them.

Although this approach removes most of the burden from developers, creating modular analytics and applications is not totally effortless. The 1st, and the most important, step is to establish a common understanding of the definition of components in a particular domain. In case of UIMA, logical components are already defined in a declarative way. To create appropriate physical components, UIMA developers just need to encapsulate all required resources into OSGi bundles. Developing such bundles requires following OSGi best practices (see [25]), as a superset of Java best practices. This requirement is mandatory, because OSGi, as a full featured component based framework, enforces the use of these best practices, which results in cleaner and more robust software.

References

[1] OSGi Technology, http://www.osgi.org/osgi_technology/?section=2
[2] JSR 277: Java Module System, http://jcp.org/en/jsr/detail?id=291
[3] JSR 291: Dynamic Component Support for Java SE, http://www.jcp.org/en/jsr/detail?id=291
[4] OSGi Projects @ JavaOne 2006, http://www2.osgi.org/div/massive.ppt
[5] About the OSGi Service Platform, http://www.osgi.org/documents/collateral/TechnicalWhitePaper2005osgi-sp-overview.pdf
[6] UIMA @ IBM Research, http://www.research.ibm.com/UIMA/
[7] UIMA @ Wikipedia, http://en.wikipedia.org/wiki/Uima
[8] Ferrucci, D., Lally, A., "UIMA: an architectural approach to unstructured information processing in the corporate research environment", Natural Language Engineering, Volume 10 , Issue 3-4 (Sep 2004), pp. 327-348, Cambridge University Press, New York, NY, USA, 2004, ISBN: 1351-3249
[9] Goetz, T., Suhre, O., "Design and Implementation of the UIMA Common Analysis System", IBM Systems Journal, v.43, # 3, 2004, pp.476-489, http://www.research.ibm.com/journal/sj/433/gotz.html
[10] Ferrucci, D., Lally, A., "Building an example application with the Unstructured Information Management Architecture", IBM Systems Journal, v.43, # 3, 2004, pp.455-475, http://www.research.ibm.com/journal/sj/433/ferrucci.html
[11] Schor, M., "An Effective, Java-Friendly Interface to the CAS", Research Report RC23176, IBM T.J. Watson Research Center, Yorktown Heights, NY, 2004, http://domino.research.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b360066f0d4/d55875841121943e85256e78004bd826?OpenDocument
[12] Eclipse Plug-in Development Environment (PDE), http://www.eclipse.org/pde/
[13] Eclipse Server Side project, http://www.eclipse.org/equinox/server/

[14] Eclipse Foundation,
http://www.eclipse.org
[15] Eclipse Equinox,
http://www.eclipse.org/equinox/
[16] JonAS
https://wiki.objectweb.org/jonas/Wiki.jsp?page=JOnASonDemand
[17] JOnAS move to OSGi
http://objectwebcon06.objectweb.org/xwiki/bin/download/Main/DetailedSession/M-Desertot-T-Abdellatif.pdf
[18] IBM WebSphere Application Server V6.1
http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.was_v6/was/6.1/Architecture/WASv61_Componentization/player.html
[19] BEA microService Architecture
http://www.bea.com/framework.jsp?CNT=msa.jsp&FP=/content
[20] JBoss Application Server
http://www.jboss.org/index.html?module=bb&op=viewforum&f=256
[21] Apache Geronimo Application Server
http://www.mail-archive.com/dev@geronimo.apache.org/msg10923.html
[22] Interface21
http://www.interface21.com/
[23] Aspects and OSGi
http://www.eclipse.org/equinox/incubator/aspects/
[24] Apache Harmony - Open Source Java SE
http://harmony.apache.org/
[25] Jahn, M. "OSGi Applied: Lessons Learned und Best Practices", to be published in Java Magazin, Frankfurt, Germany, August 2007
[26] Dynamic Module Enabler for Unstructured Information Management Architecture Component Analytics, IBM alphaWorks, June 2007,
http://www.alphaworks.ibm.com/tech/dmeuima

Appendix

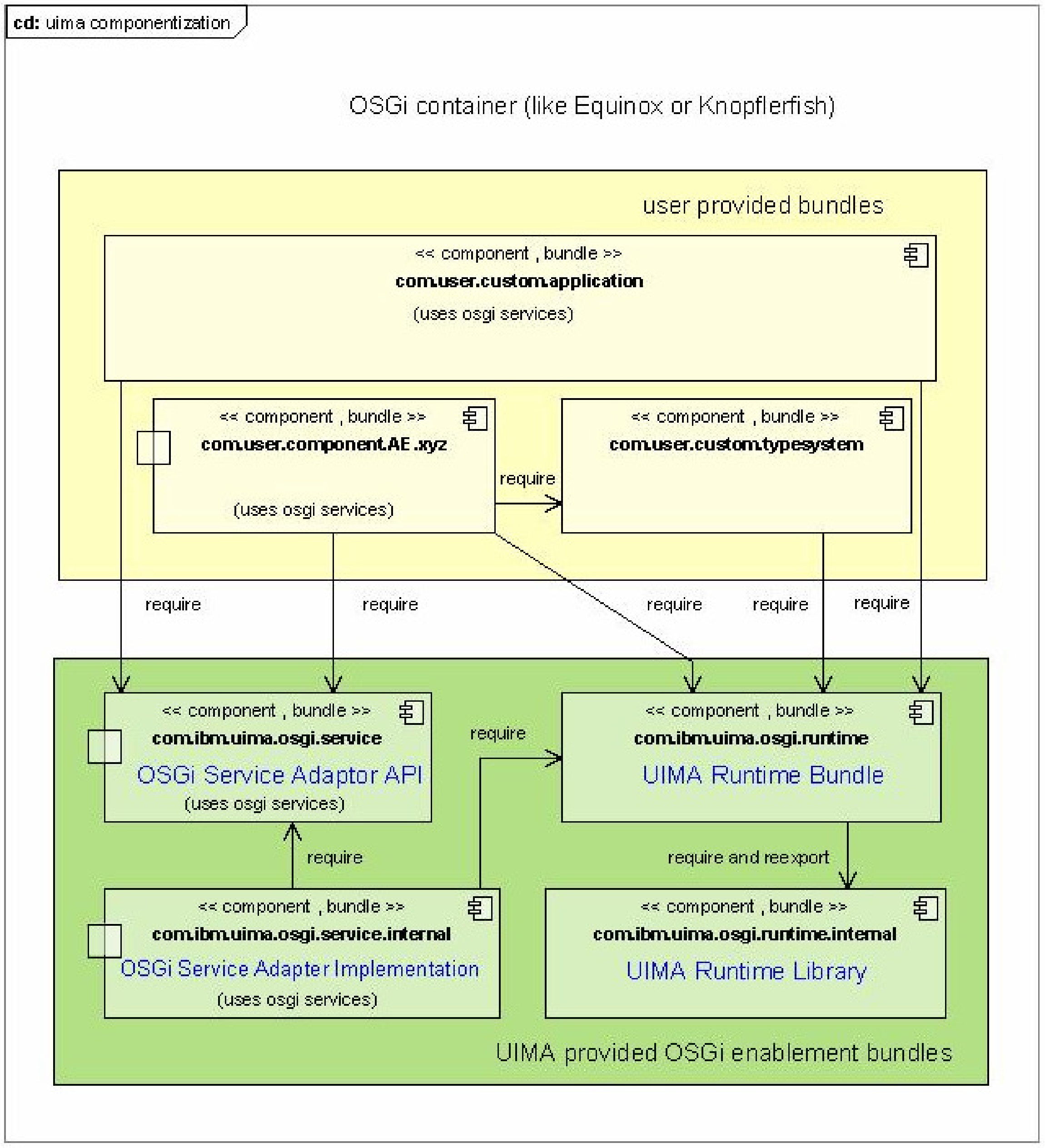

Figure 5. Implementation diagram showing dependencies between application bundle, component bundle, type system bundle, UIMA runtime bundles and Service Adaptor bundles.