TOWARD A PIVOT
SERVICE ORIENTED COMPONENT MODEL
UPON OSGi

Master report presented by

Bassem Debbabi

September 15, 2008

realized at the
Laboratoire d’Informatique de Grenoble,
LIG (UMR CNRS 5217), ADELE team
(http://www-adele.imag.fr/)
under the direction of
Clément Escoffier and Didier Donsez
Table of contents

Table of contents............................................................................................................. 3
List of figures.................................................................................................................. 4
Liste of tables................................................................................................................. 5
1 Introduction.................................................................................................................... 6
  1.1 The Adele team....................................................................................................... 6
  1.2 Context................................................................................................................... 7
  1.3 Paper organization................................................................................................. 7
2 Background..................................................................................................................... 7
  2.1 Component-Based Software Engineering............................................................... 7
  2.2 Service-Oriented Computing and Software Oriented Architecture..................... 9
  2.3 The OSGi Service Platform.................................................................................. 10
3 Service-Oriented Component Model............................................................................. 12
  3.1 Principles............................................................................................................... 12
  3.2 Existing Service-Oriented Component Models.................................................... 13
3 Toward a common service oriented component model................................................. 17
  3.1 Objectives............................................................................................................. 17
  3.2 Requirements......................................................................................................... 18
  3.2 Approach............................................................................................................... 18
4 Realization.................................................................................................................... 19
  4.1 The migration tool “Spring DM 2 iPOJO”............................................................. 19
5 Conclusion..................................................................................................................... 23
References....................................................................................................................... 23
List of figures

Fig. 1. Service-Oriented Interaction pattern................................................................. 9
Fig. 2. OSGi Platform ................................................................................................. 11
Fig. 3. Example of component descriptors in DS. ............................................... 14
Fig. 5. Example of metadata in iPOJO................................................................. 16
Fig. 6. An iPOJO instance and its container ....................................................... 17
Fig. 7. Spring-DM to iPOJO transformer.......................................................... 20
Fig. 8. Spring-DM configuration file ................................................................. 21
Fig. 9. The generated iPOJO metadata............................................................ 21
Fig. 10. A part of SpringDM to iPOJO XSLT Stylesheet................................. 22
Liste of tables

Table 1. Pros and Cons of CBSE .............................................................. 8
Table 2. Pros and Cons of COC ................................................................. 10
Table 3 Pros and Cons of OSGi ................................................................. 12
Toward a pivot service oriented component model upon OSGi

Bassem Debbabi

Supervisor: Didier Donsez and Clement Escoffier

LIG, ADELE, 220 Rue de la Chimie, BP53
38041 Grenoble Cedex 9, FRANCE
bassem.debbabi@imag.fr

Abstract. The dynamic service platforms become a more and more used in the industry. These platforms allow executing services which appear and disappear during the execution of an application. Two popular representatives are Web Services for the distributed platforms and OSGi for the centralized platforms. Currently, OSGi becomes the de facto standard for the development and the deployment of modular applications and flexible middlewares. Having focused rather constrained environments (embedded) on its first versions, the latest, the 4.1 release covers a large spectrum of application areas such as management of complex real estate, industrial supervision, automobiles, mobile phones, JavaEE application servers, Plugin-based applications such as Eclipse. However, applications' development is difficult and it requires the usage of Component models supporting the dynamic reconfiguration of applications. A several Service Oriented Component Models that introduces the services concept to component models were proposed. Our aim by this work is to provide a common Service Oriented Component Model by merging them all in one pivot model. We propose a set of transformations allowing automatic migrations from different models to iPOJO. Indeed, iPOJO is a perfect challenger to be our pivot model as it provides all the features required to create highly dynamic service based application.

Keywords: Component-Based Software Engineering, Service Oriented Computing, Service-Oriented Component Model, OSGi, iPOJO

1 Introduction

1.1 The Adele team

The Adele team has always emphasized issues raised by software evolution, more specifically for very large industrial software. The previous work of the team was on software configuration management, reverse engineering and software processes. Evolution is now seen in a more general way, considering also evolution during
execution, not only evolution during development or maintenance. Dynamic evolution addresses issues related to dynamic update, dynamic deployment and adaptation to the evolution of software, hardware or user needs and characteristics.

1.2 Context

Component-Based Software Engineering (CBSE) and Service-Oriented Computing (SOC) are among today's most prominent software architecture approaches. But these two approaches have pro and cons, so since 2002, a third trends appears to merge concepts from these two approaches in one: the Service Oriented component models. A several SOCMs were proposed. Each of them has its features and tools. But all relies of the same concepts and techniques. This paper explores concepts and solution to a potential merge of these component models on one central SOCM.

1.3 Paper organization

This paper is organized as follow. The section 2 gives an overview of Component-Based Software Engineering and Service Oriented Computing approaches, followed by a short description of the OSGi framework. Section 3 presents the Service-Oriented Component Model, with some examples of the most popular models. Section 4, describes our approach to merge between the different models. And the last section presents our realization.

2 Background

Almost all evolutions in recent years intend to develop a type of architecture that allows loose coupling and high reusability and integrity of its components[13]. One of the most famous software architecture approaches are Component-Based Software Engineering (CBSE) and Service-Oriented Architecture (SOA).

2.1 Component-Based Software Engineering

Component Based Software Engineering (CBSE) promotes the construction of applications as composition of reusable building blocks called components. One of the widely referenced definition of the term component is given by Szyperski[8]:

“A software component is a binary unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”
The goal of component-based software engineering is to increase productivity, quality, and time-to-market in software development by building software systems from standard components rather than reinventing the wheel each time.

A component model is probably used for the developing and executing of components. This model defines a framework, which defines structural requirements for connection and composition options as well as behavior-oriented requirements for collaboration options to the components. Beyond that a component model provides an infrastructure which implements frequently used mechanism like persistence, message-exchange, and security.

To support composition, a component must provide information that describes the external structure of its instances, such as provided and required functional interfaces, and configurable properties. The description of the external structure is used by the application assembler to connect and customize the various component instances into a structural composition. Composition is performed using either a standard programming language or a specialized one. Hierarchical composition is achieved when the external structure of a component is itself implemented by a composition.

Components are delivered and deployed independently as binary code along with their required resources, such as images and libraries. When installed, a component may require deployment dependencies be fulfilled before it can be instantiated. Finally, component instances require an execution environment that provides run-time support, normally through what is known as container. Run-time support includes life-cycle management and handling of non-functional characteristics such as remote communication, security, persistence and transactions.

Different manufacturers offer platforms supporting Component-based software engineering like COM, DCOM, Enterprise JavaBeans, and .NET.

<table>
<thead>
<tr>
<th>Table 1. Pros and Cons of CBSE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pro</strong></td>
</tr>
<tr>
<td>The main benefits of the CBSE approach are the re-usability, the composition allowing building applications by assembly of components.</td>
</tr>
</tbody>
</table>
2.2 Service-Oriented Computing and Software Oriented Architecture

Service-oriented computing (SOC) [1] is a paradigm that utilizes services as fundamental elements for application design. The central objective of the service-oriented approach is to reduce dependencies among “software islands,” where an island is typically some remote piece of functionality accessed by clients. By reducing such dependencies, each element can evolve separately, so the resulting application is more flexible than monolithic applications.

SOC is based on three actors (see Figure 1):

- A service provider offers a service.
- A service requester uses a service.
- A service registry contains references to available services.

Fig. 1. Service-Oriented Interaction pattern

Another central concept to SOC is the service description (or service specification), which is a description of the functionality provided by a service. Service providers implement a specific service specification and service consumers know how to interact with services implementing the specifications they require. Among these three actors are three kinds of interactions: service publication between the provider and the broker to offer services for use, service discovery between the consumer and broker to find desired services, and service invocation between the consumer and provider to actually use the service.

To design complex service-oriented applications, it is necessary to compose services to provide higher-level services, which means that providers may require other services to provide their own service. Current approaches to SOC, specifically
in web services, offer process-oriented solutions to this issue; however, service-oriented applications can be difficult to develop since the mechanisms tend to require a lot of developer effort. Indeed, developers need to manage service publication/revocation, required service discovery, and run-time tracking of services.

Table 2. Pros and Cons of COC

<table>
<thead>
<tr>
<th>Pro</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Loose coupling:</em> as the only information shared between suppliers and consumers is the specification of service, the coupling between these two entities is low.</td>
<td><em>Lack of structural composition:</em> there is no common method to create application by assembling services together structurally. This is mainly due to the lack of expressed dependencies.</td>
</tr>
<tr>
<td><em>Late binding:</em> the link between suppliers and consumers takes place only when a supplier is found and if the consumer requests it.</td>
<td><em>Difficult to implement:</em> service based application must manage both the internal business logic and the service interactions (publication, discovery, binding)</td>
</tr>
<tr>
<td><em>Heterogeneity:</em> thanks to the provided loose-coupling, a consumer does not know the details of implementation of the service provider or its precise location.</td>
<td></td>
</tr>
</tbody>
</table>
− **Service Layer**: bundles are built around a set of services available from a shared services registry. Each bundle can register his services in this registry, so that other bundle can use it. We speak here about a publish/discover/bind cycle.
− **Security Layer**: It is an optional layer based on the security architecture of Java. This layer provides a reliable infrastructure for deploying and managing applications that must be carried out in a controlled environment.

Based on the framework previously presented, many OSGi Services has been specified. These are services relevant and widely used. An example is the HTTP service.

OSGi dispose all Service Oriented Computing relative notions [11]. The Framework provides a Service Registry where services can be registered and served by others. Those services are referenced by a Service Description (represented by a Java Interface), and a set of properties (version, vendor, language, etc) which refine service finding using LDAP filters. The Framework has also, an event system, which allows components executing in OSGi platform to be notified to manage their life cycle dynamically.

---

**Fig. 2. OSGi Platform**
Table 3 Pros and Cons of OSGi

<table>
<thead>
<tr>
<th>Pro</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>– Provide a very effective mechanism to manage the deployment of service delivery units.</td>
<td>– Dependencies between services are not managed by the OSGi framework and must be manage manually by the developers</td>
</tr>
<tr>
<td>– Dynamic deployment without interrupting the platform.</td>
<td>– Mix between the application code and dynamism management code</td>
</tr>
<tr>
<td>– Management of dependencies between bundles.</td>
<td></td>
</tr>
</tbody>
</table>

OSGi technology provides an interesting platform for creating dynamically extensible applications; however, dealing with dynamism is not simple. Developers should concentrate on application logic, not low-level OSGi mechanisms. This is way Service-Oriented Component Models are introduced on the top of OSGi.

3 Service-Oriented Component Model

The OSGi platform provides a good mechanism to build modular and dynamic applications. However it lacks a rich component model for declaring components within a bundle and for instantiating, configuring, assembling and decorating such components when a bundle is started.

A Service-Oriented Component Model (SOCM) introduces concepts from service orientation into a component model to provide a viable solution to developers to build dynamic applications upon OSGi.

3.1 Principles

SOCM principles were introduced in [4]. Those principles are:

− A service is characterized by a service specification, which describes some combination of a service's syntax, behavior, and semantics as well as dependencies on other services.
− Components implement service specifications, which may exhibit implementation-specific dependencies on services.
− The service-oriented interaction pattern is used to resolve service dependencies at run time.
− Compositions are described in terms of service specifications.
− Service specifications provide a basis for substitutability.

When using a service-oriented component approach to build an application, the application is decomposed into a collection of interacting services. The semantics and
behavior of these services are carefully described independently of any implementation. By doing so, it is possible to develop the constituent services independently of each other as well as to have variant implementations that are easily interchangeable. Variant implementations can be used, for example, to support different platforms or different non-functional requirements.

Once the application’s services have been defined, it is possible to define a mapping to a set of components for implementation. A component may implement zero or more services and there is no requirement on how the mapping from service specification to component implementation occurs. For example, if certain services are related, then it might make sense from a cohesion or performance point of view to implement them using a single component; however, it is not required. Additionally, whether a service represents local or remote functionality is largely irrelevant. If a service does represent remote functionality, then the resulting component implementation is merely a proxy stub for the remote service, which can be treated like any other component implementing a local service.

In traditional component-oriented composition, the component selection process for a composition occurs at design time. The selection process for a service-oriented composition occurs at run time as component instances are created inside the execution environment. The execution of the application starts the moment the main component instance’s dependencies are satisfied. The application composition is thus an abstract descriptor that could be used, for example, by a deployment system to deploy components that satisfy the service specifications required by the composition. The resulting application configuration depends on the specific set of deployed components, which may vary per platform or even dynamically at run time.

### 3.2 Existing Service-Oriented Component Models

#### 3.2.1 Declarative Services

Declarative Services (or “DS” also called Service Component Runtime) specification is part of OSGi specification. It is a component service model inspired from the Service Binder [5], introducing automated service dependency management, and then developers focus only on the job of writing their services. The wiring between services was handled declaratively, and the declarations were made in XML.

The Declarative Services introduces the concept of service component to the OSGi framework. A service component is similar to the concept of a logical bundle but the difference is that multiple service components can be deployed inside a single physical bundle.

A service component declares a set of provided service interfaces, and a set of required service interfaces. During execution, an instance of a service component implements the provided services and is connected to other instances to create an
application. Service properties identify the instance and are used when its services are published in the service registry.

Service components are described inside an XML file called a component descriptor. Inside this descriptor, components are declared as shown in figure 3. A bundle can contain several components.

```
Server component

<component name="serverComponent">
   <implementation class="org.example.ServerImpl"/>
   <service>
      <provide interface="org.example.Server"/>
   </service>
</component>

Client component

<component name="clientComponent">
   <implementation class="org.example.ClientImpl"/>
   <reference name="SERVER" interface="org.example.Server" bind="setServer" unbind="unsetServer" cardinality="0..1" policy="dynamic"/>
</component>
```

Fig. 3. Example of component descriptors in DS.

During execution, every component instance is managed independently by an instance manager that takes in charge service registration and service dependency management activities based on the information present in the component descriptor.

### 3.2.3 Spring Dynamic Modules

The Spring Framework is an open source application framework which has become popular in the Java community as an alternative, replacement, or even addition to the Enterprise JavaBeans (J2EE) model.

Spring Dynamic Modules (or Spring-DM) focuses on integrating Spring Framework, non-invasive programming model and concepts with the dynamics and modularity of OSGi platform. It allows transparent exporting and importing of OSGi services, life cycle management and control.

The combination of Spring Dynamic Modules and the OSGi platform provides:
- Better separation of application logic into modules, with runtime enforcement of module boundaries
- The ability to deploy multiple versions of a module (or library) concurrently
- The ability to dynamically install, update and uninstall modules in a running system.
- Use of the Spring Framework to instantiate, configure, assemble, and decorate components within and across modules.
- A simple and familiar programming model for enterprise developers to exploit the features of the OSGi platform

Spring-DM can export Spring beans as OSGi Services, and define references to services obtained via the Service Registry. These actions are defined in XML configuration files (see Figure 4).

In Spring DM, there are two kinds of XML configuration files, one containing the definition of beans for each bundle, and the second containing OSGi definitions (services, references...). This separation is to have an easy integration testing outside of an OSGi environment, but there is no requirement to do this split.

Spring-DM manages life cycle of exported services, that is, when service dependencies of a bean are not satisfied, exported services of that bean are not published.

```
Server component

<beans>
    <osgi:service interface="org.example.Server">
        <bean class="org.example.ServerImpl">
        </bean>
    </osgi:service>
</beans>

Client component

<beans>
    <reference id="server" interface="org.example.Server"/>
    <bean id="ClientBean" class="org.example.ClientImpl">
        <property name="m_server" ref="server"/>
    </bean>
</beans>
```

Fig. 4. Example of configuration data in Spring-DM.

3.2.4 iPOJO
Apache iPOJO\(^1\) is a Java-based, dynamic component framework implemented on top of the OSGi service platform.

One of the main goals of iPOJO is to make developing dynamic OSGi applications as simple as possible. To this end, the overall approach is to define components as “Plain Old Java Objects” (POJOs). The name, iPOJO, is derived from “injected POJO”, since the general approach is to inject POJOs with code to manage nonfunctional behavior so that the component code can focus on business logic. A POJO component is converted into an iPOJO component by declaring component type metadata that iPOJO uses to configure the component instance container. This metadata describes the component's run-time management requirements (see Figure 5).

\[
\begin{array}{|l|}
\hline
\text{Server component} \\
\hline
<ipojo>
\hspace{1cm}<component \hspace{3cm}classname="org.example.ServerImpl"
\hspace{1cm}name="ServerComponent">
\hspace{1cm}<provides/>
\hspace{1cm}</component>
\hspace{1cm}</instance>
\hspace{1cm}</ipojo>
\hline
\text{Client component} \\
\hline
<ipojo>
\hspace{1cm}<component \hspace{3cm}classname="org.example.ClientImpl"
\hspace{1cm}name="ClientComponent">
\hspace{1cm}<requires \hspace{3cm}field="m_server="/>
\hspace{1cm}</component>
\hspace{1cm}</instance>
\hspace{1cm}</ipojo>
\hline
\end{array}
\]

\textbf{Fig. 5.} Example of metadata in iPOJO.

iPOJO provides an extensible component container that manages all issues regarding dynamism. The container is not monolithic, but is composed of handlers (see figure 6). Each handler manages a non-functional concern. Only the required handlers are attached to the container at run time. The resulting container manages the interaction between the POJO and the external world. Custom handlers can be developed, allowing developers to handle other non-functional concerns, such as configuration, persistence, and security.

\[1\] Http://www.ipojo.org
iPOJO provides an ADL\(^2\) based on the notion of services. This kind of "composition" can be named Structural Service Composition. Generally, the application is described in terms of components or instances. iPOJO compositions are described in terms of services. This composition allows more flexibility and allows service implementations to evolve without breaking the composition.

As a composition can export/implemented a service, a composition can be used inside another composition transparently. This brings a hierarchical composition model.

### 3 Toward a common service oriented component model

#### 3.1 Objectives

Where each SOCM has its development model, and a set of others features, they have all as target: OSGi Service Platform. Those differentiations make it difficult to developers to maintain applications especially when the different pieces are made by different SOCMs.

We propose to do merging of different SOCMs for the set of bellow reasons:

- Different development model: each SOCM provide his development model, so each one treat concepts like injection, callbacks differently.
- Share common tools: in OSGi Service Platform there are some common operations like: deployment and administration. Each SOCM provide its own tools to manage these operations.

\(^2\) Architecture Description Language
− Support different descriptions: each SOCM uses a different way to describe configurations.
− And finally to have a common efforts on this development area.

### 3.2 Requirements

To have a common Service Oriented Component Model upon OSGi Service platform, we need to have these requirements:

− A simple development model
  Application developers need always a simple development model, the proposed SOCM should simplify developers tasks.

− hierarchic structural composition
  Propose concepts and mechanisms to control bindings between components and in particular the connections between components and their sub-components, that is the hierarchical structure of a system.

− introspection / reconfiguration
  Those are capabilities necessary to control the structural composition of components (component bindings, life cycle, state...). In a more general way, the needs for such mechanisms have already been motivated in “industrial component models” such as EJB or CCM, to control technical services and non-functional aspects of components.

− Extensibility
  This is a very important property that adds to the model the capability to add extra pieces to the container to handle aspects non-functional aspects.

### 3.2 Approach

Some core features of the enterprise programming models the market is moving to include:

− A focus in writing business logic in “regular” Java classes that are not required to implement certain APIs or contracts in order to integrate with a container
− Dependency injection: the ability for a component to be “given” its configuration values and references to any collaborators it needs without having to look them up. This keeps the component testable in isolation and reduces environment dependencies.
− Declarative specification of enterprise services. Transaction and security requirements for example are specified in metadata (typically XML or annotations) keeping the business logic free of such concerns. This also facilitates independent testing of components and reduces environment dependencies.
Our goal is to provide a common service component model that supports all the requirements cited above and covering the majority of discuss features. Our approach is not to introduce a new one, but choosing the good candidate and to transform other models to this one.

The descriptions made before of the most popular SOCMs, give us a good affirmation that iPOJO is a perfect challenger to be a pivot. In the next section, we give the set of reasons that proof our choice.

### 3.2.4 Why iPOJO?

Several SOCM are available and are used in industrial and Open Source projects. The most famous SOCM are Declarative Services, iPOJO and Spring DM.

However some of existing SOCM are no more supported such as Service Binder, Dependency Manager… Declarative Services implementations still not robust.

The case of Spring DM is different since Spring is very popular for Enterprise side developers and provides an bullet-proof implementation. However, the minimal footprint of Spring DM is at least 10 MB which does meet with the embedded devices constraints.

Apache iPOJO, is good candidate to be a pivot of SOCM as it provides the set of required capabilities. Indeed it provides both very simple development model and a dynamic structural composition. Moreover it provides an adequate introspection and reconfiguration layer allowing the collection of the system structure and reconfiguration of the system. In addition, iPOJO is successfully used in different domains such as residential gateways, application servers and mobile phones.

### 4 Realization

Initially, our project aims to provide migration tools for projects using 3 relevant SOCM cited previously: Service Binder, Declarative Services, and Spring DM. In the timeline of the project, only the migration tool from Spring DM to iPOJO was partially developed. The migration tool from DS to iPOJO was started by Didier Donsez.

#### 4.1 The migration tool “Spring DM 2 iPOJO”

#### 3.2.4 Goal

As the two models Spring-DM and iPOJO uses XML files to describe the configuration information, and while the two also are based on the POJO concept, an
XML based transformation can be performed between the two with some exceptions when Spring-DM application uses API classes from Spring-DM core.

### 3.2.4 Description of the transformation

Our transformer uses XSLT [13] technology to transform the set of Spring-DM configuration files to iPOJO metadata.

XSLT (the Extensible Stylesheet Language Transformations) is an XML-based language used for the transformation of XML documents into other XML documents.

The XSLT processing model involves:

- one or more XML source documents;
- one or more XSLT style sheet modules;
- the XSLT template processing engine (the processor); and
- one or more result documents.

Our transformer has the following job:

![Diagram of Spring-DM to iPOJO transformer](image)

**Fig. 7. Spring-DM to iPOJO transformer**

It can take a set of Spring-DM configuration files and make one iPOJO metadata file. The transformer will be integrated in an Ant task, and farther as an Eclipse plugin.
3.2.4 Example

We take the following example of configuration from a Spring-DM project:

```xml
<beans>
  <bean id="helloBean1" class="osgi.hello.impl.StandardHello1"/>
  <service id="helloService1" interface="osgi.hello.HelloService1">
    <bean class="osgi.hello.impl.StandardHello2"/>
  </service>
  <service id="helloService2" interface="osgi.hello.HelloService2">
    <service-properties>
      <entry key="myOtherKey" value="aStringValue"/>
    </service-properties>
  </service>
  <service ref="helloBean1">
    <interfaces>
      <value>osgi.hello.HelloService3</value>
      <value>osgi.hello.HelloService4</value>
    </interfaces>
  </service>
</beans>
```

Fig. 8. Spring-DM configuration file

By using our developed XSLT Style sheet (Figure 10), we will be able to generate the iPOJO metadata file (Figure 9).

The style sheet consists of a set of templates that matches elements in Spring-DM configuration files, and produces new data. The XSLT support condition statements as well as loop instructions.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ipojo>
  <component name="helloBean1">
    <classname>osgi.hello.impl.StandardHello1</classname>
    <provides interface="osgi.hello.HelloService1"/>
    <provides interface="osgi.hello.HelloService2">
      <property name="myOtherKey" type="java.lang.String" value="aStringValue"/>
    </provides>
    <provides>
      <interfaces>
        <value>osgi.hello.HelloService3</value>
        <value>osgi.hello.HelloService4</value>
      </interfaces>
    </provides>
  </component>
  <instance componentet="helloBean1"/>
</ipojo>
```

Fig. 9. The generated iPOJO metadata
This actual version of the transformer takes on consideration only the registration of services aspects. Other aspects will be taken on the next step.
5 Conclusion

By August 5th, the OSGi Alliance has proposed an early draft of the 4.2 version of OSGi specification. One of the big changes of this update is the introduction of a common Component Model for OSGi. This model is inspired from Spring DM and iPOJO projects.

Therefore, a lot of applications will migrate to this new standard, thus the need of transformers to do migration of existing applications to support this new specification.

References

2. RS Hall, H Cervantes : “Challenges in building service-oriented applications for OSGi”, Communications Magazine, IEEE, 2004
13. W3C : “XSL Transformations”, http://www.w3.org/TR/xslt