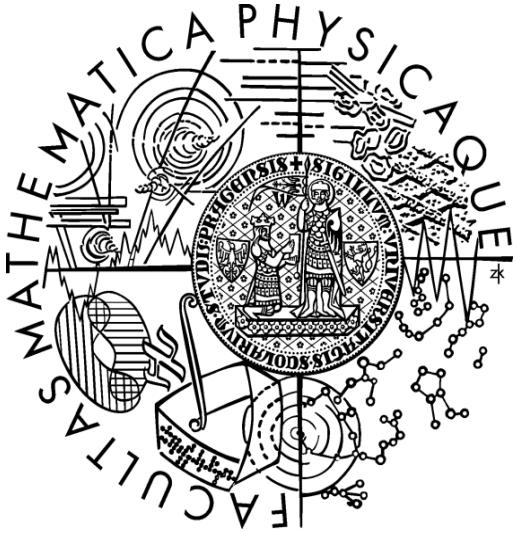
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Karel Mašek

Interoperability between component-based and service-oriented systems

Department of Software Engineering
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Study Program: Informatics, Software systems

Chtěl bych poděkovat vedoucímu práce RNDr. Petru Hnětynkovi, Ph.D. za rady a užitečné připomínky, provázející tvorbu této práce. Dále, velké poděkování rodině a blízkým za podporu během psaní práce a po čas studia. Díky za Javu a podporu spolupracovníkům ze společnosti Sun Microsystems.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 8.8.2008 Karel Mašek

Contents

1. Introduction....8
1.1. Component-based systems....8
1.2. SOA-based systems....9
1.3. Interoperability....10
1.4. Goals of the thesis....10
1.5. Structure of the text....11
2. Background....12
2.1. SOFA2 component system....12
2.1.1. Component model....13
2.1.2. Control part of components....14
2.1.3. Utility interface pattern....15
2.2. OSGi Service Platform....16
2.2.1. The Framework....16
2.2.2. Bundles....18
2.2.3. Service model....19
3. Integrating SOFA2 and OSGi....21
3.1. Embedding OSGi Framework....21
3.2. Extending SOFA2 meta-model....22
3.3. Service Tracker aspect....23
3.4. ServicePublisher aspect....24
3.5. OSGi Service Tracker controller....24
3.5.1. Service Proxy....26
3.5.2. Service Events Listener....29
3.6. OSGi Service Publisher controller....30
3.6.1. Proxy bundle....32
3.7. Cushion....34
3.7.1. Command osgi....34
4. The usage and use cases....37
4.1. Using OSGi services....37
4.1.1. Service Listener interface....38
4.1.2. UPnP robot example....39
4.2. Publishing SOFA2 interfaces as OSGi services....42
4.2.1. Dictionary service example....43
4.3. Remoting for OSGi....46
5. Related work....47
5.1. Spring Framework and OSGi....47
6. Conclusion and future work....50
6.1. Goals review....50
6.2. Future work....51
7. References....52
. Appendix A: Content of the CD....54
. Appendix B: OSGi deployment dock....55

List of figures

Figure 1: publish-find-bind pattern....10
Figure 2: SOFA2 runtime environment....13
Figure 3: SOFA2 application example....14
Figure 4: Utility interface example....15
Figure 5: OSGi architectural overview....16
Figure 6: Interaction between layers in the Framework....17
Figure 7: Bundle state diagram....19
Figure 8: OSGi Service model....20
Figure 9: Embedding OSGi Framework....22
Figure 10: Service Tracker control interface....23
Figure 11: Service Publisher control interface....24
Figure 12: OSGi Service Tracker controller....26
Figure 13: Service Proxy handling a method invocation....28
Figure 14: SOFAServiceListener interface....29
Figure 15: RegistrationListener interface....30
Figure 16: Service Publisher aspect....31
Figure 17: Using OSGi services....37
Figure 18: SOFAServiceListener interface....39
Figure 19: UPnP robot example....40
Figure 20: Publishing OSGi services....43
Figure 21: Dictionary service example....44
Figure 22: Remoting for OSGi services....46

Název práce: Interoperabilita mezi komponentovými a servisně orientovanými systémy
Autor: Karel Mašek
Katedra (ústav): Katedra sofwarového inženýrství
Vedoucí diplomové práce: RNDr. Petr Hnětynka, Ph.D.
e-mail vedoucího: Petr.Hnetynka@dsrg.mff.cuni.cz

Abstrakt:

Pro vývoj a návrh rozsáhlých softwarových systémů, se používají převážně dva přístupy: komponentový a servisně-orientovaný návrh. V systémech, kde se oba přístupy kombinují, je interoperabilita (t.j. jejich vzájemná spolupráce) klíčovou vlastností.

Cílem práce je navrhnout a experimentálně implementovat řešení pro interoperabilitu mezi komponentovým systémem SOFA2 a servisně orientovanou platformou OSGi.

Výsledné řešení je založené na použití aspektů a annotací. Annotace slouží k deklarativnímu označení komponent, které využívají (t.j. volají nebo publikují) OSGi servisy. Naopak, pomocí aspektů se označeným komponentům poskytne OSGi funkcionalita, t.j. kontrolní logika. Kromě komponentového systému SOFA2, byla podpora pro OSGi přidána i do nástroje, který slouží pro vývoj SOFA2 aplikací. Navržené řešení je použitelné nejen pro integraci SOFA2 a OSGi, ale i obecně pro komponentové a servisně-orientované systémy.

Klíčová slova: interoperabilita, komponentové systémy, servisně-orientované systémy

Title: Interoperability between component-based and service-oriented systems
Author: Karel Mašek
Department: Department of Software Engineering
Supervisor: RNDr. Petr Hnětynka, Ph.D.
Supervisor's e-mail address: Petr.Hnetynka@dsrg.mff.cuni.cz

Abstract:

The component-based and service-oriented development have become commonly used techniques for building high quality, evolvable, large systems in a timely and affordable manner. In this setting, interoperability is one the essential issues, since it enables the composition of heterogeneous components and services.

The aim of the thesis is to analyze possibilities of interoperability between the SOFA2 component system and the OSGi Service Platform, and based on that propose and implement a solution for mutual collaboration.

The actual integration is based on the use of aspects and annotations. The issues connected with the runtime service management (e.g. binding/unbinding services) are handled by the control part of components using the aspects. While, the annotations serve for specifying service-enabled SOFA2 components in a declarative way. The OSGi support is incorporated in both the SOFA2 runtime environment and the tool for developing SOFA2 components. Furthermore, the outlined approach is general and can be easily reused for integrating other SOA-based systems as well.

Keywords: interoperability, component-based systems, service-oriented systems

1 Introduction

Designing and implementing a large scale, evolvable, enterprise software system is a challenging task. The component-based and service-oriented development are the two key techniques for building such a system in timely and affordable manner. In this setting, interoperability is one the essential issues, since it enables the composition of heterogeneous components and services, developed by different people, at different times, and possibly with different uses in mind.

The component-based development (CBD) [1] provides support for building software systems through the composition and assembly of software components. It has become a commonly used approach in many software engineering domains, such as enterprise and web-based systems, desktop and graphical applications, and recently in the embedded system domain. Unlike former development techniques, components allow for specifying services provided by them, as well as, services required from other components and/or the environment. Thanks to this, components has brought easier reuse, integration and rapid development of application.

Currently, there are many component-based systems used both in industry and academia. The industrial component systems, such as EJB [2] and CCM [3], are typically relatively mature. They provide stable runtime environment, convenient user interface to control the life-cycle of component-based applications, etc. On the other hand, they don't address advanced features like the hierarchical component model, multiple communication styles, composition verification, support for seamless distribution, etc. Such advanced features are addressed mostly by the academical component systems (e.g. SOFA2 [4] and Fractal [5]).

The service-oriented architecture (SOA) [6] is another paradigm for building software systems. The functionality is grouped around business processes and packaged as interoperable services. It promotes a loose coupling of services to minimize their dependencies, thus reducing the risk that a change in one part of an application will force a change in other parts. The SOA paradigm is implemented by many systems (e.g. WebServices [7], OSGi [8], etc.) that are commonly used in the production environment. It has found use in enterprise and web-based applications, in various module systems, in embedded system, etc. Furthermore, the SOA paradigm is heavily used in the enterprise integration domain.

1.1 Component-based systems

The component-based development (CBD) has gained recognition as the key technology for building large scale enterprise systems. The main goal is to significantly increase software reusability and shorten time to market. It has found use in many software engineering domains, such as distributed and web-based systems, desktop and

graphical applications, and recently in the embedded system domain. It provides support for building software systems through the composition and assembly of software components. Unlike former development techniques, components allow for specifying services provided by them, as well as, services required from other components and/or the environment. Thanks to this, components has brought easier reuse, integration and rapid development of application.

There are many definitions of what a component is. Typically, a component is considered to be a *black-box* software entity with well-defined interfaces and behavior that can be reused in different contexts and without knowledge of its internal structure. The set of component features, composition rules, etc. is referred to as a *component model*.

From the composition point of view, component models can be divided into two categories – flat and hierarchical component models. Unlike the flat ones, the hierarchical component models allow for building composite components – a component is hierarchically composed of other components. In this case, a component can be viewed as a *gray-box* entity with an internal structure (i.e. a number of interconnected sub-components).

Currently, there are many component-based systems used both in industry and academia. The flat component systems, such as EJB [2] and CCM [3], are typically relatively mature. They provide stable runtime environment, convenient user interface to control the life-cycle of component-based applications, etc. On the other hand, they don't address advanced features like the hierarchical component model, multiple communication styles, composition verification, support for seamless distribution, etc. Such features are addressed mostly by the academical component systems (e.g. SOFA2 [4] and Fractal [5]).

1.2 SOA-based systems

Service Oriented Architecture (SOA) is an architectural style for building software systems and applications. It utilizes *services* as the fundamental units of functionality. A service exposes its functionality through a well-defined interface, which is then used by service clients and other services to invoke the service. The service implementation and the interface are usually decoupled.

The service-oriented design facilitates seamless integration of distributed systems that are built on various platforms and technologies. Further, it pushes focus on software reusability and development efficiency.

There are many common principles and patterns used throughout different SOA implementations. An example of such a principle is *loose coupling* of services that minimizes service dependencies, thus reducing the risk that a change in one part of an application would force changes in other parts. On the other hand, one of the most

essential patterns is the *publish-find-bind* pattern, depicted in Figure 1.

The *publish-find-bind* pattern describes the relationship among service providers, consumers and a service registry. A service provider can *publish* services in the service registry. A service consumer (client) then queries the registry to *find* appropriate service. In response, the consumer gets a service handle, e.g. a service reference or URL. The handle it used for *binding* the service in order to invoke it, later on.

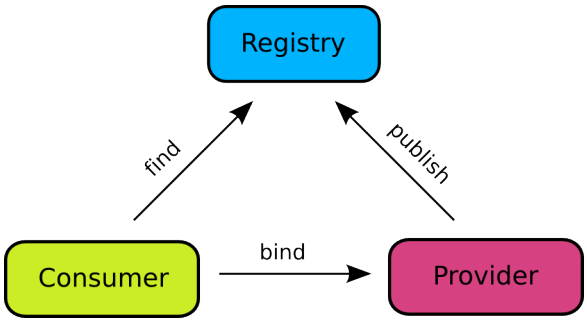


Figure 1: publish-find-bind pattern

The SOA paradigm is employed by many software systems and platforms (e.g. WebServices [7], the OSGi Service Platform [8]) that are commonly used in the production environment. It has found use in wide range of software engineering domains, from enterprise and web-based applications to embedded systems. Furthermore, the SOA is heavily used in the enterprise integration domain (e.g. JBI [9]).

1.3 Interoperability

Interoperability is a property referring to the ability of diverse systems to work together, exchange data and make use of the data that has been exchanged. In terms of software systems, this is achieved by using a common set of exchange formats, interaction patterns and the same protocols.

In a high-level view, the SOA and CBD paradigms are similar to each other in many senses [10] – both a service and component have a well-defined interface, their internal structure is not visible to their environment, and they can be reused in different contexts. Despite of the similarities, each approach has different targets though many of them overlap.

The SOA paradigm is preferable for exposing some coarse-grained functionality (through services). While, components are suitable for providing finer-grained business logic. Both approaches may be combined in heterogeneous software systems (e.g. for wrapping a legacy system). Therefore interoperability is an essential issue.

1.4 Goals of the thesis

The general goal is to design and implement a solution to provide interoperability between the SOFA2 component system and the OSGi Service Platform. In more details, the thesis sets out the following goals:

(g1) Mutual interoperability

SOFA2 components can both access and publish OSGi services.

(g2) Seamless integration

It should be easy for SOFA2 components to use and incorporate OSGi services.

(g3) Handle service dynamics

Provide means to deal with the dynamic nature of OSGi services.

(g4) General approach

The proposed solution should be general so that the principles can be easily reused when integrating other SOA-based systems.

1.5 Structure of the text

Chapter one overviews the basic aspects of component-based and service-oriented systems. And presents the goals of the thesis.

Chapter two examines relevant parts of the SOFA2 component system and the OSGi Service Platform with respect to mutual interoperability.

Chapter three presents the proposed solution to provide interoperability between the SOFA2 component system and the OSGi Service Platform. It starts with showing the architectural overview of the integration and the way how the OSGi Framework is incorporated in the SOFA2 runtime. Next, it describes the newly introduced annotations that serve for specifying service-enabled components in a declarative way. Further, it examines the actual runtime interaction between OSGi services and SOFA2 interfaces. Finally, it describes the OSGi support offered by *cushion*, the command-line tool for developing SOFA2 components.

Chapter four demonstrates using OSGi services in practice. The presented examples to illustrate (in a step-by-step manner) how to create and set up a SOFA2 component in order to enable the component to access and publish OSGi services.

Chapter five discusses the related work.

Chapter six presents a summary of the thesis and evaluates the goals. Suggestions for the future work are also mentioned.

2 Background

This chapter describes the SOFA2 component system and the OSGi Service Platform in more details. Particularly, it examines features that are relevant to mutual interoperability.

2.1 SOFA2 component system

SOFA2 is a component based system that employs a hierarchical component model. Apart from that, it supports many advanced features like multiple communication styles, dynamic component updates, support for versioning, seamless distribution, clearly separated functional and control parts of components, support for SOA concepts, composition and behavior verification, etc. SOFA2 is an academic component based system, developed by the Distributed Systems Research Group [11] at Charles University in Prague. The prototype implementation, written in Java, is available as an open-source software.

A component is described by its *frame* and *architecture*. A frame is a black-box view of the component. It defines the provided and required interfaces. On the other hand, the frame is implemented by an architecture that serves as a gray-box view of the component. It specifies the internal structure of the component – the subcomponents and the bindings among their interfaces.

A SOFA2 application is executed in a distributed environment that consists of a number of *deployment docks*. A deployment dock is a component container hosted on a particular computer and providing the runtime environment for executing SOFA2 components. An application can span several deployment docks. The assignment of components to a particular deployment dock is done during the deployment.

Apart from the deployment docks, the SOFA2 runtime environment contains also a repository that serves as a storage of meta-data (component descriptions, deployment plans, etc.) and the code of components. The repository is used throughout the whole application life-cycle, namely the development, assembly, deployment and execution phases. All entries stored in the repository are versioned. The versioning model used in SOFA2 is described in [12].

An example of a SOFA2 application with respect to the runtime environment is depicted in Figure 2.

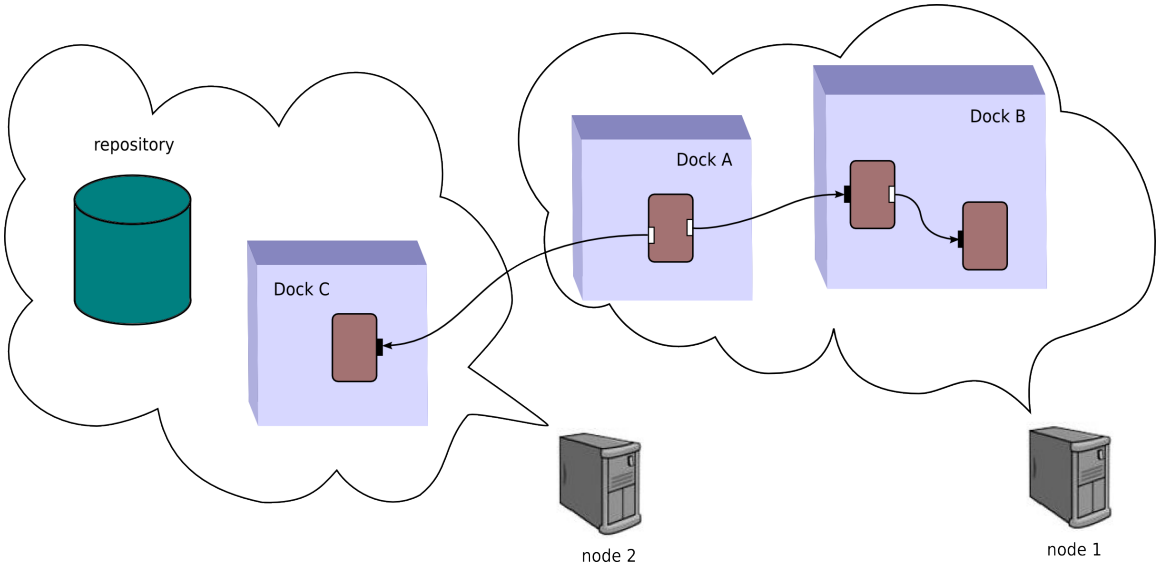


Figure 2: SOFA2 runtime environment

2.1.1 Component model

The component model and all its features are defined using a *meta-model* [13]. The meta-model directly serves for the component specification, instead of an ADL[1]. The specification stored in the repository and used throughout the application life-cycle.

Being a hierarchical model, it allows components to be hierarchically nested. Components can be either *primitive* or *composite*. A composite component is built of other components, while a primitive one contains no subcomponents.

The core element is the *frame*, which specifies the provided and required interfaces. Each *interface* has an *interface type* that is specified by its *signature* and the *code-bundle*. The signature is a fully qualified class name of the interface. While, the code-bundle holds the interface's classes (the interface class and the method types). In addition to the type, the interface has other attributes. The *communication-style* and *communication-feature* attributes allow for specifying the way components (via the interface) can communicate. Moreover, the interface can specify its *cardinality* (single or collection), *contingency* (optionally or mandatory connected) and *connection type.* The connection type has to possible values – normal or utility. For more details, see below.

The frame is implemented by an *architecture*. A single architecture can implement several frames, as well as, a frame can be implemented by several architectures. The architecture of a composite component specifies the *subcomponents* and the *bindings* among them. The bindings are performed using connectors [14] that are dynamically

1 Architecture Description Language

generated at deployment time.

On the other hand, the architecture of a primitive component is empty and the component directly implements the corresponding frame. The implementation (Java classes) is stored in the repository as a *code-bundle.*

An example of a SOFA2 application with respect to frames and architectures is depicted in Figure 3.

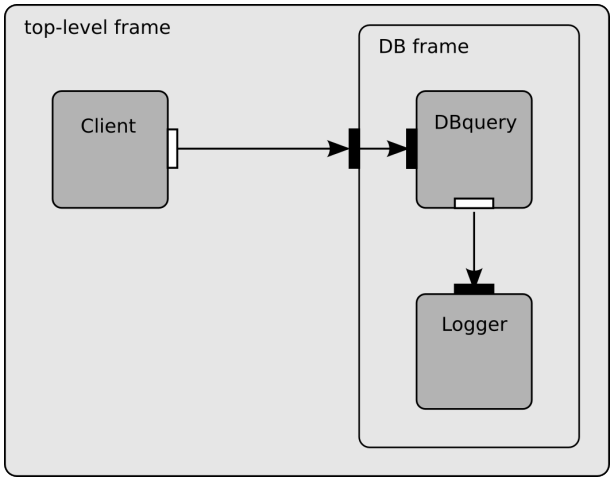


Figure 3: SOFA2 application example

Aside from that, frames and architectures can have *properties*, which are name-value pairs. They are used for component parametrization at deployment time.

Moreover, *annotations* allow to annotate frames and interfaces with additional information. For example, they can be used for specifying non-functional features.

The *interface type*, *frame* and *architecture* elements, as well as, *code-bundles* are stored in the SOFA2 repository.

2.1.2 Control part of components

In addition to business interfaces (i.e. provided and required interfaces), components have so-called *control interfaces* that correspond to the non-functional features of the component, like life-cycle management, introspection, etc. They are not usually accessed by the application logic, but rather by the runtime environment.

The control part of components in SOFA2 is modular and extensible. It is based on usage of aspects. The general idea of this approach is described in [15]. The control part of a component is modeled as composed of *microcomponents*. The microcomponent model is flat (i.e. microcomponents cannot be hierarchically nested) with no advanced features (distribution, connectors, etc.). Additionally, to avoid recursion, a microcomponent doesn't have any extensible or structured control part.

On the top of that, microcomponents are organized into aspects. An *aspect* represents a consistent extension of the control part. It defines what microcomponents to instantiate and how to incorporate them into the existing control part. Furthermore, it may introduce a new control interface to provide another entry point to the control part of the component.

2.1.3 Utility interface pattern

Although, the SOFA2 is a component based system, it incorporates a basic support for services and eventually allows for service-oriented architectures (SOA). It allows marking interfaces as *utility interfaces*, and relaxes on some rules for handling such interfaces – a required utility interface may be freely bound and unbound at runtime, etc. Moreover, the reference to a utility interface can be freely passed among components and the connection to the interface is established orthogonally to the architecture hierarchy. An example of using the utility interface is shown in Figure 4.

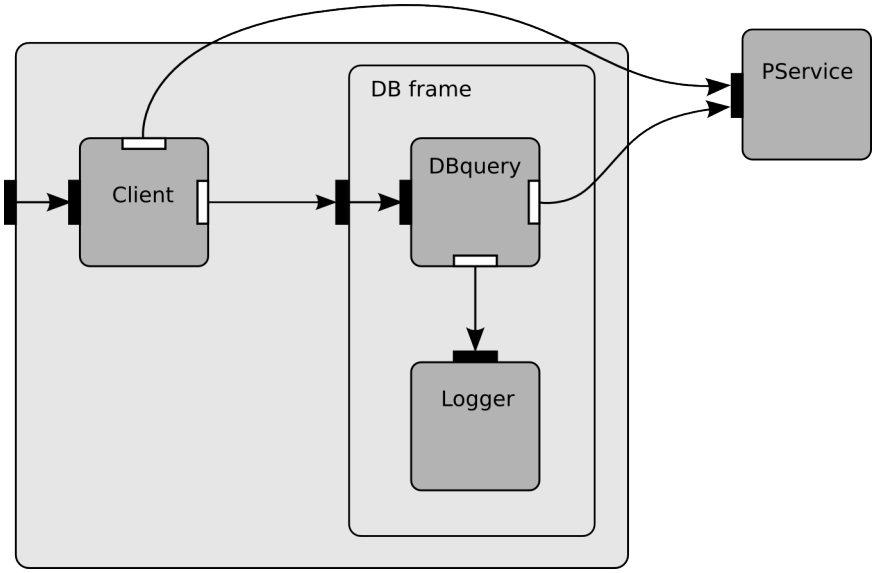


Figure 4: Utility interface example

2.2 OSGi Service Platform

The OSGi Service Platform [8] is a Java-based dynamic module system that employs a service-oriented paradigm for the module collaboration. It was developed by the OSGi Alliance, which is a non-profit open standards organization founded in March 1999. A software system is partitioned into a number of reusable and manageable modules that can be composed into an application and deployed. Initially, the specification was targeted at embedded Java and network devices. However it has found use in everything from mobile phones to cars, and recently even in enterprise applications.

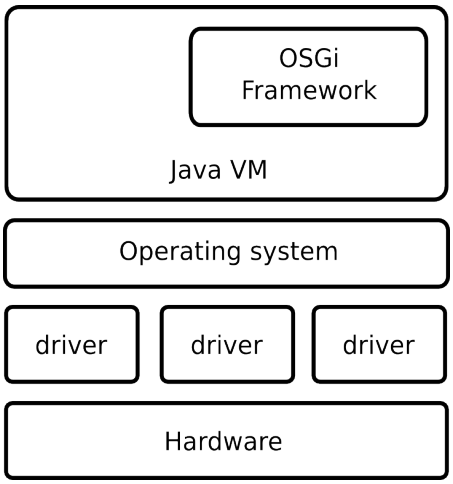


Figure 5: OSGi architectural overview

The core of the specification is the *Framework*. It defines modules (called *bundles),* their life-cycle model, service registry, security model and execution environment.

On top of the Framework, the specification defines a number of standard services [16]. For example, services to control the Framework, system services (like logging, preferences, etc.), protocol services (e.g. http, UPnP) and many more. At the time of writing, the latest release of the specification was Release 4.1 in May 2007.

The architectural overview of the OSGi Framework is depicted in Figure 5.

The OSGi platform is dynamic. The Framework manages bundles' installation, uninstallation and updates at runtime (i.e. without having to restart the Framework). It is service-oriented, bundles can dynamically *publish* services. While, other bundles can query the services through the OSGi Service Registry.

Services are used for the communication between bundles inside a single JVM[2]. A bundle can register any number of services – it uses the fully qualified name of the service interface when registering a service. Other bundles can track a service to become available (or otherwise) and respond accordingly. As services are for intra-JVM communication, the service calls are plain method invocations.

2.2.1 The Framework

The core part of the OSGi Service Platform specification is the Framework. It provides a standardized execution environment to bundles. The functionality of the Framework is divided in the following layers:

2 Java virtual machine

- Security layer
- Module layer
- Life Cycle layer
- Service layer
- Actual services

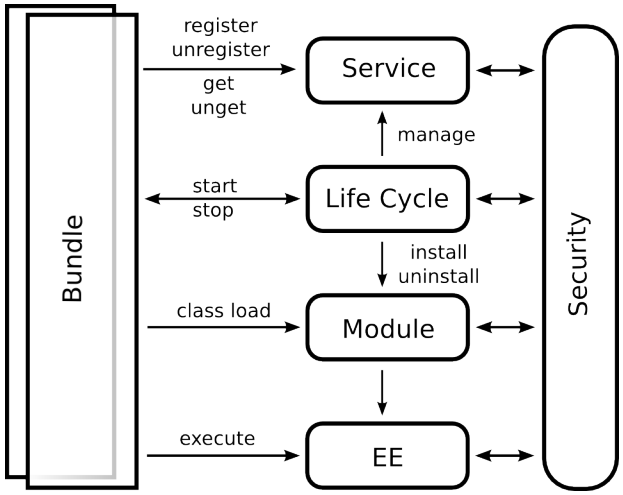


Figure 6: Interaction between layers in the Framework

The layering and the interaction between layers is shown in Figure 6.

The Security Layer is an optional layer. The rest of the layers make use of the Security Layer to provide a fine-grained controlled environment. The security model is based on Java 2 Security Architecture [17].

The Module Layer deals with modularity. It defines a module system that addresses some of the shortcomings of the standard Java modularization and deployment model. The Framework defines a unit of modularization, called a *bundle*. There are strict rules for sharing packages between bundles and hiding packages from other bundles. Bundles must specify what packages they export and import. A bundle can specify a version for each package being exported and a version range for packages being imported by the bundle. Bundles are resolved at runtime. The resolution process ensures that all bundle's dependencies are satisfied. All these features allow to keep implementations private and expose API's only.

On the top of that, the Life Cycle Layer provides a dynamic runtime model for bundles. It defines how bundles are installed, started, stopped and uninstalled. Furthermore, an installed bundle can be updated anytime. A bundle can register event listeners and get notified of other bundles' life-cycle events. The events are delivered either synchronously or asynchronously.

The Service Layer provides a dynamic service model for the communication between bundles. It contains the Service Registry, which is used for registering service and querying for services. A bundle can register any number of services – it uses the fully qualified name of the service interface when registering a service. Furthermore, a service can be registered with the set of key/value properties. Both the service name and properties can be used when querying the Service Registry.

The OSGi Service Platform specifies a number of execution environment profiles to run on different target devices. The Java Profiles, like J2SE, CDC, MIDP etc. are all valid

execution environments. Bundles that are restricted to run in a certain execution environments can enforce the Framework's execution profile.

In addition to normal bundles, the Framework presents itself as a *System Bundle*. The system bundle usually registers services that can be used by other bundles to control the Framework, such as the Package Admin and Permission Admin services.

2.2.2 Bundles

In OSGi, bundles are units of functionality, modularity and deployment. A bundle is packaged as a simple JAR[3] file. It is comprised of Java classes, the bundle's manifest file and other resources.

A bundle carry its settings and descriptive information about itself in the manifest file. The Framework defines several OSGi manifest headers such as `Export-Package` and `Bundle-Classpath`. The headers can specify a bundle's name, exported and imported packages, dependencies on some other bundles, required execution environment, etc.

Many bundles may share a single JVM. Within the virtual machine, bundles can hide packages and classes, as well as, share them with other bundles. For the purpose, each bundle uses its own class-loader.

From a bundle point of view, the bundle's visibility consists of:

- parent class loader (normally `java.*` packages from the JVM class-path)
- imported packages
- exported packages
- required bundles
- bundle's class path (private packages)
- attached fragments

A bundle is started though its *Bundle Activator*. The manifest file specifies the class that implements the `BundleActivator`[4] interface. The *start()* and *stop()* methods are called when a bundle is started and stopped, respectively.

Bundle state

A bundle can be in one of the following states:

- INSTALLED – the bundle has been successfully installed
- RESOLVED – all bundle dependencies are resolved, the bundle is either ready to be started or has stopped

3 Java ARchive

4 `org.osgi.framework.BundleActivator`

- STARTING – the bundle is being started
- ACTIVE – the bundle has been successfully activated and is running
- STOPPING – the bundle is being stopped
- UNINSTALLED – the bundle has been uninstalled and cannot move into another state

The bundle states and transitions between the stated are shown in Figure 7. Note, the full and dashed lines represent the explicit and automatic transitions, respectively.

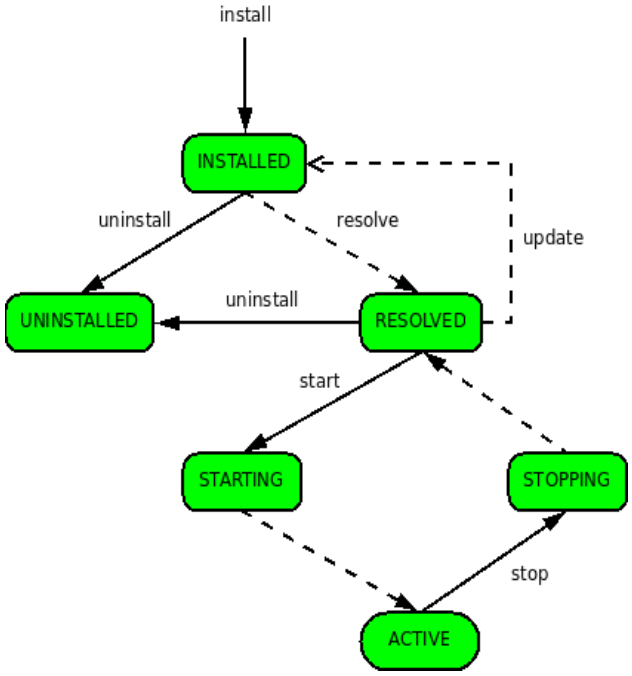


Figure 7: Bundle state diagram

2.2.3 Service model

The Service Layer defines a dynamic collaborative service model. It implements one of the key SOA[5] patterns – the *publish-find-bind* pattern, see Sect. 1.2. The Framework defines the Service Registry to store service registrations. Bundles can register any number of services and query the registry for services. Furthermore, a bundle can register an event listener and explicitly track life-cycle events of services. The interaction between bundles and the Service Registry is depicted in Figure 8.

In fact, OSGi services are simple Java object. The service model allows bundles to share objects between each other.

5 Service Oriented Architecture

To register a service, the following is needed:

- *service name* – the fully qualified name of the service interface
- *service object* – the service implementation
- *properties* – a map of key/value pairs

Both the service name and properties may be used when querying the Service Registry. The OSGi makes a heavy use of *filter* expressions to query services. The syntax is based on the LDAP search filters as defined in [18].

The Framework sends out *service events* to report registrations, unregistrations and property changes of services. Bundles can register a *service listener* to get notified when a service event is fired.

A *service reference* provides access to a service properties but not the actual service object. The service object must be acquired through a bundle's execution context.

Implementing the *ServiceFactory* interface allows the registering bundle to customize the service object for each using bundle.

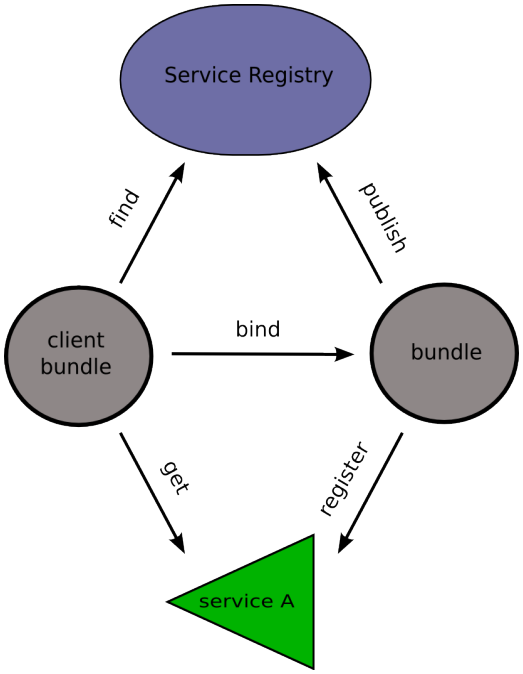


Figure 8: OSGi Service model

3 Integrating SOFA2 and OSGi

This chapter presents the proposed solution to provide interoperability between the SOFA2 component system and the OSGi Service Platform. It starts with showing the architectural overview of the integration. And the way the OSGi Framework is incorporated into the SOFA2 runtime.

Further, it describes the newly introduced annotations, which are used with SOFA2 components (i.e. their frames and interfaces). The frame annotations serve for specifying components that directly access and/or publish OSGi services. While, the interface annotations are used to mark out the interfaces that serve for accessing services, and those to be published as services.

Next, it examines the actual runtime interaction between OSGi services and SOFA2 interfaces. The service management issues (e.g. binding/unbinding services) are handled by the control part of components. In more details, the SOFA2 runtime applies (based on the frame annotations) corresponding aspects and provides components with the OSGi functionality (through the OSGi controller).

Method invocations (on the annotated interfaces) are handled by a *Service Proxy*, which acts as a mediator between the interface and OSGi services. The services behind the proxy may come and go dynamically. It is up to the corresponding OSGi controller to keep the proxy up-to-date so that it reflects the availability (or otherwise) of the service.

Finally, it describes the OSGi support that is provided by *cushion*, the command-line tool for developing SOFA2 components.

3.1 Embedding OSGi Framework

A SOFA2 application comprises a number of components. Components are hosted by deployment docks that provide the runtime environment and serve as component containers. In order to support the OSGi-enabled SOFA2 components, a new type of deployment dock has been added. To provide the OSGi runtime, the dock launches an embedded instance of the Framework when it starts. The launching scripts are included in the SOFA2 distribution. For more details, see Appendix B.

There are many implementations of the OSGi Service Platform specification, from open-source to commercial ones. They differ in the portion of the specification that they implement (e.g. number of services), as well as, in maturity, stability and licensing. Based on that, the Apache Felix was an obvious choice.

Apache Felix is an open-source implementation of the OSGi R4 Service Platform specification [8]. It includes the Framework functionality and a set of standard services. The Felix project is a community effort to provide the full-compliant implementation of the OSGi specification. Currently, a larger portion of the specification is implemented

and the Framework functionality is very stable. Furthermore, Felix's license is compatible with the license used by SOFA2.

An example of a SOFA2 application is depicted in Figure 9. It shows two deployment docks – the dock A provides support for OSGi-enabled components (C1 and C2), and the dock B that hosts standard SOFA2 components (C3 and C4). The OSGi services (S1 and S2) are accessed through dedicated SOFA2 interfaces. The provided interface (C1) makes the service (S1) accessible to the component (C3) that resides in the dock B (not embedding OSGi). While, the required interface serves for accessing the service (S2) by the component (C2) itself. The service management issues, like binding services, are described later on.

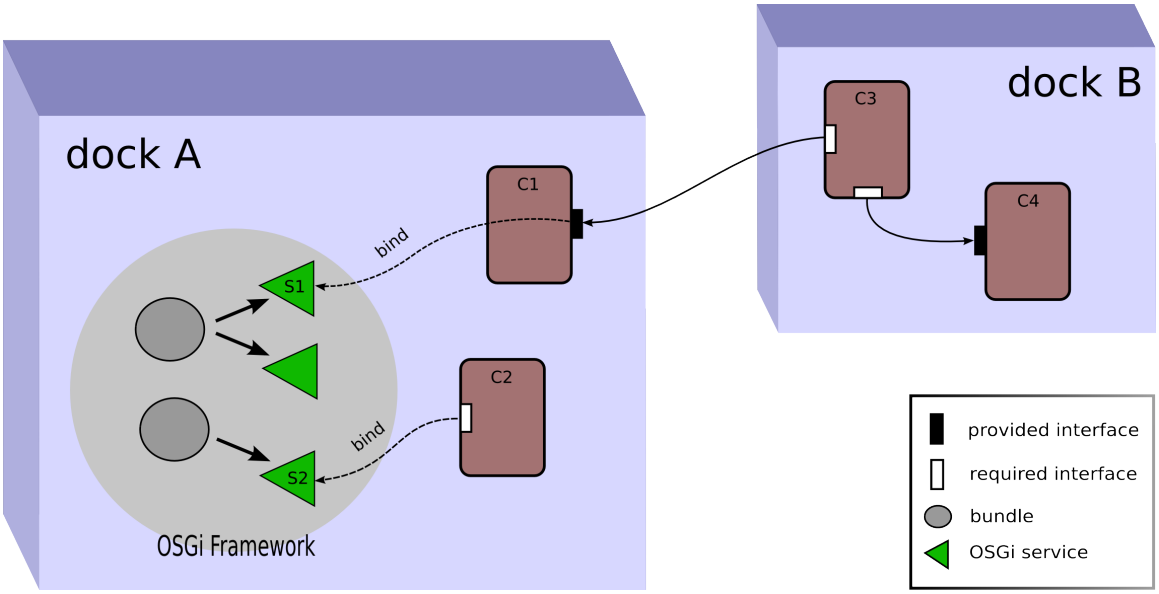


Figure 9: Embedding OSGi Framework

3.2 Extending SOFA2 meta-model

In order to specify service-enabled components, several new annotations have been introduced in the SOFA2 meta-model. They are used with frames and their business interfaces. The newly introduced annotations are as follows:

- @ServiceTracker
- @ServicePublisher
- @Service
- @Publish

The ***@ServiceTracker*** and ***@ServicePublisher*** annotations are used with frames to specify the SOFA2 components that access and publish services, respectively. The

annotations are utilized by the SOFA2 runtime to apply corresponding aspects and to provide components with the desired functionality. The controller that implements the aspect and thus the underlying service registry is determined by the *type* attribute. For example, the @ServiceTracker(type=”osgi”) annotation is used with the components that access OSGi services.

The ***@Publish*** annotation is used with SOFA2 interfaces to mark out the ones that are to be published as services (e.g OSGi services). The annotation may carry a map of key/value properties, which are then used when registering the service. Note, the @Publish annotations are valid only within @ServicePublisher annotated frames.

The ***@Service*** annotation is used with SOFA2 interfaces to mark out the ones that serve for accessing services (e.g. OSGi services). The annotation attributes carry the interface configuration and specify the set of target services. The *name* attribute specifies a service name; the optional attribute, *filter,* is used for further constraining the set of target services. As services may become unavailable at any time, the *timeout* attribute specifies the time (in milliseconds) to wait up for the service to become available again. Note, the @Service annotations are valid only within @ServiceTracker annotated frames.

3.3 Service Tracker aspect

The Service Tracker aspect has been introduced to provide components with the control logic to manage services (e.g. binding/unbinding) and to enable components to access service through their interfaces (i.e. the @Service annotated ones). The aspect is applied (by the SOFA2 runtime) to components with the @ServiceTracker annotated frames. The affected components are supplied with the Service Tracker control interface (Fig. 10) and a controller that implements the aspect (based on the *type* attribute). The control interface is used to direct the controller's life-cycle.

```
public interface MIServiceTracker {
    void open();
    void close();
}
```

Figure 10: Service Tracker control interface

The *open()* and *close()* methods are called (by the SOFA2 runtime) when the component is started and stopped, respectively.

The controller that implements the aspect and thus the underlying service registry is determined by the *type* attribute of the @ServiceTracker annotation. Frame *properties* may carry additional configuration for the controller (e.g. the URL of the service registry). For information about the OSGi Service Tracker controller, see Sect. 3.5.

The set of services being tracked (i.e. a service is bound when becomes available and vice versa) by a single @Service annotated interface is determined by the annotation attributes. The *name* attribute specifies a service name – only services registered (in the

service registry) under the given name are tracked; the optional *filter* attribute is used for further constraining the set of target services, e.g., by the service properties. The *timeout* attribute is used to configure the time to wait up for a service to become available before failing.

When the controller is active (the *open()* method was called), it is supposed to *find* suitable services (specified by the @Service annotations) and *bind* them to the corresponding interfaces. The controller should release all the services when the *close()* method is called.

3.4 Service Publisher aspect

The Service Publisher aspect has been introduced to provide components with the control logic to publish/unpublish their interfaces as services. The aspect is applied (by the SOFA2 runtime) to components with the @ServicePublisher annotated frames. The affected components are supplied with the Service Publisher control interface (Fig. 11) and a controller that implements the aspect (based on the *type* attribute). The control interface is used by the SOFA2 runtime to direct the controller's life-cycle.

The *publish()* and *unpublish()* methods are called (by the SOFA2 runtime) when the component is started and stopped, respectively.

```
public interface MIServicePublisher {
    void publish();
    void unpublish();
}
```

Figure 11: Service Publisher control interface

The set of interfaces that are to be published as services (e.g. OSGi services) is determined by the @Publish annotations.

The controller that implements the aspect and thus the service registry that is used for publishing services, is determined by the *type* attribute of the @ServicePublisher annotation. Frame *properties* may carry additional configuration for the controller. For information about OSGi Service Publisher controller, see Sect. 3.6.

When the controller is started (the *publish()* method was called), it is supposed to *publish* the @Publish annotated interfaces as services in the underlying service registry. The service implementation is provided by the corresponding SOFA2 component. The controller is supposed to unpublish all the services when the *unpublish()* method is called.

3.5 OSGi Service Tracker controller

The OSGi Service Tracker controller provides the implementation for the Service Tracker aspect. It handles the runtime interaction between SOFA2 interfaces and OSGi services, e.g. binding and unbinding services. A component is provided with the controller, when its frame is annotated with *@ServiceTracker(type="osgi")* annotation.

The controller implements the Service Tracker control interface (Fig. 10). It is used by the SOFA2 runtime for starting and shutting down the controller. The *open()* method is called to start tracking and binding suitable services. On the contrary, the *close()* method stops the controller and it releases all services that have been bound.

The @Service annotations determine the set of interfaces that serve for accessing services. While, the annotation attributes specify the services to be tracked by the controller. A service is made accessible via the corresponding @Service annotated interface, either a provided or required one. A provided interface makes a service accessible to other SOFA2 components. While, the required ones serve for accessing services by the component itself.

Every @Service annotated interface is backed up by a *Service Proxy* and a *Service Tracker* objects. The proxy handles method invocations on the interface and makes sure that the matching service method is invoked. While, the tracker keeps track of services as they come and go and updates the proxy accordingly.

The set of services being tracked by a single tracker instance is determined by the attributes of the corresponding @Service annotation. The service name is specified by the *name* attribute; the *filter* attribute specifies an LDAP[6] filter expression for further constraining the set of target services. The filter matches services based on their properties.

The controller makes use of the OSGi `ServiceTracker`[7] utility class [16], for the purpose of tracking services. The Framework sends out a service event and notifies the corresponding `ServiceTracker` instance when a service is registered/unregistered. The tracker binds/unbinds the service and updates the related proxy accordingly. So that it reflects the availability (or otherwise) of the service.

It is possible for a component to explicitly track related service events by implementing the *`SOFAServiceListener`* interface, see below.

The overall view of the OSGi Service Tracker controller is depicted in Figure 12. It shows a primitive SOFA2 component, the component's frame is annotated with the @ServiceTracker(type=”osgi”) annotation. The interfaces A and B are annotated with the @Service(name=”A”) and @Service(name=”B”) annotations, respectively. They serve for accessing OSGi services (registered under service names A and B) and for providing them to other SOFA2 components. Furthermore, the component implements the *`SOFAServiceListener`* interface get notified of related service events.

In respect of the OSGi runtime, there are two bundles, the bundle A and B. The bundle A registers a service under the service name *A*. The service is being used by the bundle B.

6 Lightweight Directory Access Protocol

7 `org.osgi.util.tracker.ServiceTracker`

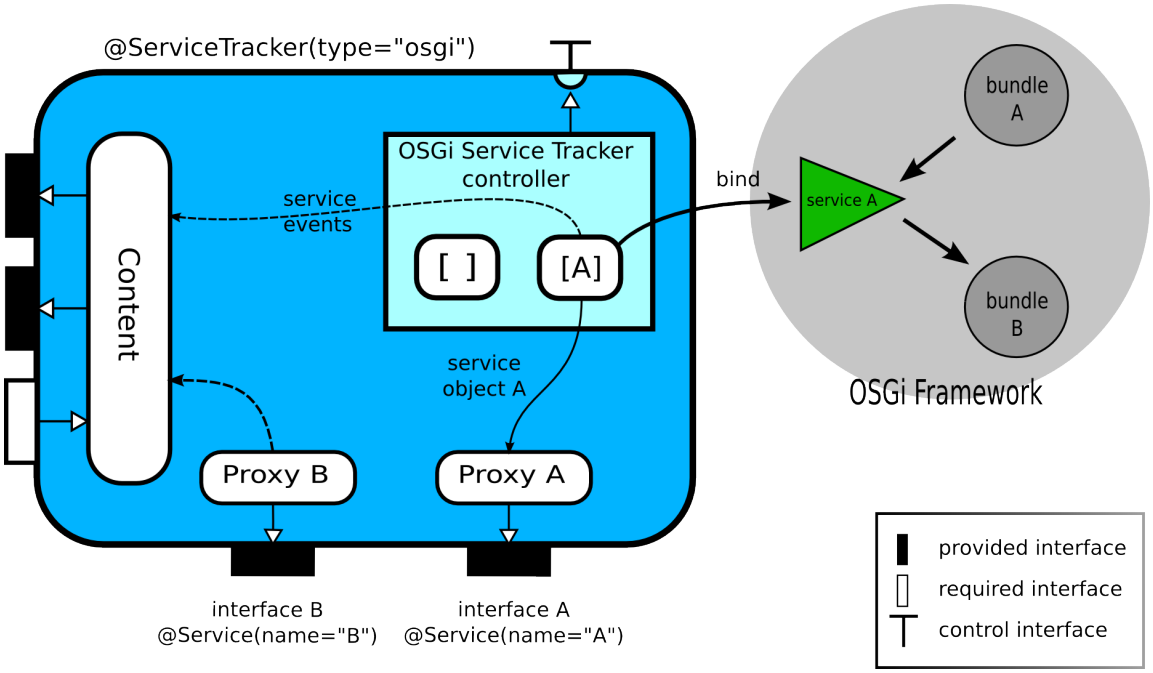


Figure 12: OSGi Service Tracker controller

When the component is started, the SOFA2 runtime calls the controller's *open()* method and the underlying `ServiceTracker` instances start tracking services that are registered under the service names A and B.

As the service A is already registered in the OSGi Service Registry, the corresponding tracker is notified. It binds the service and sets up the *Service Proxy A* with the *service object A*. The component is notified of the event, by calling the *bind()* method.

Method invocations on the interface A are handled by the *Service Proxy A*. When a method is invoked, the proxy inspects the *service object A*, it finds the matching service method and invokes it. The return value is returned as the result of the method invocation.

As there are no services registered under the service name B, the *Service Proxy B* is not set up. The method invocations on the interface B will fail with an unchecked exceptions, see below.

3.5.1 Service Proxy

The method invocations on a @Service annotated interface are handled by the *Service Proxy*. It acts as a mediator between the interface and a matching service. When a method is invoked, the proxy inspects the *service object* (if the service is available), finds the matching service method, invokes it, and returns the return value as the result of the method invocation. Furthermore, the proxy deals with method types (method parameters, return values and exceptions) when invoking service methods. It makes a heavy use of

the Java Reflection API, for the purpose.

In order to support component versioning and to prevent class name clashes, the SOFA2 uses byte-code manipulation when uploading classes to the SOFA2 repository [4]. The classes that are stored in the repository are renamed. Therefore the class names of non-primitive[8] method types used by a service (in the OSGi runtime) differ from the types used by the corresponding interface (in the SOFA2 runtime). These types cannot be used directly and a special care must be taken when dealing with them.

When an interface method is invoked, the proxy inspects the method *parameters* and makes a copy of the non-primitive ones so that the copies are instances of the service types. The service method is invoked with the new set of parameters. On the other hand, the *return value* and *exceptions* are dealt with right after the service method is invoked, to make sure they match the types used by the interface.

The replication process (i.e. making a copy) involves creating a new instance of the resulting type and copying the member attributes (public, private and protected) from the original. Similarly for Java arrays, it creates a new array and copies the elements one-by-one. Every type being copied has to provide a public no-argument constructor in order to create a new instance of the type.

The whole process of handling a method invocation and cloning the method types is depicted in Figure 13. It shows handling a method invocation on the interface A, which is annotated with the @Service(name="A") annotation.

Assuming that the *service A* is registered in the OSGi Service Registry, the OSGi Service Tracker controller has set up the *Service Proxy A* with the *service object A*.

Let the method signature be as follows:

```
Complex add(Complex a, Complex b);
```

The method adds up two complex numbers and returns the result (as a complex number). The *Complex* type is a simple POJO[9] that represents a complex number. In the SOFA2 repository, the type is stored as the $Complex_{SOFA2}$ class, for simplicity.

When the method *add()* is invoked (on the interface A), the invocation is handled by the *Service Proxy A*. It replicates the method parameters (instances of $Complex_{SOFA2}$) so that the new parameters are instances of the *Complex* type. Further, it invokes the matching service method *add()* on the s*ervice object A* using the new set of parameters. As the return value is an instance of the *Complex* type, it needs to be replicated, as well. The proxy creates a new instance of the $Complex_{SOFA2}$ type and copies all the member attributes from the return value. The newly created instance is returned as the result of the method invocation.

8 Not Java primitive types (wrapper classes) and Strings
9 Plain Old Java Object

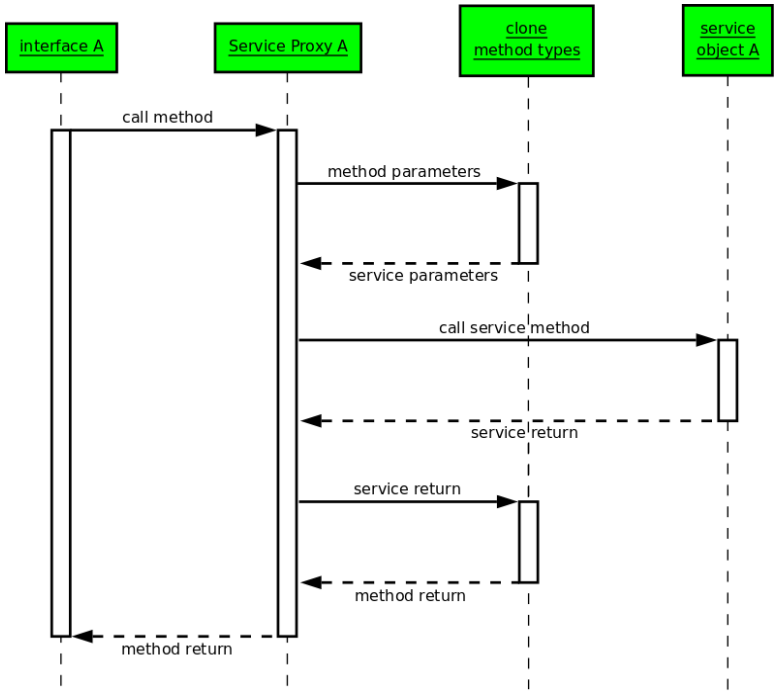


Figure 13: Service Proxy handling a method invocation

The *Service Proxy* acts as a mediator between the interface and a set of matching services. The services behind the proxy may come and go dynamically. It is up to the corresponding *Service Tracker* to keep track of them (e.g. bind a service when available) and set up the proxy accordingly.

When no matching service is available the proxy gets *disabled* and method invocations on the corresponding interface will fail with an unchecked exception, `ServiceUnavailableException`[10]. However the *timeout* attribute (of the @Service annotation) may be used to configure the time (in milliseconds) to wait up for a service to become available before failing the invocation.

The proxy gets *enabled* when there's at least one service matching the interface settings. If there are multiple services, registered under the service name (and matching the filter expression, optionally), the service with the highest *ranking* is used. The proxy is set up with the *service object* and it serves as the target of method invocations.

There's yet another way to handle unavailability of services. A primitive component can implement the @Service annotated interface by itself. This way a method invocation on the interface will not fail (when no matching service is available), but is handled by the component. This feature can be useful when testing SOFA2 applications that make

10 `org.objectweb.dsrg.sofa.osgi.ServiceUnavailableException`

use of OSGi services, since the services may not be available at the time of testing.

3.5.2 ServiceListener interface

Service events are used by the OSGi Framework to report registrations, unregistrations and property changes of services. The controller makes use of service events (through *Service Trackers*) to keep track of services and to update proxies accordingly. So that they reflect the availability (or otherwise) of services.

A primitive SOFA2 component can explicitly track related service events by implementing `SOFAServiceListener`[11] interface, see Figure 14. The component is notified whenever one of the *Service Proxies* changes its state, e.g a matching service becomes available and is bound (or otherwise).

```
public interface SOFAServiceListener {

    void bind(String service, ServiceReference reference);
    void rebind(String service, ServiceReference reference,
                                ServiceReference stale);
    void unbind(String service, ServiceReference reference);
    void modified(String service, ServiceReference reference);
}
```

Figure 14: SOFAServiceListener interface

The method *bind()* is called, when a service becomes available and the corresponding *Service Proxy* is set up using the *service object*. The *service name* (as registered in the OSGi Service Registry) and *service reference* are passed as arguments of the call. The *service reference* encapsulates the service properties and other meta-information.

When a service (used by some *Service Proxy*) is unregistered and there's no suitable replacement for the service. The corresponding proxy gets *disabled* and the component is notified by calling the *unbind()* method. It lets the component know that the service is not available anymore and calls on the interface would fail.

On the other hand, if one of the services is unregistered and there are other services that can replace it. Then, the service with the highest *ranking* is bound and the corresponding proxy is set up. The component is notified by calling *rebind()* method. The service reference of the service being unregistered, as well as, the new one are passed as arguments of the call. So the component can keep track of stale references. The *rebind()* method informs the component that the service implementation has changed even though the service availability has not.

The *modified()* method is called when one of the services have changed the properties.

11 `org.objectweb.dsrg.sofa.osgi.SOFAServiceListener`

3.6 OSGi Service Publisher controller

The OSGi Service Publisher controller provides the implementation for the Service Publisher aspect. It handles the management of OSGi service (i.e. registering and unregistering services). A component is provided with the controller, when its frame is annotated with *@ServicePublisher(type="osgi")* annotation.

The controller implements the Service Publisher control interface (Fig. 11) to direct the controller's life-cycle. The methods *publish()* and *unpublish()* are called by the SOFA2 runtime when the component starts and stops, respectively. The set of interfaces to be published as OSGi services is determined by @Publish annotations used with frame's interfaces.

The @Publish annotation may carry a map of key/value properties that are to be used when registering the service in the OSGi Service Registry. All the keys and values must be Java Strings; except for the *service.ranking* property that has an integer value. The *service.ranking* is used by the Framework when querying the Service Registry – the service with the highest ranking is returned when there are multiple matching services.

When the controller is started (the *publish()* method was called), it registers the @Publish annotated interfaces as OSGi services. The service implementation is provided by the corresponding SOFA2 component. On the other hand, the controller is unregisters all the services that have been registered when stopped (the *unpublish()* method is called).

Each @Publish annotated interface has a corresponding *Proxy Bundle (*Sect. 3.6.1). The Proxy bundle resides in the OSGi Framework and is used by the controller for registering and unregistering the interface as OSGi service. It consists of the interface type's (used by the interface) classes and a manifest file, which specifies the bundle. Furthermore, the Proxy bundle exports the classes of the corresponding interface type so that other bundles in the Framework can import them and use the service.

When the controller is initialized, it starts all the Proxy bundles that correspond to the @Publish annotated interfaces and the bundles register themselves within the SOFA2 runtime via the `RegistrationListener` interface (Fig. 15). The registrations are then used by the controller to register/unregister corresponding OSGi services.

The *register()* method is called (by the controller) to register an OSGi service. The service implementation and *properties* are passed as arguments of the call.

To unregister services, the controller calls the *unregister()* method.

```
public interface RegistrationListener {
    boolean register(Object impl,
                     Dictionary properties);
    boolean unregister(Object impl);
}
```

Figure 15: RegistrationListener interface

Every OSGi service registered by the controller is backed up by a *Service Proxy*. It handles method invocations on the service (in OSGi) and acts as a mediator between the service and the service implementation that is provided by the corresponding SOFA2 component. In a nutshell, method invocations are handled the same way as on @Service annotated interfaces – only the other way round, see Sect 3.5.1.

The overview of the OSGi Service Publisher controller is depicted in Figure 16.

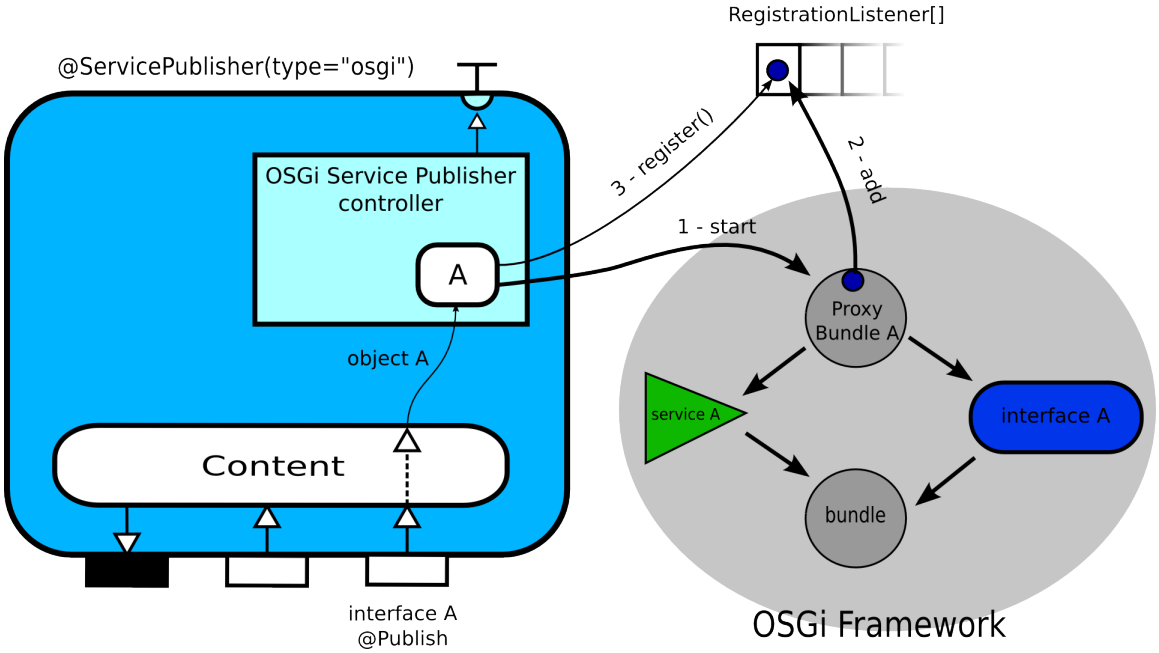


Figure 16: OSGi Service Publisher controller

It shows a primitive SOFA2 component. The component has several business interfaces and is annotated with @ServicePublisher(type=”osgi”) annotation. The required interface A is annotated with the @Publish annotation and serves for publishing services. Services are registered in the OSGi registry under the service name A.

In respect of the OSGi runtime, there are two bundles. The Proxy Bundle A – corresponding to the interface A. And a consumer bundle that is tracking services being registered under the service name A. Note, the interface A being shared between the bundles. The Proxy bundle exports classes, while the consumer bundle imports them. The consumer makes use of classes when invoking the service A.

The Proxy bundle A gets started when the controller is initialized. It registers itself within SOFA2 runtime through the `RegistrationListener` interface (Fig. 15). The registration is looked up by the controller (when *publish()* method is called) and used for registering (by calling *register()* method) the service A. The service implementation is provided by the component that is bound to the interface A.

When the service A is registered, the consumer bundle gets notified and binds the service. When the service is invoked, the method invocation is handled by the corresponding Service Proxy A. It invokes the matching interface method (using the component that is bound to the interface A). In fact, the interface A got registered in the OSGi registry as the service A.

3.6.1 Proxy bundle

Proxy bundles have been introduced in order to allow the OSGi Service Publisher controller to register and unregister OSGi services. Each @Publish annotated interface has a corresponding Proxy bundle. The Proxy bundle has to be installed in the OSGi Framework prior to registering the service (i.e. starting the corresponding SOFA2 component). For information about creating Proxy bundles, see Sect. 3.7.1.

A Proxy bundle consists of interface type's classes and a manifest file, which specifies the bundle. The bundle's implementation (activator) is provided by the SOFA2 runtime (i.e. imported). Basically, it implements the `RegistrationListener` interface and registers the bundle within the SOFA2 runtime when started.

An example of a Proxy bundle's manifest file is shown below.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Created-By: SOFA2 cushion exporter
Bundle-SymbolicName: foo.IPublish_vfunbox_E_0_E_0
Bundle-Activator:
     org.objectweb.dsrg.sofa.osgi.ProxyActivator
Export-Package: foo
Import-Package:
     org.osgi.framework;version="1.3.0",
     org.objectweb.dsrg .sofa.osgi
Service-Name: foo.Publish
Proxy-Version: _vfunbox_E_0_E_0
```

The manifest headers used by a Proxy bundle are as follows:

- `Service-Name` – the service name used for registering services

- `Proxy-Version` – the interface type's version
- `Export-Package` – exported packages (i.e. the interface type's classes)
- `Bundle-SymbolicName` – unique name of the Proxy bundle

3.7 Cushion

Cushion is a command line tool for developing SOFA2 components. It is not included in the SOFA2 distribution but delivered as a separate archive. The tool can be downloaded from the SOFA2 web site, see [19].

It provides a number of useful commands that are utilized throughout the development life-cycle. Namely, there are commands to support the *component development* (creating new frames, architectures and interface types), *application assembly* and *deployment*. The full list of commands can be found in the usage documentation, see [20].

The general usage of *cushion* is as follows:

`cushion cmd arg1 arg2 ...`

The *cmd* argument specifies the command to be executed; while the rest of the arguments are passed to the command when being executed.

In order to provide support for developing OSGi-enabled applications, *cushion* has been extended in several ways. First, the current *cushion* commands have been extended to support the newly introduced annotations, see Sect. 3.2. Secondly, the *osgi* command has been introduced. It enables to manage and configure the OSGi Framework. Moreover, it is used for importing service interfaces into the SOFA2 repository and for exporting SOFA2 interfaces as bundles (i.e. Proxy bundles).

Apart from the command line interface, there is a GUI[12] tool built over *cushion* [21]. It is based on the Eclipse platform and allows the visual development of SOFA2 applications.

3.7.1 Command osgi

The *osgi* command enables management and configuration of the OSGi Framework. For this purpose, it provides several actions.

The general usage is as follows:

`cushion osgi action arg1 arg2 ...`

The *action* argument specifies the action to be executed. If the argument is not provided, all available actions are printed out. To get a short description of the actions, use `cushion help osgi`.

The Framework configuration that is used for performing the action, is specified in the cushion configuration file.

The actions provided by the *osgi* command are as follows:

12 Graphical User Interface

Action **shell**:

Usage: `cushion osgi shell`

Starts a text-based user interface (or GUI) shell for interacting with the OSGi Framework. The location of bundles providing the shell can be configured via `SHELL_BUNDLE` and `SHELLUI_BUNDLE` variables in the cushion configuration file. The default configuration uses the Felix TUI Shell. However it can be easily reconfigured to provide a GUI shell, instead.

The shell bundles are installed when the Framework is started, and uninstalled afterwards. The *shell* action makes use of the Framework event listener to be notified when the Framework is shutting down.

The Felix TUI shell provides means to interact with the OSGi Framework. It offers many handy commands for managing the Framework. Bundles can be installed/started/stop/uninstalled and updated. Moreover, you can list all the installed bundles and registered services, a bundle's headers and exported packages, etc. To get a description of available commands, type `help` into the shell. The Framework is shut down by running `shutdown` command.

Action **service**:

Usage: `cushion osgi service` *`name`*

- *name* – a fully qualified service name

Prints out a list of methods (in the form of a Java interface) implemented by a service that is registered under the given name. This may be useful for getting details about available services. While, the Java interface can be used for creating interface types, manually. For example, when a component uses only a subset of service methods. The full print of a service interface is created by using *import* action.

Action **services**:

Usage: `cushion osgi services [`*`filter`*`]`

- *filter* – an LDAP filter expression

Prints out available services that match the filter expression. If the filter expression is not provided, it prints out all available services.

The filter expression matches services based on their properties. The syntax of the filter expression is defined in [18].

Action **import**:

Usage: `cushion osgi import service interface`

- *service* – a fully qualified service (interface) name
- *interface* – an interface type

Imports the given service interface into the SOFA2 repository. The resulting *interface type* can be used by SOFA2 components to access OSGi services that are registered under the service name. For more information about using OSGi services, see Sect 4.1.

First, a transitive closure of the service interface is formed. The types occurring in the service interface are observed by a static code analysis, using ASM.

The transitive closure consist of:

- service interface class
- method types (parameters and return types)
- exceptions
- inner classes
- super class
- implemented interfaces

Classes from `java.*` and `org.osgi.*` packages are excluded from the whole process since it is assumed that they belong to the execution environment.

Next, the interface type's signature (i.e. the fully qualified name of the interface) is updated and the transitive closure is used as the *code-bundle*. Finally, the interface type is stored in the SOFA2 repository. The interface type became a self-contained print of the service interface.

Action **export**:

Usage: `cushion osgi export interface`

- *interface* – an interface type

Exports the interface type as a Proxy bundle (see Sect. 3.6) and installs it into the OSGi Framework. The Proxy bundle is used by the OSGi Service Publisher controller for registering/unregistering services.

The command creates the Proxy bundle and stores it in the SOFA2 repository. It consists of the interface type's classes and a manifest file that specifies the bundle. Furthermore, the bundle export the interface classes (the interface and method types) so that other bundles in the Framework can import them.

4 Usage and use cases

This chapter demonstrates using and publishing OSGi services in practice. Further, it presents several examples and step-by-step guides to illustrate how to create and set up a SOFA2 component in order to access and publish OSGi services.

Finally, it shows how to set up a SOFA2 application to achieve seamless remoting for OSGi services.

4.1 Using OSGi services

A SOFA2 component is enabled to access OSGi services by using the *@ServiceTracker(type="osgi")* annotation with its frame. In more details, based on the frame annotation, the SOFA2 runtime applies the Service Tracker aspect and provides the component with the OSGi functionality though the corresponding OSGi controller, see Sect. 3.5. In order to access OSGi services, the component has to be hosted by a deployment dock embedding the OSGi Framework, see Sect. 3.1.

OSGi services are accessed through dedicated business interfaces, both provided and required. The required ones serve for accessing services by the component itself. While, the provided ones are used for providing services to other SOFA2 components that may reside in different deployment docks (Fig. 17).

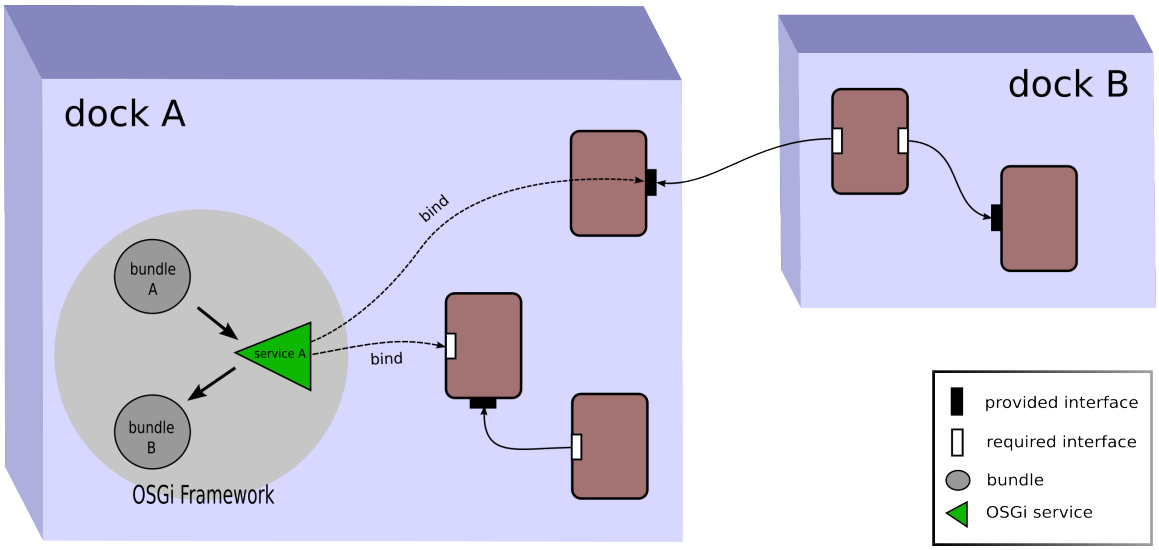


Figure 17: Using OSGi services

The *@Service* annotation is used with interfaces to mark out the ones that serve for accessing OSGi services. The annotation attributes determine the set of target services for each of the interface. The *name* attribute specifies a service name – only services

registered under the given name can be bound to the interface; the optional attribute, *filter*, specifies an LDAP filter expression that is used for further constraining the set of target services based on the service properties.

The example below matches Http services having a "port" property equal to 80.

```
@Service( name="org.osgi.service.http.HttpService",
          filter="(port=80)" )
```

Method invocations on a @Service annotated interface are handled by a corresponding *Service Proxy* (see Sect. 3.5.1). It acts as a mediator between the interface and one of the matching services. If there are multiple services registered under the service name (and optionally matching the filter expression), the one with the highest *ranking* is used. It is up to the corresponding controller to bind/unbind services and set up proxies accordingly.

When no matching service is available, method invocations on the interface will fail with an unchecked exception, `ServiceUnavailableException`[13]. The *timeout* attribute (of the @Service annotation) can be used to configure the time in milliseconds to wait up for a service to become available before failing the invocation.

Every interface has an *interface type*, which specifies the interface methods, signature, method types, etc. It stands to reason that the interface type (used by a @Service annotated interface) has to match (i.e. methods and types) the corresponding service interface. The component developer is provided with the `cushion osgi import` command for importing a service interface into the SOFA2 repository as an interface type, see Sect 3.7.1 for details.

4.1.1 ServiceListener interface

It is possible for a primitive component to explicitly track availability (or otherwise) of related services by implementing `SOFAServiceListener` interface, see Figure 18. The component is notified whenever a *Service Proxy* changes the state, e.g a service becomes available and is bound to one of the interfaces.

The method *bind()* is called, when a service becomes available and the corresponding *Service Proxy* gets set up. The *service name* (as registered in the OSGi Service Registry) and *service reference* are passed as arguments of the call. The *service reference* encapsulates the service properties and other meta-information.

When a service (used by some *Service Proxy*) is unregistered and there's no suitable replacement for the service. The corresponding proxy gets *disabled* and the component is notified by calling the *unbind()* method. It lets the component know that the service is not

13 `org.objectweb.dsrg.sofa.osgi.ServiceUnavailableException`

available anymore and further method invocations on the interface will fail.

```
public interface SOFAServiceListener {

    void bind(String service, ServiceReference reference);
    void rebind(String service, ServiceReference reference,
                                ServiceReference stale);
    void unbind(String service, ServiceReference reference);
    void modified(String service, ServiceReference reference);
}
```

Figure 18: SOFAServiceListener interface

On the other hand, if one of the services is unregistered and there are other services that can replace it. The service with the highest *ranking* is bound and the corresponding proxy is set up. The component is notified by calling *rebind()* method. Both of the service references, the old one and the new one, are passed as arguments. So that the component can keep track of stale references. The *rebind()* method informs the component that the service implementation has changed even though the service availability has not.

The *modified()* method is called when one of the services have changed the properties.

4.1.2 UPnP robot example

The following example demonstrates using OSGi services in practice. The step-by-step guide shows how to create and and set up a SOFA2 component in order to access an OSGi service. The component implementation is left out for simplicity.

The application, shown in Fig. 19, consists of a single SOFA2 component that makes use of an OSGi service to control a sample UPnP[14] robot. The component resides in a deployment dock with an embedded OSGi Framework. The Framework contains the *UPnP driver* bundle that discovers the UPnP robot and makes it available via the *robot service* (`upnp.device.robot`). The service is then used by the SOFA2 component to control the robot.

14 Universal Plug-and-Play

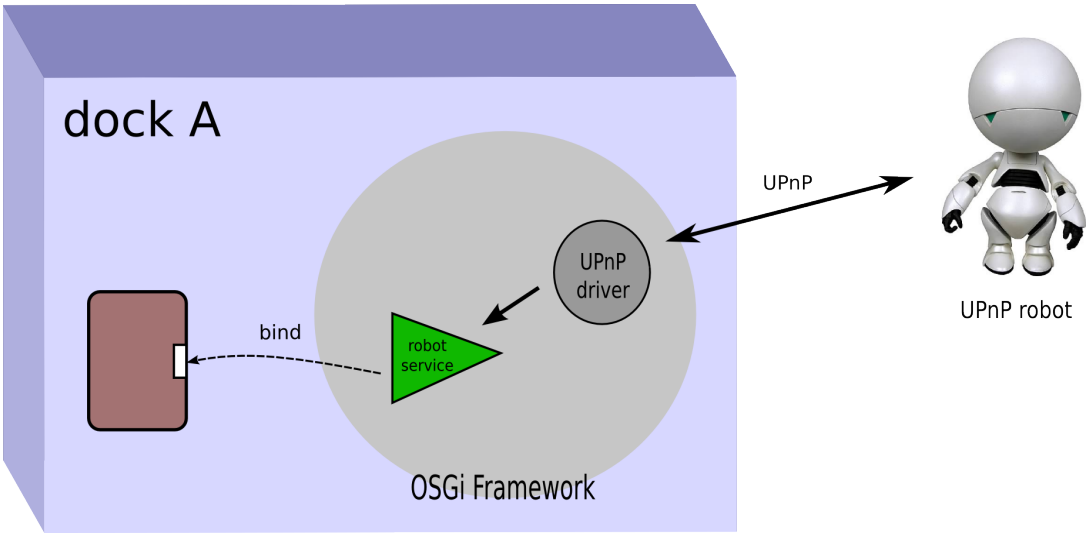


Figure 19: UPnP robot example

Steps

1. Launch the SOFA2 repository, see the SOFA2 usage documentation.
2. Create a new directory, which will be used for developing the application and change to it.
3. First, create an empty interface type to hold the service interface.

```
cushion new interface initial foo.RobotService
```

The command creates a new (empty) interface type and generates an ADL file (`adl.xml`) with the interface type definition. The file is created in the `foo.RobotService` directory.

4. Next, import the service interface into the SOFA2 repository

```
cushion osgi import upnp.device.robot foo.RobotService
```

It creates a self-contained print of the service interface (`upnp.device.robot`) and stores it in the SOFA2 repository using the `foo.RobotService` interface type.

The interface type definition looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<itf-type name="foo.RobotService"
          signature="upnp.device.robot" />
```

The service interface classes are stored in the repository as the code-bundle of the interface type. Note, the signature attribute that is set to the service interface class name.

5. Create the component's frame, `foo.FTester`

   ```
   cushion new frame initial foo.FTester
   ```

   This creates an empty frame in the repository and generates the ADL file that should be filled in. The frame defines a required interface (with the type previously created). The interface will serve for accessing the *robot service* by the component. The ADL file is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="foo.FTester" service-tracker="osgi">
 <requires name="service"
           itf-type="sofatype://foo.RobotService"
  <service name="upnp.device.robot"
           filter="(name=Marvin)" />
 </requires>
</frame>
```

   Note, the *service-tracker* attribute of the `<frame>` tag, it represents the @ServiceTracker annotation used with the frame. The attribute value is used as the *type* attribute of the annotation.

   The @Service annotation is specified with the `<service>` element. The attributes determine the set of target services. As shown in the ADL file above, a suitable service has to be registered under the service name `upnp.device.robot` and the *name* property must equal to *Marvin.*

6. Next, create the component's architecture, `foo.ATester`

   ```
   cushion new architecture initial foo.ATester
   ```

   This creates an architecture for the component and generates the ADL file that should be filled in. The component is primitive thus the architecture is empty. It specifies the frame that the component implements, and the Java class that implements the component. The ADL file is shown below. Though the implementation is left out for simplicity.

```
<?xml version="1.0" encoding="UTF-8"?>
<architecture name="foo.ATester"
              frame="sofatype://foo.FTester"
              impl="..." />
```

6. Commit all changes in the ADL files into the repository

   `cushion commit`

   Note, without any parameters, it commit changes in all working elements.

Now, the component is created and stored in the SOFA2 repository. Next, you need to provide the component's implementation, create the assembly descriptor and deployment plan; setup the runtime environment and finally launch the application.

4.2 Publishing SOFA2 interfaces as OSGi services

A SOFA2 component is enabled to publish OSGi services by using the *@ServicePublisher(type="osgi")* annotation with its frame. In more details, based on the frame annotation, the SOFA2 runtime applies the Service Publisher aspect and provides the component with the OSGi functionality though the corresponding OSGi controller, see Sect. 3.6. In order to publish OSGi services, the component has to be hosted by a deployment dock embedding the OSGi Framework, see Sect. 3.1.

The interfaces that are to be published as OSGi services are marked out with the *@Publish* annotation. The services are registered in the deployment dock (in the OSGi Framework) where the component resides. The service implementation is provided either by the component itself or by some other SOFA2 component, depending on whether the corresponding interface is provided or required one. This allows a SOFA2 component to publish OSGi services in a different deployment dock than the one that hosts the component. An example of publishing SOFA2 interfaces as OSGi services is depicted in Figure 20.

The *@Publish* annotation can carry a map of key/value properties that are used when registering the service in the OSGi Service Registry. All the keys and values are restricted to be Java Strings; except for the *service.ranking* property that has to have an integer value. The *service.ranking* is used by the Framework when querying the Service Registry – the service with the highest ranking is returned when there are multiple matching services.

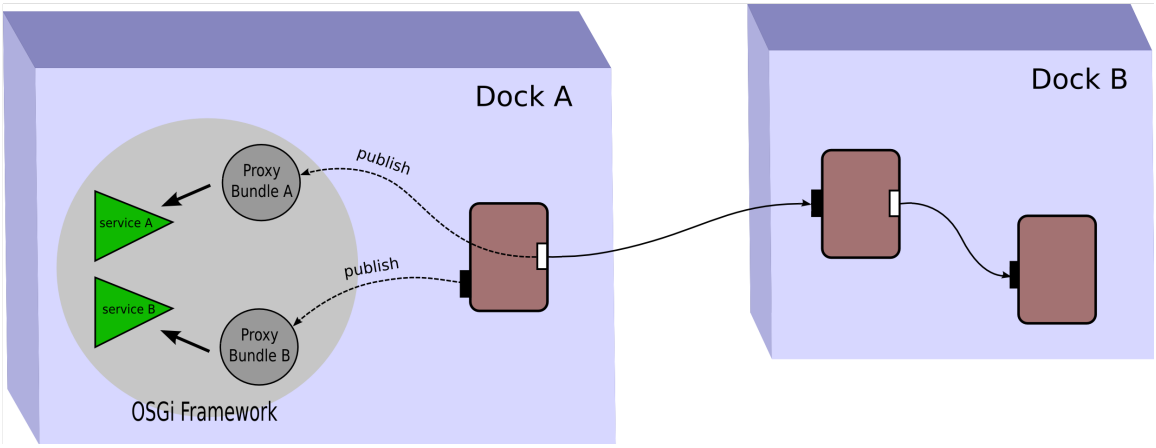


Figure 20: Publishing OSGi services

Each @Publish annotated interface has a corresponding *Proxy Bundle* (Sect. 3.6.1) that resides in the OSGi Framework and is used (by the controller) for publishing the interface as an OSGi service. Furthermore, the Proxy bundle exports the classes of the corresponding interface type so that other bundles in the Framework can import them and use the service.

The Proxy bundles that correspond to the @Publish annotated interfaces of a SOFA2 component, have to be created and installed (by the component developer) in the OSGi Framework, prior to starting the component. The component developer is provided with the `cushion osgi export` command. The command creates a Proxy bundle for an interface type and installs it into the OSGi Framework, see Sect. 3.7.1 for details.

4.2.1 Dictionary service example

The following example demonstrates publishing OSGi services in practice. The step-by-step guide shows how to set up a SOFA2 component in order to publish some of its interfaces as OSGi services. The component implementation is left out for simplicity.

The application, shown in Fig. 21, consists of a single SOFA2 component. It implements a simple dictionary interface and publishes the interface as an OSGi service. The purpose of the example, let the dictionary interface be as follows:

```
public interface DictionaryService {
     String translate(String word);
}
```

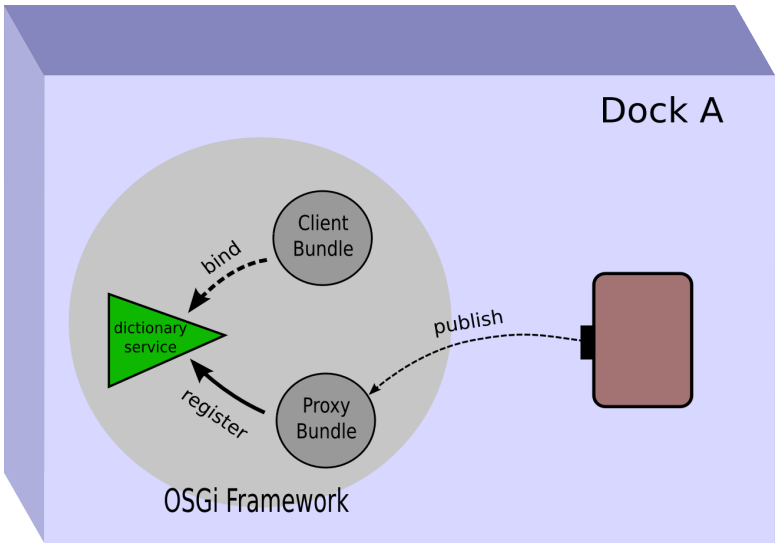


Figure 21: Dictionary service example

Steps

1. Launch the SOFA2 repository, see SOFA2 usage documentation.
2. Create a working directory for the application and change to it.
3. First, create an interface type for the dictionary interface.

   `cushion new interface initial foo.Dictionary`

The command creates an empty interface type and generates an ADL file (`adl.xml`) with the interface type definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<itf-type name="foo.Dictionary"
          signature="foo.sample.DictionaryService" />
```

Note, the signature attribute that specifies the class name of the dictionary interface, see above.

4. Create the component's frame, `foo.FDictionary`

   `cushion new frame initial foo.FDictionary`

This creates an empty frame in the repository and generates the ADL file that should be filled in. The frame defines a provided interface (with the type previously created). The ADL file is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="foo.FDictionary" service-publisher="osgi">
 <provides name="service"
           itf-type="sofatype://foo.Dictionary"
  <publish>
     <property name="lang" type=”English”>
  </publish>
 </provides>
</frame>
```

Note, the *service-publisher* attribute of the `<frame>` tag, it represents the @ServicePublisher annotation. The @Publish annotation is specified with the `<publish>` element. The `<property>` elements specify the service properties.

5. Next, create the component's architecture, `foo.ADictionary`

```
cushion new architecture initial foo.ADictionary
```

This creates an architecture for the component and generates the ADL file that should be filled in. The component is primitive so the architecture is empty. It specifies the frame that the component implements; and the Java class that implements the component. The implementation is left out, for simplicity.

```
<?xml version="1.0" encoding="UTF-8"?>
<architecture name="foo.ADictionary"
              frame="sofatype://foo.FDictionary"
              impl="..." />
```

6. Commit all changes in the ADL files into the repository

```
cushion commit
```

Note, without any parameters, it commit changes in all working elements.

7. Provide the implementation of the interface type and the component

8. At last, create a Proxy bundle for the `foo.Dictionary` interface type

   ```
   cushion osgi export foo.Dictionary
   ```

   This command creates a proxy bundle for the interface type and installs it into the OSGi Framework.

The component is now created, set up and stored in the SOFA2 repository. All that is left is to create the assembly descriptor and the deployment plan; setup the runtime environment and finally launch the application.

4.3 Remoting for OSGi services

OSGi services allow for the communication between bundles within a single OSGi Framework. Considering the OSGi support that has been built in the SOFA2 (i.e. the ability to access and publish OSGi services by SOFA2 components), OSGi services can take advantage of the SOFA2 distributed runtime environment and work in a distributed setting. An example of a SOFA2 application that is set up to provide an OSGi service across deployment docks is shown in Figure 22.

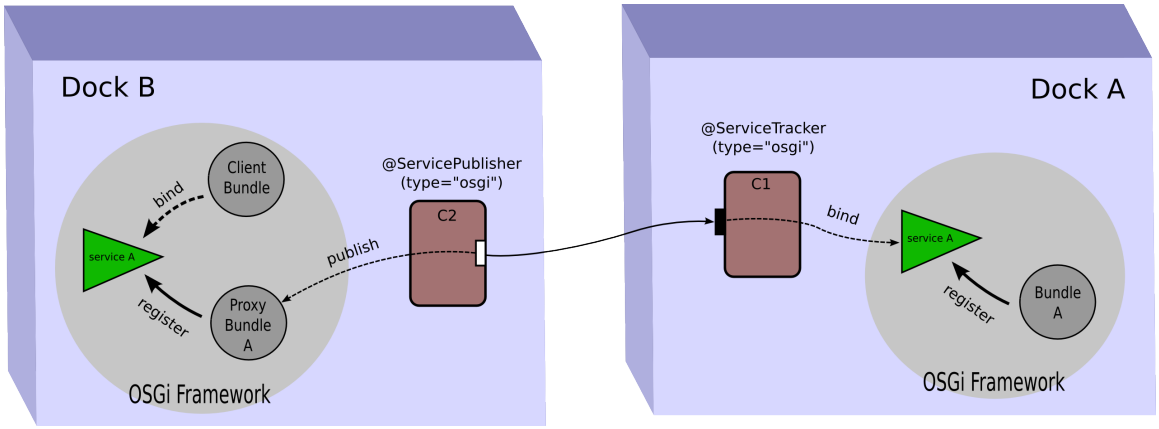


Figure 22: Remoting for OSGi services

The deployment dock A embedding the OSGi Framework hosts a SOFA2 component (C1) with the frame annotated with the @ServiceTracker annotation. The component provides access to the service A (registered by the Bundle A) through the provided interface annotated with the @Service annotation. In the deployment dock B, the component C2 with the @ServicePublisher annotated frame binds the provided interface to its required interface annotated with the @Publish annotation.

When the component C2 starts, it publishes the service A in the local OSGi Framework (i.e. in the deployment dock B) and thus provides the service A to the Client Bundle. The bundle is able to use the service even though it resides in a different OSGi Framework than where the service was originally registered.

5 Related work

This chapter overviews the OSGi support provided by the Spring Framework. Eventually, it discusses differences in the OSGi support provided by the Spring and the SOFA2 component model.

5.1 Spring Framework and OSGi

The Spring Framework is an open source full-stack Java application Framework [22]. It provides a lightweight container and a non-invasive programming model enabled by the use of dependency injection, aspect-oriented programming, POJOs[15], etc. On the other hand, OSGi offers a dynamic application execution environment in which components(bundles) can be installed, updated, or removed at runtime. It has excellent support for modularity and versioning, see Sect. 2.2.

The OSGi support [23] makes it possible to write Spring applications that can be deployed in an OSGi execution environment, and that can take advantage of the services offered by other bundles in the OSGi Framework.

In Spring, the primary unit of modularity is an *application context*, which contains a number of *bean* (objects managed by the application context). Application contexts can be configured in a hierarchy, such that a child context can see beans defined in a parent. The Spring concepts of *exporters* and *factory beans* are used for exporting references to beans outside of the application context, and to inject references that are defined outside of the application context.

The integration with the OSGi happens at the module and service levels. A Spring application context is modeled as an OSGi bundle. While, Spring beans represent OSGi services.

An application context is configured using one or more XML configuration files that are placed in the META-INF/spring folder in the bundle. Optionally, the Spring-Context manifest header can be used for specifying the configuration. The configuration files define beans and related OSGi services.

Spring will automatically create an application context whenever a bundle with a Spring-Context manifest header or resources in the META-INF/spring folder is activated. For this purpose, it uses the org.springframework.osgi.extender bundle, which resides in the OSGi Framework.

OSGi services are represented as Spring beans. An application context's configuration defines the beans that are to be published as OSGi services, and those that are injected with a proxy reference to access OSGi services.

15 Plain Old Java Object

Using OSGi services

The `<osgi:reference>` element is used to define a bean that acts as a proxy to an OSGi service (or set of services). The required attributes are *id* – which defines the name of the local bean; and *interface* – specifies the fully qualified (OSGi) service name. Furthermore, you can specify the following optional attributes:

- *filter* – an OSGi filter expression to constrain the set of target services
- *depends-on* – ensures that the named dependency is instantiated before the bean
- *cardinality* – allows a reference cardinality to be specified (0..1, 1..1, 0..n, 1..n). For references with cardinality 0..n or 1..n, the element resolves to a collection with elements of the interface type.

Spring gives you a constant object reference for the bean that is defined by the `<osgi:reference>` element (either a proxy or Spring-managed collection). The services behind the reference may come and go dynamically.

Method invocations on a service reference may fail at any time with an unchecked ServiceUnavailableException (e.g. the bundle providing the service has been stopped). By specifying the *timeout* attribute in the element, the proxy can be configured to wait up to a given milliseconds for a service to become available before failing.

Furthermore, it is possible to explicitly track the availability (or otherwise) of OSGi services backing a service reference by specifying one or more *listeners* – using the nested `<osgi:listener>` elements.

Exporting Spring beans as OSGi services

A bean is registered as an OSGi service using the `<osgi:service>` element. The *ref* attribute specifies the bean to be registered, and the *interface* attribute defines the interface name that the bean is to be registered under. The element may optionally include one or more nested `<osgi:service-property>` elements to specify the service properties used when registering the service. Furthermore, as an alternative to OSGi *ServiceFactory* interface, the bean may implement Spring's *FactoryBean* interface.

Apart from the ability to access and publish OSGi services, Spring provides support (i.e. defines elements) for some of the standard OSGi services, like Configuration Admin service and Initial Provisioning [16].

Considering the OSGi support provided by the Spring Framework and SOFA2 component model, the set of supported features is comparable. Though there is no support for reference cardinality and OSGi service factories in the SOFA2, yet. The main difference is in the runtime execution environment for components. Spring-enabled

bundles are executed (with other bundles) in the OSGi Framework. While, SOFA2 components are hosted by a deployment dock embedding the OSGi Framework.

For accessing OSGi services, both the SOFA2 and Spring use a Service proxy that mediates the invocation. In addition to that, SOFA2 deals with methods types (parameters, return values and exceptions) when handling a method invocation. Considering the SOFA2 distributed runtime environment, it is easy to set up a SOFA2 application to achieve seamless remoting for OSGi services.

6 Conclusion and future work

In the thesis, a solution providing interoperability between the SOFA2 component system and the OSGi Service Platform has been described.

The proposed solution is based on aspects and annotations. The annotations serve for specifying service-enabled components and interfaces in a declarative way. While, the aspects provide the components with the OSGi functionality – through the corresponding OSGi controllers.

Furthermore, the OSGi support is incorporated in the tool for developing SOFA2 components. It supports for the newly introduced annotations and provides means to interact with the OSGi Framework.

6.1 Goals review

This section reviews the solution with respect to the goals that are outlined in the Sect. 1.4.

(g1) mutual interoperability

OSGi services are accessed through dedicated SOFA2 interfaces, which are marked out with the @Service annotations. The service management issues (e.g. binding/unbinding services) are handled by the corresponding OSGi Service Tracker controller (Sect. 3.5).

The other way round, SOFA2 interfaces that are marked out with the @Publish annotation get published as OSGi services. The service life-cycle is managed by the corresponding OSGi Service Publisher controller (Sect. 3.6).

(g2) seamless integration

The proposed approach uses existing SOFA2 features, like aspects and annotations. It introduces several new annotations (Sect. 3.2). The annotations are used for specifying service-enabled components – the frame annotations; and interfaces – the interface annotations. An obvious advantages of this approach is the possibility of mixing normal business interfaces and service-enabled interfaces within a single SOFA2 component.

Moreover, the control logic (i.e. the OSGi functionality) and the business logic are clearly separated, since the control logic resides in the control part of components.

(g3) handle service dynamics

Every @Service annotated interface is backed up by a Service Proxy (Sect. 3.5.1). It handles method invocations and acts as a mediator between the interface and OSGi services. The services behind the proxy may come and go dynamically. It is up to the

corresponding OSGi controller to keep the proxies up-to-date so that they reflect the availability (or otherwise) of OSGi services.

Method invocations on a @Service annotated interface may fail at any time with an unchecked exception, since the service may not be available at the time of invocation. The *timeout* attribute of the @Service annotation can be used for configuring the time (in milliseconds) to wait up for a service to become available before failing.

Furthermore, a primitive SOFA2 component can explicitly track related service events (e.g. when binding/unbinding services) by implementing the service event listener interface, see Sect. 3.5.2.

(g4) general approach

The outlined approach (i.e. based on aspects and annotations) is general and can be easily reused for integrating other SOA-based systems. The controller that actually implements the service-related aspect (see Sect. 3.3 and 3.4) is determined by the *type* attribute of the frame annotation, which is used with the component. The frame *properties* may carry additional configuration for the related controller. For example, the URL of the service registry.

6.2 Future work

The solution doesn't employ some of the SOFA2 features (e.g. interface cardinality) that are yet to be implemented. The future work might include support for such features. For example, the interface cardinality can utilized for accessing services when there are multiple services registered under the same name.

Another possible improvement might be to replace the service event listener interface (Sect. 3.5.2) with a set of annotations, e.g. the @bind, @unbind, @rebind and @modified annotations.

The OSGi support need to be incorporated in the GUI tool for the visual development of SOFA2 applications [21]. It should support the newly introduced annotations and the *osgi* command. Furthermore, it may offer handy features, like checking the validity of annotation attributes (e.g. the filter expression).

7 References

[1] Szyperski C., Murer S, Gruntz D.: Component Software: Beyond Object-Oriented Programming, 2nd edition, 2002

[2] Sun Microsystems, JSR 220: Enterprise JavaBeans 3.0, May 2006 http://jcp.org/en/jsr/detail?id=220

[3] Object Management Group, Corba Component Model (CCM) V4.0, Jun 2006 http://www.omg.org/technology/documents/formal/components.htm

[4] Bures T., Hnetynka P., Plasil F.: Runtime Concepts of Hierarchical Software Components, International Journal of Computer & Information Science, Vol. 8., Sep 2007

[5] Fractal component model, http://fractal.objectweb.org/

[6] Krafzig D., Banke K., Slama D.: Enterprise SOA: Service-Oriented Architecture Best Practices, 2004

[7] W3C, Web services, http://www.w3.org/2002/ws/

[8] The OSGi Alliance, OSGi Service Platform Core Spec. R4, Version 4.1, Apr 2007 http://www.osgi.org/Specifications/

[9] Sun Microsystems, JSR 208: Java Business Integration (JBI), Aug 2005 http://jcp.org/en/jsr/detail?id=208

[10] Pei Breivold H., Larsson M.: Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles, 33rd EUROMICRO Conference on Software Engineering and Advanced Applications, 2007

[11] Distributed Systems Research Group, Charles University in Prague http://dsrg.mff.cuni.cz/

[12] Hnetynka P., Plasil F.: Distributed Versioning Model for MOF, published by Computer Science Press, ISBN 0-9544145-3-5, pp. 489-494, Jan 2004

[13] Bures T., Hnetynka P., Plasil F..: SOFA2 metamodel, Tech. Report No. 2005/11, Department of Software Engineering, Charles University, Dec 2005

[14] Galik O., Bures T.: Generating Connectors for Heterogeneous Deployment, ACM Press, New York, NY, ISBN 1-59593-204-4, pp. 54-61., Sep 2005

[15] Bures T., Mencl V.: Microcomponent-Based Component Controllers: A Foundation for Component Aspects, in Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005), pp. 729-738, Dec 2005

[16] The OSGi Alliance, OSGi Service Platform Compendium R4, Version 4.1, Apr 2007 http://www.osgi.org/Specifications/

[17] Oaks S.: Java Security, O'Reilly, 2nd edition, 2001

[18] RFC 2254: The String Representation of LDAP Search Filters http://www.faqs.org/rfcs/rfc2254.html

[19] SOFA2 component model, http://sofa.objectweb.org/

[20] Cushion usage documentation, http://sofa.objectweb.org/docs/cushion.html

[21] Pivoluska M.: Visual development of hierarchical components, Master thesis, Charles University in Prague, June 2008

[22] Spring Framework, http://www.springframework.org/

[23] Spring OSGi Specification v0.7, http://www.springframework.org/osgi/specification

[24] Apache Felix, http://felix.apache.org/

Appendix A: Content of the CD

The content of the CD is as follows:

- Directory **bin –** launching scripts for starting the SOFA2 runtime and attached examples
- Directory **cushion –** Cushion distribution
  - **cushion/bin** – launching scripts
  - **cushion/conf** – configuration files
  - **cushion/jdoc** – JavaDoc documentation
- Directory **sofa** – SOFA2 distribution
  - **sofa/bin** – launching scripts
  - **sofa/bundles** – example OSGi bundles
  - **sofa/conf** – configuration files
  - **sofa/jdoc** – JavaDoc documentation
  - **sofa/_repdir** – repository data
- Directory **src** – source files
  - **src/sofa** – SOFA2 source files
  - **src/cushion** – Cushion source files
- Directory **thesis** – electronic version of the thesis in the PDF format
- Directory **workspace** – example SOFA2 applications
  - **workspace/bundles –** source files for example bundles
  - **workspace/serviceTester** – example of using OSGi services
  - **workspace/servicePublisher** – example of publishing OSGi services
- File **README –** content of the CD and a guide for launching the examples

Appendix B: OSGi deployment dock

The `sofa-dockosgi(.sh|.bat)` script launches a deployment dock with an embedded OSGi Framework (i.e. the Apache Felix).

There are two variants of the script:

- `sofa-dockosgi.sh` for UNIX-like systems
- `sofa-dockosgi.bat` for Windows systems

The general usage is as follows:

```
sofa-dockosgi(.sh|.bat) name
```

The dock name is passed as a parameter when launching the script. The Felix configuration is specified in the SOFA2 configuration file – `_setenv(.sh|.bat)`. The `FELIXCONF` variable defines the location of the Felix configuration file. While, the `FELIXCACHE` variable is used for setting up the bundle cache, see the Felix usage documentation [24].

The `sofa-caps(.sh|.bat)` script is used to print out capabilities of deployment docks (e.g. whether the dock runs an embedded instance of the OSGi Framework. The first parameter specifies the name of the deployment dock. Without parameters, it prints out capabilities of all running deployment docks.