

A Comparative Analysis of State-of-the-Art Component Frameworks for the Java Programming Language

Andreas Pieber, Jakob Spoerk

September 25, 2008

Abstract

This document is about the state-of-the-art in the field of component frameworks and standards for component frameworks for the Java programming language. At first, the terms component and component framework are defined. Technologies, concepts and patterns related to the field of component based programming are presented and a way to categorize component frameworks is developed. Following, five major component frameworks and standards for Java (OSGi Service Platform (OSGi), Spring, Java Platform, Enterprise Edition (J2EE) with Enterprise JavaBeans (EJB), Service Component Architecture (SCA) and Java Business Integration (JBI)) are presented and discussed. Furthermore, they are characterized and analyzed based on the criteria defined in the first chapter. In addition, the component frameworks and standards are compared to find ways how they can be combined, which frameworks have the same area of application and are therefore competitors and where the borders of combining different state-of-the-art component frameworks are.

Table of Contents

1	Introduction	5
2	Component based Software Engineering	6
2.1	Definition of Software Components	6
2.2	Definition of Component Frameworks	8
2.3	Motivation for component-based Software Systems and related concepts, patterns and technologies	9
2.4	Criteria to Categorize Component Frameworks	12
2.4.1	Language Support and Platform Compatibility	12
2.4.2	Learning Curve	12
2.4.3	Security	12
2.4.4	Performance and Stability	13
2.4.5	Life-Cycle Management and "Plugability"	13
2.4.6	Reusability and Meta-Data	13
2.4.7	Adaptability and Control	13
3	State-of-the-Art Component Frameworks and Standards	14
3.1	OSGi Service Platform	14
3.1.1	Architecture	14
3.1.2	Interaction	15
3.1.3	Bundle	16
3.1.4	Security	16
3.1.5	Execution Environment	17
3.1.6	Module	17
3.1.7	Life-Cycle	18
3.1.8	Service	19
3.1.9	Standard Services	20
3.1.10	Restrictions	20
3.1.11	Problems	21
3.1.12	Products	22
3.1.13	Conclusion	22
3.2	Spring	23
3.2.1	Architecture	23
3.2.2	Beans	25
3.2.3	Life-Cycle	28
3.2.4	Extension Points	29
3.2.5	Language	29
3.2.6	Problems	29
3.2.7	Portfolio	30
3.2.8	Products	30
3.2.9	Conclusion	31
3.3	Java Platform, Enterprise Edition (J2EE) / Enterprise JavaBeans (EJB)	31

3.3.1	Architecture of J2EE	31
3.3.2	Enterprise JavaBeans (EJB)	34
3.3.3	Architecture of Enterprise JavaBeans	34
3.3.4	Problems	38
3.3.5	Products	38
3.3.6	Conclusion	38
3.4	Service Component Architecture (SCA)	39
3.4.1	Architecture	40
3.4.2	Components	42
3.4.3	Composites	44
3.4.4	Domains	45
3.4.5	Policy Framework	45
3.4.6	Implementation	46
3.4.7	Problems	47
3.4.8	Products	47
3.4.9	Conclusion	47
3.5	Java Business Integration (JBI)	48
3.5.1	Architecture	49
3.5.2	Component Framework	50
3.5.3	Messaging	53
3.5.4	Management Component(JMX)	55
3.5.5	Integration Standardization Beyond JSR 208	56
3.5.6	Problems	56
3.5.7	Products	57
3.5.8	Conclusion	57
4	Comparison of Component Frameworks	58
4.1	Language Support and Platform Compatibility	58
4.2	Learning Curve	59
4.3	Security	60
4.4	Performance and Stability	60
4.5	Life-Cycle Management and "Plugability"	62
4.6	Reusability and Meta-Data	63
4.7	Adaptability and Control	64
5	Interconnection of different Component Frameworks	66
5.1	Spring – Enterprise JavaBeans	66
5.2	OSGi – Spring	66
5.3	OSGi – Enterprise JavaBeans	67
5.4	Service Component Architecture – Spring	68
5.5	Service Component Architecture – Enterprise JavaBeans	68
5.6	Service Component Architecture – OSGi	69
5.7	Java Business Integration – Enterprise JavaBeans	69
5.8	Java Business Integration – Spring	69

5.9 Java Business Integration – OSGi	70
5.10 Java Business Integration – Service Component Architecture	70
5.11 Further Combinations	71
6 Conclusion	72

1 Introduction

Component based software engineering is today a very important discipline, although as described by Oscar Nierstrasz and Dennis Tsichritzis [1], at present, programming with object-orientated languages is much more important than composition. Composition means combining already existing portions of code to build an application.

This approach allows real reusability of software and enables to sell just components instead of full applications. These points in combination with the rapid changing requirements in todays business are the main reasons why component based applications are very important today. They allow reacting faster and cheaper to changes in enterprise structures and allow building more efficient enterprise information systems.

Applications composed of components require a framework allowing components to communicate and to handle the components in terms of management, control, security, resource sharing and life-cycle. This is the main motivation for this paper. Today, several technologies are available providing some kind of a component framework.

This paper presents the state-of-the-art component frameworks in the Java programming language which is the language in which most of todays enterprise information systems are written. These frameworks are discussed and characterized as well as analyzed for means of using them in combination.

Chapter 2 describes the basics of component based programming and component frameworks. Therefore, the terms component and component framework are discussed as well as motivations for using component based programming and technologies related to this software engineering approach are presented. At last, section 2.4 tries to formalize criteria to characterize different component frameworks in the view of software developers so that they are able to decide for a component framework which allow them to best fulfill the requirements.

Chapter 3 presents state-of-the-art component frameworks for use with the Java programming language. This includes the OSGi Service Platform (see section 3.1), Spring (see section 3.2), Java Platform, Enterprise Edition with the Enterprise JavaBeans Container (see section 3.3), the Service Component Architecture (see section 3.4) and Java Business Integration (see section 3.5).

Chapter 4 compares the presented component frameworks concerning the criteria formalized in section 2.4.

Finally, in chapter 5, means are discussed how the different component frameworks can be combined and which motivation may lead to such a combination.

2 Component based Software Engineering

Before describing state-of-the-art component frameworks for Java and their differences and/or similarities, it is essential to get a clear definition of the term software component and to describe the basics and fundamentals of component frameworks. Furthermore, this chapter discusses possible motivations behind the usage of component based programming and component frameworks as well as concepts and technologies tightly coupled with these topics. Finally, general criteria are developed which shall help to characterize different component frameworks to have a basis for comparing them and for making the decision which framework is most suitable for a given project.

2.1 Definition of Software Components

The term software component is a very abstract term. For example, Bruneton et al. [2] defines software components as "*[...] units of software configuration, units of dynamic system configuration, units of development, units of distribution and mobility [...]*".

A more concrete definition of software components is given by Nierstrasz and Tsichritzis [1], saying that a component is a "*static abstraction with plugs*". This definition includes three attributes of software components.

- **Static:** A software component is independent of the applications in which it has been already used, because it can be stored in a software base.
- **Abstraction:** A software component encapsulates its software with some kind of boundary. This additional abstraction layer enables the component to hide its inner structure from other components.
- **With Plugs:** A software component has well-defined interfaces that can be used to interact or communicate with it and that it uses itself to interact with other software components.

Therefore, a software component can be represented visually as a single and mobile entity where plugs can be bound to other entities [1] (see figure 1).

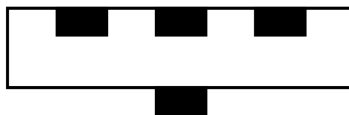


Fig. 1: Simple graphical representation of a software component: A single and mobile entity with "plugs" where other entities can bind to the component or where it can bind itself to other components taken from [1].

To get the right picture of components, Czarnecki and Eisenecker [3] explains component based programming with the example of car manufacturing at automated assembly lines. Software consist out of different parts all having their own tasks and which work together to build the system. Using the picture of a car, there is for example a body, an engine, a transmission, different styles of seats or optional parts like an air condition. Between these parts, there are constraints which can be compared to the plugs used in the definition of Nierstrasz and Tsichritzis [1]. A car body needs a transmission, an engine and seats, but there can be different types of all these components. Every component needs a basic

interface describing its type, but there can be different implementations of the same component. For example, it is possible to have simple gray seats or yellow seats with a heating system. Therefore developing a software system by component based programming or just composition is not the development of a single application, but the development of whole families of applications [3]. This is somewhere similar to general object orientated programming, where there can be different classes sharing the same interface.

Czarnecki and Eisenecker [3] describe four different types of constraints between components. There are plugs where a component has to be connected (mandatory features) whereas there are pluggs where this is optional (optional feature). Another difference between plugs can be if only one component is allowed to be connected (alternative feature) or if it is possible that any number of components is connected to a single plug. For example, a system can have only one database component (one out of several implementations) whereas it can have any number of graphical user interface at a time to enable the user to switch between different representations.

A third definition of the term software component is given by Wolfgang Weck [4]. He defines software components as independently extensible software systems. An extensible system is a system consisting out of different parts that are only loaded and started when required. If a new part is needed, it can be loaded, or if not yet available, retrieved over a network. Independently means that parts can be developed by any company and is although able to be integrated (see section 2.2).

The main goal of using components is that if requirements change, only the part affected directly by the changing requirements has to be modified while the rest of the application can stay untouched. For example, if a customer requires a modification of a system so that the graphical user interface fulfills his corporate design, only that part of the system shall be changed while the rest of the system has not to be reinterpreted or recompiled.

Components and objects are in some aspects very similar entities. They encapsulate data, are accessed through a well-defined interface, are considered to improve the reuse of software and to alleviate the software evolution phase and are abstractions of real world entities [5]. But there are several facts that explain why objects and components are totally different concepts as shown in table 2.1.

Tab. 1: Differences between objects and components [1, 5].

Objects	Components
Abstraction of real world entities	Abstraction of services of real-world entities
Mathematical modeling approach to software	Engineering approach to software
Reusability through inheritance	Reusability without modification
Dynamic entities that may change behavior or state	Static entities
Describe the problem domain (they are a partition of the state space)	Describe the functionality domain (they are a partition of the service space)
Using an object is using one specific object	Using a component provided service is using one specific service provided by any component

Although there are differences between objects and components, it is possible that a component only consists of a single object described in a single class.

There is also a big difference between software libraries and software components. As Reid et al. [6]

stated, software libraries provide services to unknown users, while software components do the same and additionally also consume services from unknown suppliers. Because of this difference, Reid et al. [6] declares the key of efficient software components is the balance between general components that can be used in as many situations as possible while keeping the interface as simple as possible to help other developers to use the component.

Nierstrasz and Tsichritzis [1] defines software composition as *"[...] the process of constructing applications by interconnecting software components through their plugs."* Therefore, Nierstrasz et al. [7] distinguishes between two kinds of development areas in component based software engineering.

- **Application Engineering:** This area analysis the domain and develops reusable software components for the domain that are documented as "Generic application frames" (GAF).
- **Application Development:** This area builds applications for given requirements by just composing prebuilt components of the application engineering area.

Application engineering is therefore a field where highly trained personal is required which have knowledge about the domain as well as about the technologies to build software components. On the other side, application development can be done by less trained personal which only require domain knowledge and which additionally require tools to simplify the interconnection of the components or GAFs. Nierstrasz et al. [7] also mention that investment is only necessary in the field of application engineering while the return of investment is done by the field of application development.

2.2 Definition of Component Frameworks

Weck [4] and the specification of the JBI framework [8] believe, that the purpose of component based software engineering is to create some kind of market for software components, so that companies no longer have to develop whole and because of this often monolithic software systems, but can build just smaller software components that can be used by other companies to build final systems. This would strengthen smaller vendors which do not have the financials to build big and complex applications by their own. On the other hand, the importance of software quality would increase because competition would start at a lower level.

This aim for component based software engineering — the market for software components — is not yet established because of several problems. Beside the problem of missing license systems [4], the biggest problem is to have a standard or guidelines [4] defining how software components have to be developed and structured and how they can be interconnected to work as intended. Component frameworks enable and enforce obedience to such guidelines [4]. They present the basic rules, mechanisms for communication between components as well as resource management and security systems.

Bruneton et al. [2] proposes seven key requirements for a general component-based system model.

1. Encapsulation and Identity

Components have to be encapsulated so that their inner implementation is hidid through well defined interfaces which handle the whole communication and all interactions. Identity on the other hand means that components are well defined so that they can be used easily as intended.

2. Composition

There has to be the possibility to compose different components to form new, higher order components as also stated by Weck [4]. Furthermore, there have to be several ways to compose components as well as the possibility to exchange components without shutting down the whole system.

3. Sharing

There have to be multiple mechanisms to share or multiplex resources between different components. These dependencies on resources have to be explicitly maintainable, enforceable and modifiable at runtime.

4. Life-Cycle

There has to be support for different forms and spans of component life-cycles, for example bootstrapping, deployment, installation, initialization, suspension or termination. This life-cycle support is required for different management processes.

5. Activities

It should be possible to made activities of components explicit and that there are means to manipulate activities happening in the system, even if they involve several components.

6. Control

There has to be the possibility to create controller components that can manipulate the interactions between components, for example by intercepting messages, delaying them or preventing the handling of messages by using pre- and post handlers and therefore alter the interactions in the system. This is one of the reasons why also the point "activities" is one key requirement for general component-based systems.

7. Mobility

Changing modifications of components, introducing new components or terminating components in a general component-based system has to be possible.

Component frameworks are therefore the basis of component-based software systems that declare rules and guidelines so that components can interact with each other, that enables the modification of components at run-time, helps to handle the life-cycle of software components and helps to handle resources that have to be shared between several components.

2.3 Motivation for component-based Software Systems and related concepts, patterns and technologies

There are several reasons why the importance of component-based software system has increased over the last years.

As mentioned in section 2.2 based on Weck [4], there is a need for some kind of a software component market. Today, most software is built only by one company while only smaller parts like libraries are bought from other companies. There are no possibilities to buy components so that a company has only to combine them to build a new product. Having a market where companies can sell just software

components would lead to several advantages. On the one hand, it would be possible for small companies to concentrate on one specific part of a software system and to develop and sell a component encapsulating this part. Therefore, smaller companies would be able to concentrate their limited resources to build competitive software [8]. On the other hand it would be also possible to develop new and innovative software with little developing effort because a lot of components could be bought from other companies. Another advantage would be increasing quality in software [4]. Today, missing quality is a big problem in software engineering. But if some kind of software component market would exist, it would be no longer necessary to develop parts of systems where a company has no competence and therefore quality is very low. Big system developers would try to buy the best components in every field so that the quality of the final systems is as good as possible. Therefore, competition between software vendors would start at the component level and not as today at the level of final products. To be competitive selling just components, these would have to have a high quality standard.

Another motivation for the propagation of component-based software engineering is the problem of rapidly changing requirements. Today, software systems cannot be developed and used over a long time without changes. In economics, everything is changing very fast and therefore, software managing these sectors has also to change and develop. Nierstrasz and Tsichritzis [1] stated to view applications as compositions of reusable and configurable components. This allows handling rapid changing requirements by only unplugging and reconfiguring the affected parts, which made it much more efficient and cheaper to evolve software systems.

In addition to these motivations, there are several concepts, patterns and technologies that have a big importance in the field of component frameworks, which shall be described here.

Service Orientated Architecture (SOA): Service Orientated Architecture describes an in most cases distribute software architecture where functionality is encapsulated into modules called services, which can be accessed over a network. Because the access of such services is independent of its language, platform and implementation through for example the use of web service technologies, business applications build with SOA are interoperable and allow the integration of several applications even if they are from different vendors.

The authors of [9] describe the goal of SOA as *"[...] a world wide mesh of collaborating services, which are published and available for invocation on the Service bus."* Because of that, SOA *"[...] provide a distributed computing infrastructure for both intra- and cross-enterprise application integration and collaboration."* [10]

Therefore, component frameworks can be seen as service orientated architecture because both share the same ideas. While services are a very general concept that can be implemented very flexible, components have more rules because they can not only communicate with each other but can also be managed and controlled by one container. That is one of the reasons why component frameworks and Service Orientated Architecture are often used in the same context.

Enterprise Service Bus (ESB): An Enterprise Service Bus is often described as an architectural approach as well as a middleware product. Its name comes from a physical bus that transfers bits between computers to enable communication.

ESBs are often used in the context of Service Orientated Architecture (and therefore component

frameworks) to provide the medium of communication between services to facilitate SOA [11]. This allows using only one access point in communication and it is no longer necessary that every service has its own protocol to every other service, because the ESB does all required conversions and routing operations [11]. Because of this mediation role, enterprise service buses are standard based to allow a high degree of interoperability.

In few words, services can register to an Enterprise Service Bus, convert their protocols to a standard protocol required by the bus and are then able to access other services through the bus and are themselves accessible.

Inversion of Control (IOC) and Dependency Injection (DI): Inversion of Control (IoC) is often seen as a key characteristic of containers handling components or services. Martin Fowler [12] describes Inversion of Control on the example of the introduction of user interfaces. In former programs, the control of the user interaction was in the hand of the main application. There was a loop that handled every input. Later, when graphical user interfaces occur, a framework managing the graphical elements handled the user input. Pressing for example leads to an event which invokes a method handling the input. Therefore, the control was inverted from the main application to a framework.

In the context of component frameworks, Inversion of Control occurs in the field of dependencies. Instead of instantiating other modules by themselves, modules get these other modules "magically" injected and therefore lose the control over its dependencies, because someone else decides which dependencies are fulfilled by which modules.

Because it is not clear which kind of control is really inverted, Martin Fowler [12] proposes another name for this technology: Dependency Injection (DI). This term describes more accurately what happens: Someone else injects dependencies into a module to decouple modules and increase reusability.

Enterprise Information System (EIS): The term Enterprise Information System (EIS) describes an information system which is specifically built to fulfill the requirements in today's business. Moiss Daniel Daz Toledano [13] describes information systems as

"[...] the compound of components (or elements) that operate together in order to catch, process, store, and distribute information. This information is generally used for taking decisions, for the co-ordination, the control, and the analysis in an organization."

Therefore, Enterprise Information Systems are systems capable of handling and managing all information important for an enterprise so that required information for making the right decisions in an enterprise are available when needed.

To increase flexibility and efficiency of such EIS, they are today mostly built using component frameworks or other Service Orientated Architectures.

Business-to-Business (B2B): Business-to-Business (B2B) in few words is the term for direct, commercial activities between two businesses. For example, the above mentioned market for software components would be a B2B market because companies would buy component directly from other companies without consumers or governments in between.

Enterprise Application Integration (EAI): Enterprise Application Integration (EAI) is the general term for activities aiming to make different enterprise applications of different vendors interoperable. Service Orientated Architecture or Enterprise Service Busses are technologies allowing Enterprise Application Integration. For integrating enterprise applications, four main patterns are described by the book "Enterprise Integration Patterns" [14]. Shared file, shared database, Remote procedure calls and messaging. Today, messaging is the most used concept as can be seen by SOA or ESBs building on this technology.

Java Naming and Directory Interface (JNDI): The Java Naming and Directory Interface is part of the Java Platform and allows to access multiple naming and directory services with a common interface. This allows to store and retrieve named Java objects of any type or to associate objects with attributes to simplify the search for them [15]. This is base functionality of several Java based component frameworks because it allows the retrieval of objects by name and not by reference and this increases decoupling of objects.

2.4 Criterias to Categorize Component Frameworks

This chapter tries to find criteria to characterize component frameworks from the view of software developers. Therefore, the main intention is to find criteria that will help a developer to decide between different component frameworks based on a given project or requirements. It shall be possible to find the framework most suitable for given specifications without knowing much about the different frameworks in advance.

2.4.1 Language Support and Platform Compatibility

This is a very fundamental criterion for the decision which component framework suits for a specific project. It describes which Platform (Hardware, Operating System, ...) and which programming languages are supported by a framework.

2.4.2 Learning Curve

The learning curve describes how hard it is for a developer to learn how a component framework works and how components for a component framework are developed, structured and connected. For this point it is important if the basic development with a framework can be learned quickly (for example if the project has critical time requirements), how hard it is to use a framework most efficiently and how complex a framework is to master.

2.4.3 Security

Security is a very important aspect in software engineering. This criterion tells how efficient security means of a framework are. Is there a possibility to define access rights to resources or networks? How are these security means defined? Who is allowed to interact with whom? Is the system restrictive or flexible? Is it possible to adopt the system to fulfill special requirements?

2.4.4 Performance and Stability

Performance and Stability are also very important criteria. In contrast to a classical, monolithic application, is there a lost in performance? Are there memory leaks in the framework? Do smaller errors disable a whole system? How resistant is a system based on a framework to components that do not work as intended? What happens if the update of a component fails?

2.4.5 Life-Cycle Management and "Plugability"

Life-Cycle Management is one key requirement for component-based software systems [2]. Are the basic life-cycle steps — bootstrapping, deployment, installation, initialization, suspension and termination — included into the framework? How complex is the process of changing the state of a component? Does a change in component life-cycle require a restart of the system or is it possible to for example to add or remove components at run-time?

2.4.6 Reusability and Meta-Data

Are software components based on a component framework really reusable? Can software components be built as general as possible? Are there means to define meta-data for components so that other developers can use components easily as intended?

2.4.7 Adaptability and Control

Is the framework flexible enough so that it can be adapted to requirements or has it so strict rules that it can happen that requirements cannot be fulfilled because the framework cannot handle these situations? Do the developers have full control and insight of the communication between components?

3 State-of-the-Art Component Frameworks and Standards

The concept of component frameworks was already discussed in section 2.2. This chapter describes several state-of-the-art component frameworks and standards for the Java programming language.

3.1 OSGi Service Platform

The OSGi specification is released by the OSGi Alliance (formerly known as Open Services Gateway Initiative [16]), a non profit organisation founded in 1998. This initiative was founded by Ericsson, Sun and IBM. The number of members has since grown from 15 to about 80 members [17].

“The OSGi specifications provide the platform for universal middleware, establishing a standard service-oriented, component-based environment from which businesses can speed development and better serve customers. The OSGi Service Platform greatly increases the value of a wide range of computers and devices that use the Java platform. In a world increasingly reliant on efficient and easy-to-use personal and business productivity software and services, OSGi technology leverages industries and their networks to easily and securely deliver new capabilities to devices”

as stated by the OSGi business benefits whitepaper [18].

With the OSGi specification, a standard was developed which promises a reliable and standard delivery system. With the OSGi middleware a platform was created enabling a delivery system and creates a market for applications to be delivered anywhere and anytime, easily and securely without interruption of the user. Adoption of the component-based platform also reduces time-to-market and development costs because it enables integration of pre-built and pre-tested modules. Further maintenance costs and aftermarket opportunities are increased because networks are used to dynamically update or deliver services and applications to the field [19].

Originally the OSGi framework targeted the residential internet gateways with home automation applications. However, the standard is also applicable and attractive for other markets. Today the middleware is used by Nokia and Motorola for a next generation of smart phones. The car industry adopted the OSGi standard and make it part of the GST specification which is supported by many car manufacturers as Volkswagen or BMW. OSGi could also be found at desktop environments with Eclipse [20], and the server market as the JOnAS OpenSource Java EE Application Server [21] or the Spring Application Platform [22, 23]. Implementations of the OSGi specification could also be found for very restricted environments with Concierge [24]. Many other possible uses for OSGi could be found as seen in [25, 26, 27].

3.1.1 Architecture

OSGi consists of six components where not all of them are required but all are recommended.

Figure 2 shows how the layers are clustered in OSGi. Bundles are the base components of an OSGi implementation and are required. Bundles are JAR files which extend the MANIFEST file which describes the meta information of JAR packages. The Security layer is based on the Java 2 security but adds a number of constraints and files in some of the blanks that the standard Java leaves open. This service defines the runtime interaction of the bundles as well as the interaction with the Java 2 security layer.

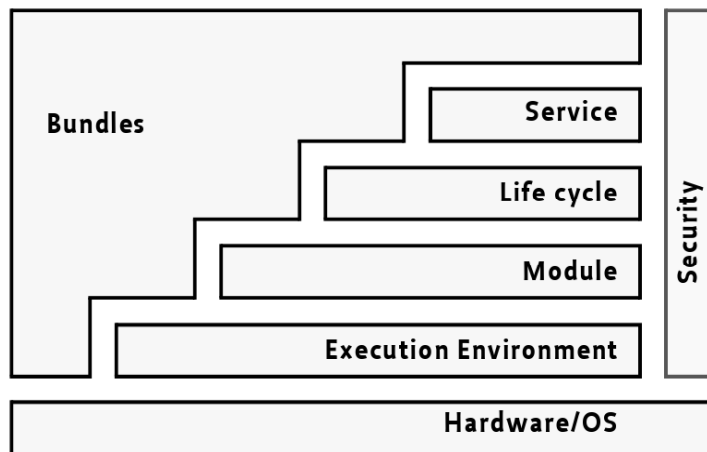


Fig. 2: The architecture of the OSGi component framework taken from [28].

The security service of the OSGi specification is completely optional for the implementation and usage of the framework. The Hardware/OS layer is located directly to the bundle layer because bundle tells the framework how to handle native libraries. Also the security service directly touches the hardware layer, because the Java 2 Security model is quite strongly weaved to the hardware model. The Execution Environment defines the environment needed to run the OSGi implementation. At the moment, only Java is supported. For more information see section 3.1.10. The Module layer defines a modularization model for Java. It addresses some of the shortcomings of the Java deployment model. In this layer strict rules for the sharing and hiding from packages between bundles are defined. This the modularization layer could be used without the Security and Life cycle layers provided by the specification. The Life cycle layer describes a runtime model for bundles. The service defines how bundles are started, stopped, installed, updated and uninstalled. Additionally a comprehensive event API is provided to allow a management bundle to control the operations of the service platform. As seen in figure 2, the Life cycle model requires the Module layer to run. Nevertheless the security layer is optional. The Service Layer sits at the top of the middleware and allows a dynamic, concise and consistent programming model for Java bundle developers simplifying the development and deployment of service bundles by decoupling the service's specification from its implementation. According to this abstract developing model according to interfaces, the selection of a specific implementation, optimized for a specific need or vendor, can thus be deferred to run-time.

In a nutshell it could be concluded that the Module Layer controls how bundles are allowed to work together, the Life Cycle Layer provides the power to manage the bundles in the Module Layer and the Service Layer provides a communication model for the bundles [28].

3.1.2 Interaction

According to the special architecture in OSGi the interactions are biased to runtime of the software. Bundles could register and retrieve available service implementations at run-time through the service registry. Bundles are started and stopped by the Life Cycle Layer which also install and uninstall bundles to the registry which could be loaded by the bundles class loader. Exactly this behaviour makes bundle

extensive at runtime via the installation of updates or new bundles at runtime. The interaction between the layer could also be seen at figure 3 [28].

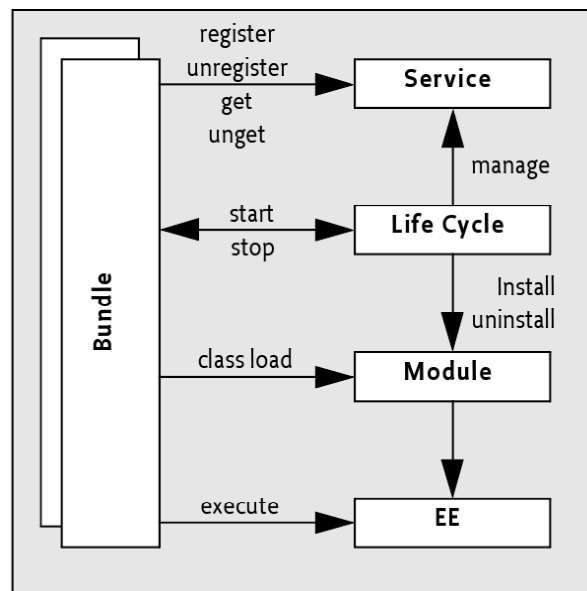


Fig. 3: The layer interaction model of the OSGi component framework taken from [28].

3.1.3 Bundle

The bundle itself can not be described in its functionality quite easy because it has to fulfill different jobs in different layers. However it can be expressed by the definition of a bundle in the official OSGi reference [28]:

"A bundle represents a JAR file that is executed in an OSGi Framework"

Furthermore, every bundle contains a manifest.mf file in a META-INF directory. A JAR Manifest stores information about the contents of the JAR in headers. Some headers are predefined by the JAR Manifest specification but the total set of headers is extendable and the values can be localized. The OSGi Alliance has defined a number of additional Manifest headers to allow the JAR file to be used in an OSGi Service Platform [28, 23].

3.1.4 Security

One of the goals of the OSGi Service Platform is to run applications from a variety of sources under strict control of a management system. A comprehensive security model, present in all parts of the system is therefore a necessity. The OSGi specifications use the following mechanisms:

- Java 2 Code Security
- Minimized bundle content exposure
- Managed communication links between bundles

As the Java 2 Code Security is specified and developed by Sun and not by the OSGi specification, it shall not be further described here. Instead further information can be found at [29].

Java already contains four access modifier used to control access to code. Classes, methods and fields can be made private, package private, protected or public. The OSGi specification adds another security level to this system which makes packages only visible within bundles. Within a bundle the code of this packages could be used as it is public, but outside of the bundle it is private.

Package- and Service-sharing between bundles is a possible attack route for malicious bundles. Therefore the OSGi specification contain a package and service permissions which only allow specific bundles to get access to bundles or services. With this system a quite more powerful security model is added to the java 2 model [23, 28].

3.1.5 Execution Environment

Although the OSGi specification aims to be operation system and programming language independent it is based on a Java Virtual Maschine (JVM). At the time the OSGi Alliance was founded, this was the most logical choice because the Java Runtime Environment provides all the required features for a secure, open, reliable, well supported, mature, rich, portable and platform independent computing environment. However today, there are more languages becoming available on the Java Virtual Machine making the JVM a portable environment and not just a language. Nevertheless a possible alternative may be Microsoft .NET, but there are two reasons the Alliance prefer the JVM. First of all a widely accepted open standard requires an open, multi-vendor platform and secondly the loading concept of the .NET platform could not support all ideas of separated "classspaces" as smoothly as Java does [17, 23, 28].

3.1.6 Module

Within the OSGi specification it's possible to run multiple applications in one JVM. According to this a number of sharing and coordination issues occur which must be addressed. The OSGi framework is the entity for addressing these issues. It has a number of distinct responsibilities which should be explained in the following paragraphs.

- Version incompatibility
- Package export/import

Before these problems could be discovered the private and public bundle policy has to be further discussed. Private bundle means that all classes in a bundle are private and not accessible by other bundles. This model is the easiest solution and also part of other application server standards like Java Platform, Enterprise Edition (J2EE) and Mobile Information Device Profile (MIDP). However as OSGi is also designed for mobile devices which require a very small footprint of the packages class duplication would be very likely in a such a scenario. Therefore the OSGi implementation decides for shared classes which have the advantage sharing libraries for other bundles which allows bundles to be much more flexible and smaller.

According to the decision of a public bundle policy the OSGi framework handles the cases two classes with different version are exported by different bundles. Also incompatibility between the same class of

different bundles can occur. Furthermore the OSGi implementation has to handle the case that bundles export and import classes as shown in figure 4, where export means that packages are made available to other bundles and import means that packages need to be available from other bundles.

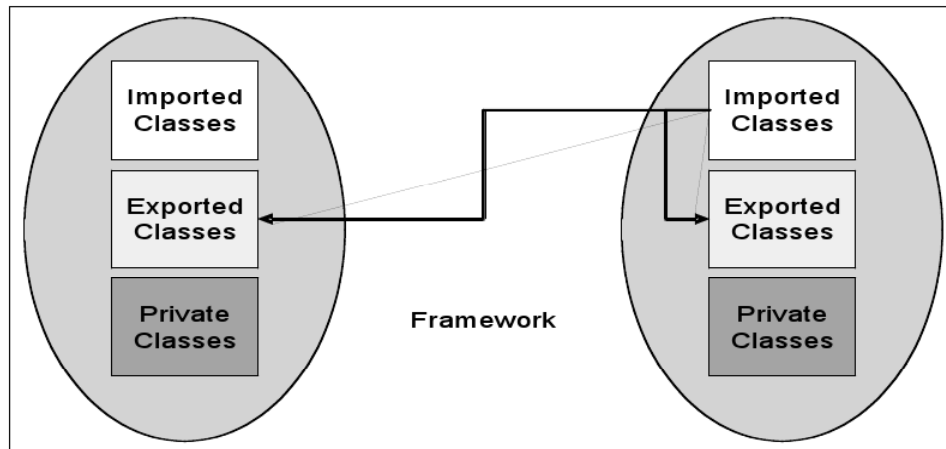


Fig. 4: The class importing scheme of the OSGi component framework taken from [23].

These problems are handled by the Module Layer in the OSGi specification in a rigorously specified and deterministic fashion [23, 28].

3.1.7 Life-Cycle

Although the Life Cycle Layer is optional for the OSGi runtime it is one of the most important parts of the framework. The service is responsible for installing and removing of bundles in (remote) JVMs, which also provides the basis for networked services.

The OSGi specification supports six life-cycle states. The graphical representation of these states could be seen in figure 5.

- **installed:** The bundle has been successfully installed.
- **resolved:** All Java classes that the bundle needs are available. This state indicates that the bundle is either ready to be started or has stopped.
- **starting:** The bundle is being started, the `BundleActivator.start` method will be called, and this method has not yet returned. When the bundle has an activation policy, the bundle will remain in the `STARTING` state until the bundle is activated accordingly to its activation policy.
- **active:** The bundle has been successfully activated and is running; its `Bundle Activator start` method has been called and returned.
- **stopping:** The bundle is being stopped. The `BundleActivator.stop` method has been called but the stop method has not yet returned.
- **uninstalled:** The bundle has been uninstalled. It cannot move into another state.

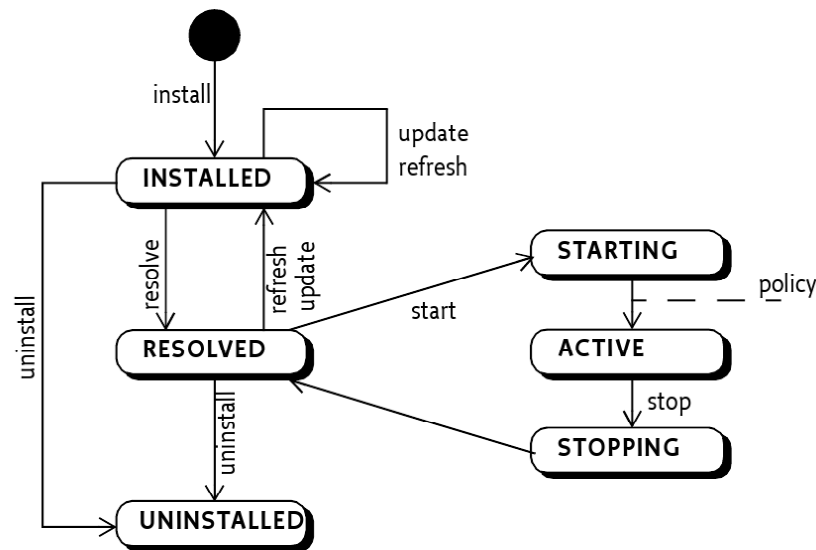


Fig. 5: The life-cycle model of the OSGi component framework taken from [28].

The exact programming model on the Life Cycle Layer would exceed the scope of this paper. Only a short overview should be given about the model because it explains one of the core concepts how to develop against an OSGi implementation.

Another point important before starting with the implementation is mentioned by the OSGi Alliance [23]:

”Access to the bundle installation function is done with an API that is available to every bundle. This may seem odd because it implies that only a bundle can install another bundle. This raises a problem of self-reference: how is the primordial bundle installed? This bootstrap problem is solved with the Initial Provisioning specification or by using command line parameters when starting an OSGi Framework implementation.”

The base object needed by the program is the BundleContext object provided by the OSGi Framework to a bundle when it starts. Via the context, bundles can be installed. By installing them, their BundleContext is returned and could be started. A bundle is Activated via the BundleActivator interface which has a *start(BundleContext)* and *stop(BundleContext)* method. The name of the class inheriting from the interface is specified by a header in the Manifest-File [23, 28].

3.1.8 Service

Although active bundles could use all standard Java mechanisms to implement their functionality, one of the most interesting aspects of the OSGi Service Platform is its dynamic nature. A bundle could suddenly become active and provide functionality may be useful for other parts of the system. Scenarios could be Plug and Play were suddenly a bluetooth phone is available or a PC is turned on. Working on such scenarios with the default Java mechanisms as registering listeners for domain objects could result in hundreds of listeners all having their own idiosyncrasies. Because its also possible to remove parts from the system in an OSGi environment, this solution is not acceptable.

Described by the OSGi specification the Service Registry is responsible for dynamically link different bundles together while tracking their state and dependencies. In a nutshell with the help of a Service Registry, bundles can

- register objects in the Service Registry,
- search the Service Registry for matching objects and
- receive notifications when services become registered or unregistered.

As defined by the OSGi Alliance [23], objects registered in the Service Registry are called services. A service is always registered with an interface name according to which it could be loaded again. The registry itself and the services are implemented at very low cost optimal for JVM running on embedded devices.

This architecture (and the registry) adding the highly dynamically behavior of OSGi and allow a service orientated architecture (SOA). The in-memory registry and its services allow application programmers to develop small and loosely coupled components, which can be adapt to the changing environment in real time. The platform operator uses these small components to compose larger systems. The Service Registry is the glue that binds these components seamlessly together [23, 28].

3.1.9 Standard Services

As described in chapter 2, one problem with component-frameworks are the definition of frameworks. To work this problem the OSGi Alliance specified the Service Compendium [30] which contains 18 services — as example the log and http service are specified. Furthermore general services, as the access of a foreign application to the framework services or the measurement and state specification are described in [30].

Every service is exactly defined via the interfaces and the methods of the services. Also an UML diagram is provided. Examples for better understanding are also presented.

This model allows OSGi application developer to program against standardized interfaces and freely choose the implementor of these afterwards.

3.1.10 Restrictions

When the OSGi platform was designed, the following requirements where included:

- Small devices
- Collaboration model
- Continuously up and running VM
- Life cycle management

The requirements of the collaboration model, the continuously up and running VM and the Life Cycle Management were already described, but the equirement for small devices come at a great cost.

Small devices are usually heavily constrained in persistent storage because they often use a flash memory. Each class file has an overhead of at least 300 to 500 bytes, which adds up surprisingly fast. Use of these classes results in additional overhead. This implies that the number of classes should be kept low. Toward this end, during the early phases of the OSGi specification development, it was decided not to create ad-hoc exceptions or adapter classes. Small devices are also constrained in performance and dynamic memory. Therefore, during the development of the OSGi specification, attempts were made to minimize the requirements for creating superfluous objects that used memory [31]. These requirements are specified by JSR 232 [32] which includes Resource management (JSR 211 [33]), the OSGi specification release 4.1 core [28] and the extension for mobile devices [34].

3.1.11 Problems

Although OSGi brings many advantages to software development and the developers, it also brings some disadvantages. As already mentioned, every bundle has its own classloader. This has the big advantage of having complete control of which classes should be published and which not. Nevertheless, exactly this system brings the most troubles into the platform, because most libraries not created for OSGi, but which are in very broad use, are not aware of this fact. The most popular of them are:

- Spring
- Hibernate
- Jakarta Commons Logging (JCL)

All of those libraries require to have direct access to the context classloader of the bundless so that they work or at least the classes are exported they have to work with. Often, exactly these classes which are required by JCL or Spring are not very likely to be published. This destroys many of the advantages provided by OSGi. With Hibernate, it is exactly the same problem. For example if there is the situation that bundle **A** contains all the Data Access Objects (DAOs), bundle **B** the domain objects or entity objects and bundle **C** is Hibernate. Bundle **A** is aware of bundle **B** and **C**. But when it requires Hibernate (bundle **C**) to load the domain object **X** from bundle **B**, it will fail, because Hibernate (bundle **C**) is neither aware of the contextloader of bundle **A** nor that bundle **B** exists at all. This problem is illustrated in figure 6. Although the example is specifically done for Hibernate, it can be used for any other of those classloading problems too.

The second big problem of the OSGi framework can be found in the classloading section, but not of directly calling classes, as in the last paragraphs. The problem can occur during service calls. This behaviour is not specified by the OSGi specification at its actual version. Also a very simple example can be used to illustrate this problem. Bundle **A** calls a bundle **B**. Bundle **B** requires to load some classes of its own Bundle via the context class loader. Because OSGi does not define the behaviour, this call will fail. Bundle **B** will be running with the contextloader of bundle **A**, which means that all private classes are not visible to the contextloader, whereas the service call will fail.

Beside those major problems really troubling the developer, there are also two minor problems. First of all, the very dynamic way of programming is a problem. Most programmers are not aware of the model and do not create bundles which can handle the situation that a service can be installed or even removed every time. The second minor problem is that the OSGi model itself is quite invasive to the

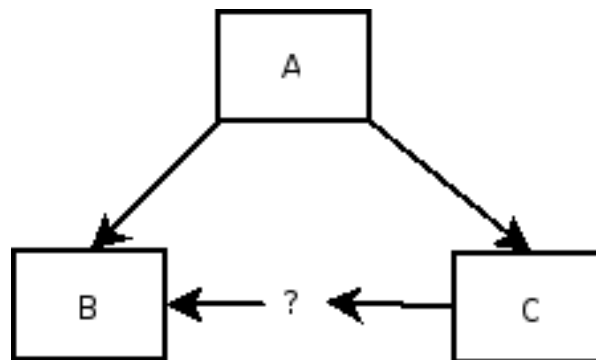


Fig. 6: Illustration of the Hibernate problem occurring in OSGi environments. A bundle **A** loading Hibernate (**C**) has the Problem that Hibernate (**B**) do not know about the contextloader of **A** and other bundles referenced by **A** (**B**).

code. Specific interfaces have to be used to work with OSGi, OSGi specific code is required to call services and to publish them, to start bundles and manage them for example.

Of course, there are solutions for all of these problems through external libraries. Those will be explained at section 5.2 and are out of the scope of this chapter.

3.1.12 Products

The following list contains the most important actual implementations of the OSGi framework.

- Equinox
- Apache Felix
- Knoplerfish
- Concierge

Equinox and Apache Felix (formerly Oscar) are Open Source based implementations of the actual release 4.1. Equinox might be, as the core of eclipse, the most used OSGi implementation, whereas Apache Felix is the best known in projects of the community. Knoplerfish also implements the last release 4.1. But only the backend of the Framework is released to the Open Source. The main features are only available for a commercial license. The actual version of Concierge only implements the release 3 of the OSGi specification, but should be mentioned in this paper because it is explicitly done for mobile devices.

3.1.13 Conclusion

The OSGi specification is one of the mightiest specification available at the moment allowing service orientated architecture at minimum cost. Because the OSGi framework was specified for mobile devices, also desktop developer profit because of the speed. Because the framework also allows to run Webcontainer, Servlets, Ajax and so on, the same packages constructed for a Rich Client Application (RCP) could be used in a webserver. The ability to run bundles also on mobile devices opens a new unified programming model.

Nevertheless, OSGi also has the disadvantage of being "language dependent". Furthermore, the invasive programming model binds the packages to the OSGi framework only allowing them to run in a OSGi runtime.

Although the disadvantages are not to be sneezed at this framework could be seen as a possible future for component frameworks.

3.2 Spring

Spring is an open source project with the main goals [35] to

- provide a consistent and layered architecture to simplify the J2EE platform and to
- promote best-practices in software engineering.

It is developed for the Java programming language and is based on the Book "Expert One-on-One Java EE Design and Development" written by Rod Johnson in 2000 [36]. In June 2003, the framework developed by Rob Johnson was used as the fundament of a community project which released the first version of Spring in March 2004 [36]. Today, Spring is available at version 2.5.5 and is developed by SpringSource, former known as Interface 21, which is also the leading company behind Apache Tomcat [37]. The principles of the Spring community [38] are

- *"J2EE should be easier to use*
- *It is best to program to interfaces, rather than classes. Spring reduces the complexity cost of using interfaces to zero.*
- *JavaBeans offer a great way of configuring applications.*
- *OO design is more important than any implementation technology, such as J2EE.*
- *Checked exceptions are overused in Java. A framework shouldn't force you to catch exceptions you're unlikely to be able to recover from.*
- *Testability is essential, and a framework such as Spring should help make your code easier to test."*

As the official reference [39] stated, Spring is light-weighted but still supports transaction management, persistence, remote access using RMI, a Model-View-Controller (MVC) framework for web applications and a transparent way of integrating Aspect Orientated Programming (AOP). Therefore, Spring is described in its specification as a *"potentially [...] one-stop-shop for all your enterprise application [...]"* while it still remain modular [39]. Spring being modular means that only the IoC-container is mandatory while all other parts of the framework are optional. The different modules are shown in figure 7.

3.2.1 Architecture

As shown in figure 7, Spring consist of six modules [39], which shall be described in this section. Because the IoC-container (Spring Core) is the fundament of the whole framework and the part of Spring that allows building applications through composition, this module shall be described in more detail in the following subchapters.

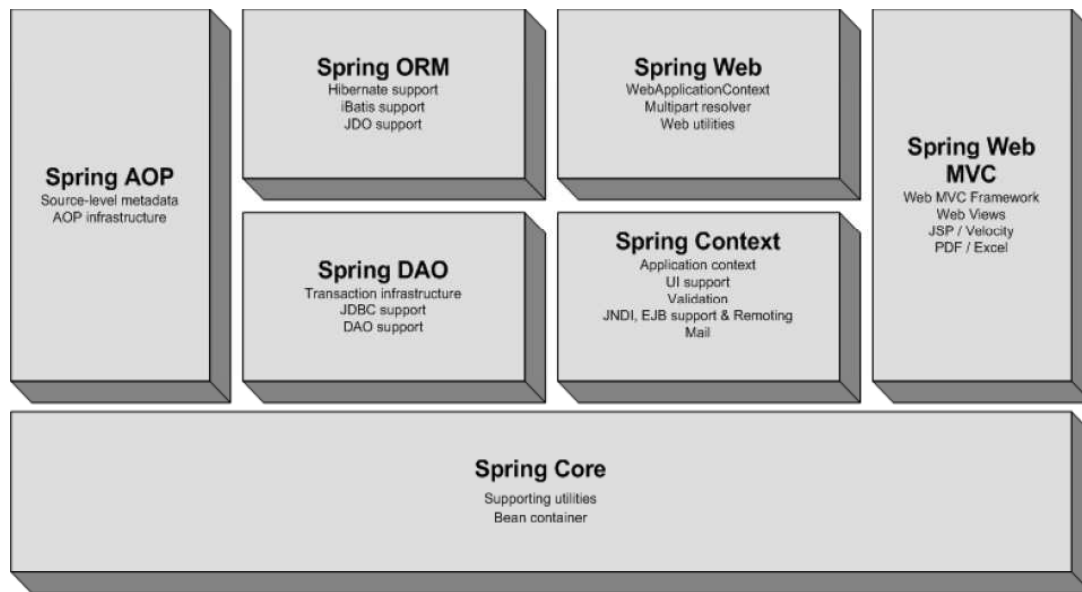


Fig. 7: Modules composing the Spring framework taken from [35].

Spring Core: The core package is the fundament of the Spring framework and in few words is an IoC-container providing formalized means for composing classes, objects or service to a full application [39]. These "components" are called Beans. To fulfill this goal, Spring work in most situations without invasive code, so that code can be reused outside of the Spring environment without changes. This allows decoupling of the Beans from configurations to build dependencies between these Beans.

The main concept of Springs IoC-container is the BeanFactory and the ApplicationContext. The latter is a superset of the BeanFactory which provides additional functionality. The BeanFactory is a generic factory allowing the retrieval of named objects, but the most powerful functionality of Spring is Dependency Injection (DI) (see section 2.3). The principle of DI in Spring is further explained in section 3.2.2.

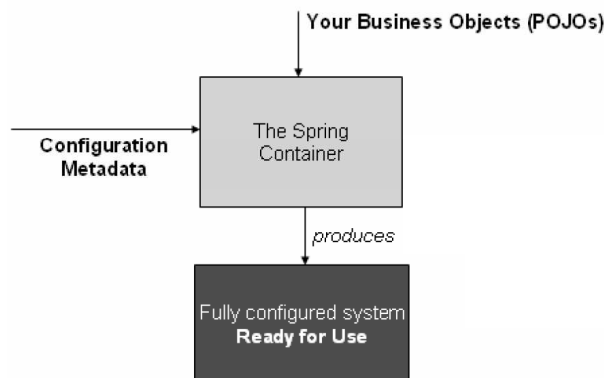


Fig. 8: Assembly model of spring to configure Plain Old Java Objects (POJOs) taken form [39].

Figure 8 shows the system with which Spring configures and assembles applications. The Spring container assembles Plain Old Java Objects (POJOs) with the help of Configuration Metadata to a full configured system.

Spring DAO: The Spring DAO (Data Access Object) module is an abstraction layer for the JDBC service to simplify this service and allows additional abstraction of database vendor specific functions. It also provides programmatic and declarative transaction management.

Spring AOP: The Spring AOP (Aspect Orientated Programming) module provides standard based support for AOP method-interceptors to additionally decouple the code. It extends the annotation system of Java to a system similar to the attribute system in Microsoft .NET and is also compatible to the largest AOP framework called AspectJ.

Spring ORM: The Spring ORM (Object-Relation Mapping) module provides integration of popular ORM APIs such as JPA, JDO, Hibernate or iBatis to handle even these APIs with the one-hand-shop Spring.

Spring Context (JEE): The Spring Context or JEE module extends the Core module with a JNDI like object registry, support for internationalization (i18n), event-propagation, resource-loading and transparent context creation through the containers.

Spring Web and Web MVC: The Spring Web and Web MVC module provides additional functionality for server side and web applications such as multipart file-upload, support for servlets and a Model-View-Controller (MVC) implementation for web-applications to cleanly separate between domain model code and web forms and therefore propagates the use of best-practices in web development.

The most important module in respect of components is the Core module with its IoC-container. This module shall be further described in the following subchapters.

3.2.2 Beans

Beans are so called Plain Old Java Objects (POJOs), meaning simple Java object without any special structure, that are referenced in configuration metadata, typically a XML document, and which are therefore accessible by the Spring IoC-container.

In Spring, there exist two classes allowing the retrieval of Beans: BeanFactory and ApplicationContext. These classes exist in different implementation like for example an XmlBeanFactory, which allows the retrieval of Beans which metadata is stored in XML documents. XML documents are also the most used way to configure the IoC-container. The concept of BeanFactories also allows defining customized factories that load metadata from any source. The ApplicationContext is a superset of the BeanFactory extending its interface by additional, enterprise centric functionality [39]. Holding these classes, any object is able to retrieve every Bean registered for the ApplicationContext or BeanFactory by calling a method with the name of the required Bean as parameter. Although this is a very straight forward solution, the reference of Spring [39] gives the advice to avoid this technique because it creates dependencies between the code and the Spring framework because a Spring specific interface has to be implemented to get a reference to the BeanFactory or ApplicationContext which prohibit to use such a

class without Spring.

The better solution to define the dependencies between POJOs in Spring is to use the Dependency Injection (DI) (see section 2.3) system. Spring supports constructor injection as well as setter injection, whereas setter injection is preferred by the developers of Spring, because it allows Spring to reinitialize Beans to a later point in time while when using constructor injection, this is not possible because the constructor can only be called once on an object. Another problem of constructor injection using Spring is that constructor injection can lead to circular dependencies that cannot be resolved by Spring because the constructor has always be called prior to every other operation. Using setter injection, Spring is able to inject not fully configured Beans to resolve circular dependencies.

Before discussing the DI system in Spring, the metadata describing beans has to be explained. The definition of a Bean in Spring do consist of several parameters [39]

- **class:** defines the class which represents the Bean and implements the business logic of the bean.
- **name:** also called id, allows to explicitly identify this Bean.
- **scope:** The Scope of a bean is discussed later in this chapter.
- **constructor arguments:** allows to inject dependencies into the constructor of a class. A further description of this parameter is provided later in this chapter.
- **properties:** allows to inject dependencies into setter methods of a Bean. A further description of this parameter is provided later in this chapter.
- **autowiring mode:** enables Spring to automatically create dependencies and perform dependency injection between classes. The autowiring functionality of Spring is discussed later in this chapter.
- **dependency checking mode:** The dependency checking mode allows modifying the way Spring checks dependencies and how it reacts if it fails to load a dependency.
- **lazy-initialization mode:** The lazy-initialization mode allows determining the moment when Beans are initialized. This parameter is only relevant if a Bean is provided in the Singleton scope (see later in this chapter).
- **initialization method:** defines a method to call after a Bean is initialized and configured. This parameter is further explained in section 3.2.3.
- **destruction method:** defines a method to call before a Bean is destroyed. This parameter is further explained in section 3.2.3.

The DI mechanism in Spring is very simple and therefore straight forward. In the metadata definition of a Bean, there is the possibility to define constructor arguments and/or properties. Through these parameters, it is possible to define simple values or Beans that are injected into a Bean when it is initialized by the Spring framework. In the case of constructor arguments, this means that a constructor that has parameters defined is called with the arguments defined in the metadata. Simple values like nulls, integers, floats or strings or collections of these types are injected directly. If a reference to another Bean is defined, this Bean is initialized and afterwards injected into the Bean that was originally requested.

Properties are used for setter injection. A Bean using this mechanism has to have a constructor without parameters and setter methods equivalent to the defined properties. After an initialization request of such a Bean, the class is instantiated and the setters are called one after the other with the specified simple values or Beans. Because all these dependencies are only defined in the Spring metadata, the code of the Beans itself has no dependencies on the Spring framework and can therefore be also used outside of Spring [39]. In addition to these two mechanisms of dependency injection, it is also possible to define Beans that are instantiated using a static factory method or even an instance factory method.

Autowiring is a special functionality of Spring allowing to delegate specification of the dependencies between Beans to Spring to decrease the amount of configuration work of the developer. Spring analyzes the class represented by a Bean and decides which other Bean shall be injected into a property or constructor. There are five modes for autowiring in Spring [39]:

- **no:** No autowiring is used.
- **byName:** Spring searches for Beans that have exactly the same Name as a specified property and will inject it.
- **byType:** Spring searches for Beans that have the same type as a property requires. If more than one matching Beans are found, an exception is raised. If no matching Bean is found, null is injected or an exception is raised.
- **constructor:** The same as byType but for constructor arguments.
- **autodetect:** Spring uses the byType mode if a default constructor is found in a Beans class, otherwise, it uses constructor mode.

Spring defines several additional mechanisms to customize the autowiring mechanisms so that it can be used more efficiently which can be found in the official reference [39].

Spring distinguishes five different Scopes for Beans. The scope defines the instantiation rules of a Bean and has impact on its lifecycle. The Singleton and the Prototype scope of Spring are available for all applications while the Request, Session and Global Session scope are only available for web applications [39].

- **Singleton:** The singleton scope defines that a Bean shall only be instantiated once. If several Beans request an instance of a singleton Bean, Spring returns the same instance to all requestors.
- **Prototype:** The prototype scope defines that Spring returns a new instance of a Bean to every requestor.
- **Request:** The request scope defines that Spring uses a new instance of such a Bean for every HTTP request.
- **Session:** The session scope defines that Spring uses a new instance of such a Bean for every HTTP session.
- **Global Session:** The session scope defines that Spring uses a new instance of such a Bean for every global HTTP session.

Beside these base functionalities of the Spring IoC-container, Spring allows additional configurations to simplify and extend the use of Beans. There are several ways to reference a Bean when defining dependency injection which have slightly different behavior concerning type checking. Furthermore, it is possible to define aliases for beans to use the same bean with different names, to use different styles of writing metadata XML documents, to use automatic dependency checking, to inject only methods instead of full objects, Resource Management, support for Testing, Transaction management and so on. It is also possible to import configurations into other configurations to reuse beans of other configurations. It is also possible to build hierarchies of beans and to reference parent beans although having the same name. Another additional functionality of Spring is that it is possible to define dependencies although no dependency injection is performed. Such a depends-on reference is for example required if a Bean has to access static methods of another bean which has to be available prior to the completion of the instantiation process.

3.2.3 Life-Cycle

The life-cycle of Beans in Spring depends on the scope for which the Bean is defined. Using Singleton scope, the whole life-cycle of the Bean is handled by the container. Singleton Beans are by default instantiated when the container is instantiated. This leads to higher initialization times but increases the performance during execution of the application. Another advantage of this early instantiation of Singleton scoped Beans is that errors are detected very early in the process [39]. Spring allows the developer to define Beans with lazy-initialization, what means that such Beans are only instantiated if they are requested by other Beans. Beans using a Prototype scope are only instantiated if they are requested by another Bean. After the Bean is fully configured and handed to the requestor, the IoC-container has no longer any reference to the instance and therefore the remaining life-cycle is handled by the client.

Spring provides several mechanisms to define life-cycle callbacks to customize the behavior of a Bean when being created or destroyed [39]. When using initialization callbacks, it is possible to either implement a special interface that defines a method called after all properties are set, to define an initialization method in the configuration metadata or to use an Annotation to indicate a method as initialization method. Defining the initialization method in the metadata configuration has the advantage of not coupling the code to Spring. The same techniques are also available to define a method that is executed before a Bean is destroyed to release resources. To simplify the task of defining these methods in the configuration metadata, it is possible to define a default initialization or destroy method that is used for every Bean defined in the metadata source. When several of these techniques are applied to a single Bean, there is a strict ordering in which the defined methods are called:

1. Annotation based initialization or destroy method
2. Initialization or destroy method inherited through use of the Spring provided interface
3. Initialization or destroy method defined in the configuration metadata

As mentioned above, Spring stops managing the life-cycle of Prototype scoped Beans when handing them to the client. Therefore, destroy methods are not called automatically by the container and it is the responsibility of the client to perform cleanup operation by its own to free resources [39].

Because Spring manages the life-cycle of all Singleton scoped Beans, shutting down a non-web-based Spring application gracefully requires to register a shutdown hook with the Java Virtual Machine (JVM) which can be done using the `ApplicationContext` class provided by Spring.

3.2.4 Extension Points

Spring additionally allows customization of many aspects of the IoC-container without the need to subclass the `BeanFactory` or `ApplicationContext` [39]. Therefore, Spring provides interfaces allowing to define extensions to the container. All types of possibilities of this mechanism would exceed the scope of this paper, but for example it allows to

- Customization of the Beans instantiation process using `BeanPostProcessors`
- Modification of the configuration metadata using `BeanFactoryPostProcessors`
- Customizing instantiation logic to for example express complex instantiations in Java code using `FactoryBeans`

3.2.5 Language

Spring is not really an implementation independent standard, because it was developed in Java to illustrate the examples presented in the book "Expert One-on-One Java EE Design and Development" written by Rod Johnson in 2000 [36]. Therefore, Spring was originally only available in the Java programming language, but because of the overwhelming success of the Spring framework and the fact that Spring do not require special functionality only provided by Java, there are today also Spring version for other programming languages such as Python [40] C# based on the Microsoft .NET framework [41], but this do not represent a one-to-one port of Spring to the .NET platform but was developed by its own only inspired by the Java Spring framework.

3.2.6 Problems

One problem of Spring is that it is not language independent. Although implementations in other programming languages than Java exist (see section 3.2.5), Spring is not a standard and therefore do not define its principles in general.

The largest problem of Spring concerning this paper is, that Spring is not a full component framework and therefore miss several important things that would make Spring even more applicable. These includes for example

- Management and control mechanisms for the communication between components (that is because in Spring, Beans do not communicate over the framework but are only composed by the framework and communicate directly).
- The life-cycle of Spring Beans is very limited and not comparable to a full component framework where components can be deployed and removed at runtime. Beans are not as mobile as for example the key requirement of [2] demands.

Because these characteristics of a component framework are not available with Spring, Spring do not allow that Beans are bought from other vendors and included into an application without at least configure the metadata configurations of the Beans.

3.2.7 Portfolio

Beside the Spring framework itself and Spring.NET, Spring provides several other platforms for the development and deployment of enterprise applications [42]:

- **Spring Security** (formerly Acegi) provides comprehensive authentication, authorization, instance-based access control, channel security, human user detection capabilities and is configurable, allows the reusability of security components and can be used without Spring [43].
- **Spring Web Flow** provides a domain-specific-language for defining reusable controller modules, an advanced controller engine for managing conversational state, first-class support for using Ajax and first-class support for using JavaServerFaces with Spring [44].
- **Spring Web Services** provides a framework to simplify the use of web services and to propagate best practices in this field. It supports several standards in the field of web services [45].
- **Spring Dynamic Modules for the OSGi Service Platform** provides means to easily integrate Spring applications into the OSGi framework [46].
- **Spring Batch** provides reusable functions for processing large volumes of records, including logging/tracing, transaction management, job processing statistics, job restart, skip, and resource management as well as more advanced technical services and features to enable extremely high-volume and high performance batch jobs through optimization and partitioning techniques [47].
- **Pitchfork** provides JSR-250 dependency injection, annotation processing and EJB 3.0-style interception, developed by Interface21 and BEA Systems [48].
- **AspectJ**, that is the most used AOP framework for Java.
- **Spring IDE** is a graphical user interface for the configuration files used by the Spring Framework for the integration into the Eclipse platform [49].
- **Spring LDAP** is a simplification of the Lightweight Directory Access Protocol for the Spring framework [50].
- **Spring Rich Client** provides a solution for developers that need a platform for constructing high-quality Swing applications quickly [51].
- **Spring Integration** provides support for well-known Enterprise Integration Patterns while building on the Spring Framework's existing support by enabling messaging within application and integration with external application. It states to help producing maintainable, testable code [52].

3.2.8 Products

Spring builds on no explicit specification and therefore is a product by itself. Compared to all other presented frameworks and concepts of this paper, there are no implementations for Java that can be

named here as products of Spring. There are only ports of Spring to other programming languages such as for example Phyton [40] and the Spring Portfolio products presented in section 3.2.7.

3.2.9 Conclusion

Spring framework is a very straight forward solution to introduce loc/DI into an application. Its modular one-hand-shop approach allows to only using that parts of spring that are really required.

Spring allows composing Plain Old Java Objects (POJOs) in many situations without invasive code and therefore allows using these objects also without Spring. The way Spring enables this is easy to learn but still allows a lot of flexibility and possibilities. Because Spring was developed to ease the development of J2EE applications, it provides perfect integration into this environment.

All in all, Spring is a very good solution for DI but is not really a component framework as defined in section 2.2, because several aspects of components frameworks cannot be fulfilled using this framework.

3.3 Java Platform, Enterprise Edition (J2EE) / Enterprise JavaBeans (EJB)

Another state-of-the-art component framework is the Java Platform, Enterprise Edition (J2EE) especially one part of this platform called Enterprise JavaBeans (EJB). The J2EE platform is mostly used for web applications and is integrated in most application servers (tomcat, glassfish, JBoss, BEA WebLogic, and IBM WebSphere, ...), but there is also the possibility for other http-servers (jetty) to use J2EE as extensions.

The first version of the J2EE platform specification was released in 1999 and is now available in version 3.0 since May 2006. It was developed in cooperation of Apache Software Foundation, BEA, Fujitsu-Siemens, IBM, IONA, JBoss, Macromedia, Nokia, Novell, Oracle, Pramati, SAP, SAS Institute, Sun Microsystems, Tibco, Tmax Soft, Versant and Xcalia [53].

3.3.1 Architecture of J2EE

The official J2EE platform specification [54] defines that the key requirements for business services extending existing enterprise information systems (EISs) are

- High availability
- Security
- Reliability and Scalability.

Java tries to fulfill these self created targets with the specification of the Java Platform, Enterprise Edition (J2EE). The specification [54] states that this platform reduces the cost and complexity of developing multitier, enterprise services which can be rapidly deployed and easily enhanced. Therefore, a standard architecture exists containing the following elements:

- The Java EE Platform as standard platform for Java EE applications.
- The Java EE Compatibility Test Suite to verify the standard compatibility of products.

- The Java EE Reference Implementation to simplify prototyping and as operational standard of the Java EE Platform.
- The Java EE Blueprints as best practices for developing multitier, thin-client services.

The architecture of the Java Platform, Enterprise Edition is shown in figure 9.

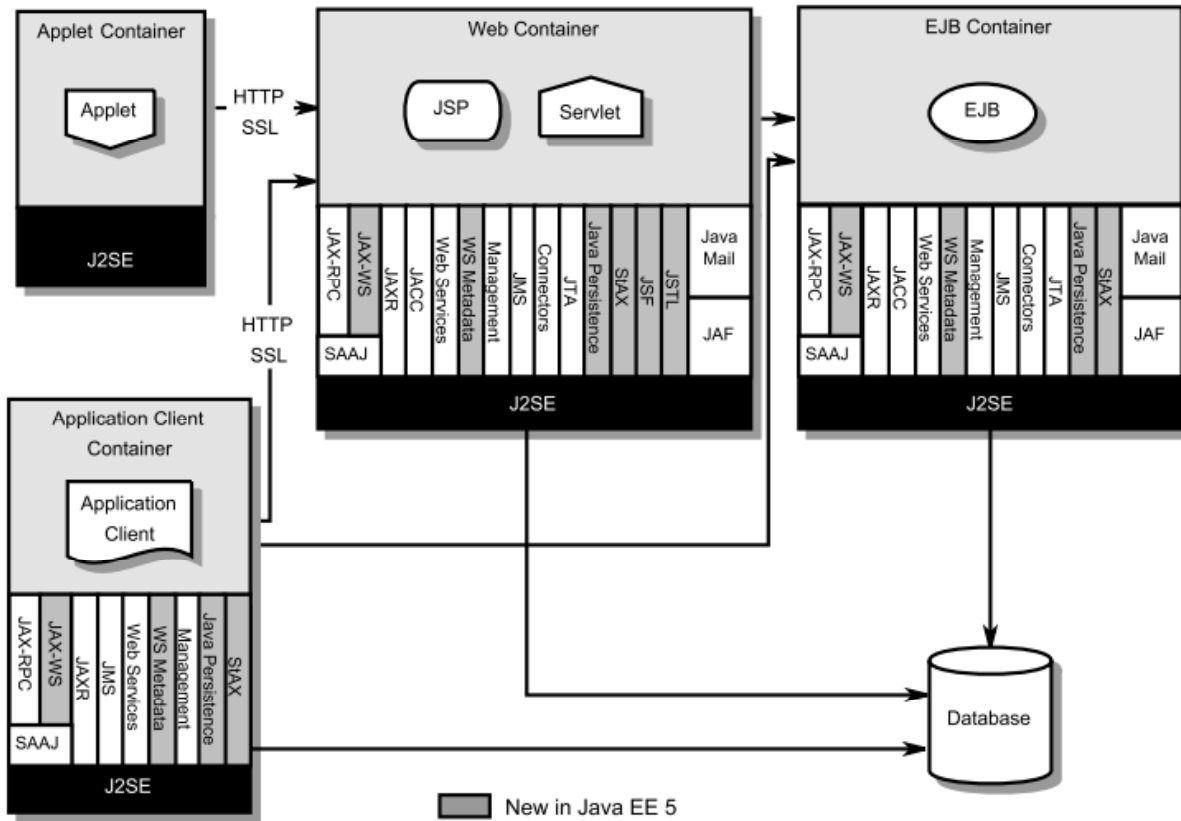


Fig. 9: Logical architecture of the J2EE 5 Platform taken from [54].

As shown in figure 9, the J2EE platform consists of four different parts, shown as rectangles. All these parts are containers that handle different types of applications. All these containers are in a logical or functional relationship illustrated by arrows connecting them. The specification [54] describes the four containers as

- **Applet Container:** Applets are graphical user interfaces shown in browsers or other applications or devices that support the applet programming model. These applets have only access to the Java Standard Edition API (J2SE).
- **Application Client Container:** The application client container handles standard java applications with graphical user interfaces. Application clients have access to several standard services like web services or the Java Message Service (JMS). Furthermore, the application client container enables the communication with web containers, EJB containers and databases.
- **Web Container:** The web container handles Java Service Pages (JSP) as well as Servlets. These

are also called web components [54] and handle HTTP requests from a client and respond with the creation of HTML pages or over web services. The container enables access to the ejb container as well as to databases. Furthermore, it provides other protocol support and security support as well as services like web service support, persistence, JMS or mail.

- **EJB Container:** The EJB container handles Enterprise JavaBeans (EJB) that are small components of business logic executed in the container. EJBs can access through the container different J2EE services and databases.

All these components can be deployed, managed and executed in a J2EE platform, but there are differences to which extend [54]. Application clients are only to a small part managed by the J2EE platform, because deployment and management is not completely defined by the specification. Some components are only deployed and managed by the platform, like HTML pages, whereas web components and EJBs are fully deployed, managed and executed in a J2EE platform. Therefore, these parts can be described as component frameworks.

The standard J2EE services as shown in figure 9 are

- HTTP
- HTTPS
- Java Transaction API (JTA)
- RMI-IIOP to enable CORBA interoperability
- Java IDL to enable CORBA interoperability
- JDBC API database connector
- Java Persistence API
- Java Message Service (JMS)
- Java Naming and Directory Interface (JNDI)
- Java Mail and JavaBeans Activation Framework (JAF)
- XML Processing
- Java EE Connector Architecture to support access to Enterprise Information Systems (EIS)
- Security Services such as Java Authentication and Authorization Service (JAAS), Plugable Authentication Module (PAM) or Java Authorization Service Provider Contract for Containers (JACC)
- Web Services such as Java API for XML Web Services (JAX-WS), Java API for XML-based RPC (JAX-RPC), Java Architecture for XML Binding (JAXB), SOAP with Attachments API for Java (SAAJ) or Java API for XML Registries (JAXR)
- Management API
- Deployment API

The following sections (3.3.2 to 3.3.5) describe the component framework of Enterprise JavaBeans.

3.3.2 Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) technology is the server-side component architecture for Java Platform, Enterprise Edition (J2EE). EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology [55].

Enterprise beans are small units of business logic executed in a container that handles the communication between enterprise beans, provide different services for the beans and handles the life-cycle of the enterprise beans. Therefore, the EJB container is a component framework. Enterprise beans can be accessed by so called clients through the container [56].

3.3.3 Architecture of Enterprise JavaBeans

The Enterprise JavaBeans (EJB) architecture is an architecture for the development and deployment of component-based applications, which are scalable, transactional and multi-user secure [56]. As for all component frameworks, EJB applications can be deployed on any server platform supporting the Enterprise JavaBeans specification.

This section only describes the third version of the EJB specification. Several aspects of EJB can be completely different in former versions of the specification. For example, while in former specifications, there was the need of declaring EJB relevant classes, interfaces and properties through a Deployment Descriptor, EJB 3.0 allows to do all declarations using metadata annotations [53].

As described in the EJB core specification [56], there are five different types of beans to compose EJB applications.

Stateless Session Beans: A stateless session bean is a small unit of business logic that has no internal state and therefore cannot be in a conversational state with a client, but it is allowed to have instance variables that contain data while handling the call of business methods provided by this bean. The container prevents so called re-entrant calls. This means that it is not possible that two clients enter the same business method of the same instance of a stateless session bean at the same time and therefore interfering each others call.

The container handles the whole life-cycle of a stateless business bean as well as the decision which instance of a stateless session bean handles a call. That enables the container to pool several instances of stateless session beans to handle client calls. Correlating to the number of work available for a stateless session bean, the container is able to instance more beans or remove bean instances if the amount of work decreases. That allows a good scalability for systems composed of stateless session beans. As all enterprise beans, stateless session beans have access to databases and can be executed in a transactional environment provided by the container. The life-cycle of stateless session beans is shown in figure 10.

As by all Enterprise JavaBeans it is possible to define interceptor methods, for example after the construction of the bean (and after all dependencies were injected) or before a bean is destroyed. This allows a greater flexibility and better handling of resources like database handlers.

The functionality of a stateless session bean (so called business methods) can be accessed by a client through the business interface of the bean. The business interface is implemented by helper

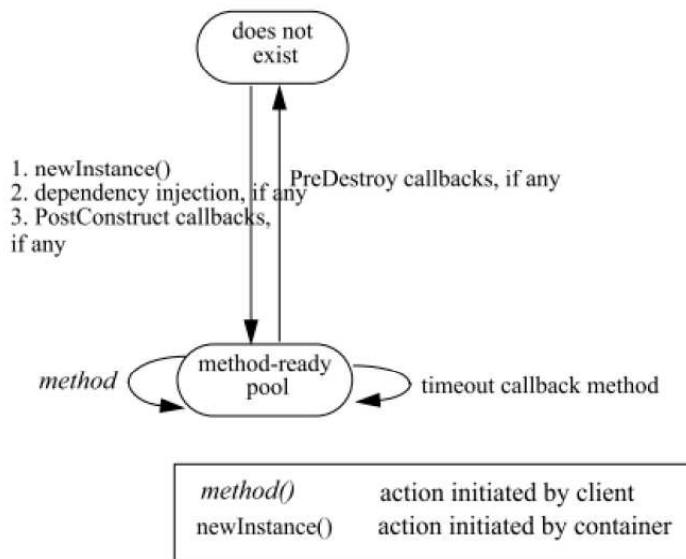


Fig. 10: Life-cycle of a stateless session bean handled exclusively by the container taken from [56].

classes provided by the container. The container then routes calls to the business interfaces to any available instance of the same session bean. There are two types of interfaces: remote or local. Remote interfaces enable the access of beans running in different Java Virtual Machines (JVM) and data is passed by value. The implementation of the interface is therefore location independent. Local interfaces enable only the access to a bean running in the same JVM and data is passed by reference, but the implementation of the interface is not location independent. It is possible for a stateless session bean to have a local and a remote interface. Because stateless session beans only implement business logic and do not have an internal state, they do not survive a crash of the server.

Stateless Session Beans with web service endpoint: These beans are very similar to stateless session beans. The only difference is that they have a web service endpoint as business interface to be accessed through protocols like SOAP. It is not possible to provide stateful session beans that have a web service endpoint.

In contrast to other beans, the client view of such a bean is not a Java class but it is described by a WSDL document which is accessed by HTTP or HTTPS protocols. Therefore, the interface is location independent and remoteable. By default, stateless session beans with web service endpoint access the web service with the JAX-WS or JAX-RPC client APIs.

Stateful Session Beans: The biggest difference between stateless and stateful session beans is that a stateful session bean's life-cycle is controlled by the client and not by the container, because such a bean is always in a conversational state with the client and therefore would lose data if the container would destroy it. This allows a session bean to hold its state across several methods and transactions.

To better handle the resources of the J2EE server, the container can passivate stateful session beans. Passivation means that the bean is saved to a secondary storage. Therefore, instance fields of the beans have to fulfill certain criteria. To enable the bean to get into this state, additional interception methods are provided. If a call to a passivated instance of a stateful session bean occurs, the bean is activated and its state and references are restored. Passivated instances can be removed by the container after a predefined time period (timeout).

Because the life-cycle of a stateful session bean is controlled by the client and not by the container, this life-cycle is far more complex as shown in figure 11.

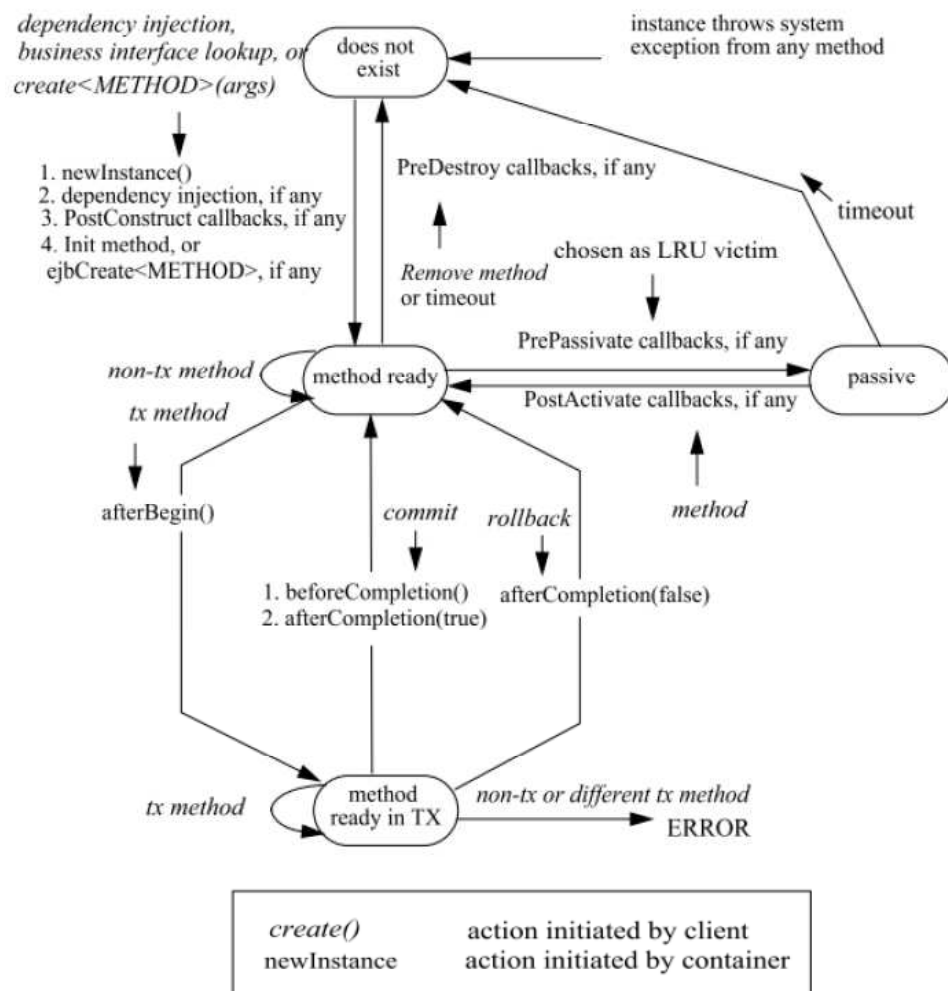


Fig. 11: Life-cycle of a stateful session bean taken from [56].

A stateful session beans is instantiated when a reference to the beans is retrieved or the *create<METHOD>* is invoked at the beans home interface, all dependencies are injected and the bean is ready to use. Is the bean ready, it is possible to invoke business methods on the bean and the bean can be passivated by the container. A bean that is activated is always in the ready state. If a transaction based method is invoked on the bean, the bean goes into the "ready in TX" state.

In this state, it is only possible to perform transaction based methods while all other methods results in an error. If the transaction is committed or rolled back, the transaction ends and the bean enters the state "ready". To better handle the transaction management, there are additional interception methods after the beginning of a transaction (before the first business method is invoked) and before and after the end of the transaction.

Message-Driven Bean: A message-driven bean is an asynchronous message consumer. That means that a message driven bean is similar to a stateless session bean, but instead of exposing an interface through which it can be accessed, listening on a message queue or topic. All message-driven beans are defined for a single message type and have no client-visible identity. They are stateless and their life-cycle is controlled by the container. Messages are handled asynchronously and it is possible to process a stream of messages concurrently.

The dependency to such a message source for which a message-driven bean is registered can be injected or looked up using JNDI and can be located in any container anywhere in a network. The container provides in addition to the life-cycle management security, concurrency, transactions and other services.

All other aspects of message-driven beans are similar to stateless session beans.

Entity Bean: Entity Beans represent persistent data in the J2EE platform [57]. They are related to a persistence context and are handled by an EntityManager. The persistence context allows entity beans to be a representation of relational data in a database that can be inserted, updated or removed. The EntityManager handles all these operations. An entity bean can have four different life-cycle states:

- **New:** A new entity does not exist in a database and therefore is not associated to a persistence context.
- **Managed:** A managed entity is associated with a persistent context and represents data in a database. This is the operational state of an entity bean.
- **Detached:** A detached entity was associated with a persistent context but is no longer.
- **Removed:** A removed entity is associated with a persistent context but represents data that is scheduled for remove from the database.

To allow entity beans being more flexible, there are interception methods for all state transitions.

There are several methods to handle entities. At first, there are possibilities to persist (insert), merge, remove and refresh entities. Refresh is required because if an entity is updated in the database, there is no automatic to refresh all other references to this identity. Furthermore, there are methods to create queries to find an entity in the database or to lock the entity.

As all other Enterprise JavaBeans, entity beans can be injected or retrieved using the JNDI service.

As described earlier, there are two technologies to get references to another bean. The first one is lookup through the Java Naming and Directory Service (JNDI) (see section 2.3). As stated by the official JNDI homepage [15], the JNDI allows to retrieve and store named java objects of any type. The

other one is through dependency injection (see section 2.3) where the EJB container performs the JNDI lookup for the developer [56].

3.3.4 Problems

One problem J2EE/EJB applications is that because J2EE is a Java technology, applications can only be developed in the java language, but because there are several integration technologies and protocols like for example JRMP, IIOP, HTTP/SSL or SOAP/HTTP supported by EJB, it is possible to integrate applications developed in other languages with J2EE/EJB applications.

Furthermore, J2EE is only provided as a standard. That means that although many aspects of it are standardize, the implementations can differ slightly from vendor to vendor which leads to the problem that it is possible that components developed for one implementation may not work correctly in the implementation of another vendor.

EJB uses annotations to identify EJB specific classes, interfaces or methods. It is still possible to define these elements in a declarative way, the actual version of the standard prefers the annotation based style [56]. This means that programming for EJB requires invasive code which can only be reused if the EJB libraries can be accessed.

J2EE is a big platform and it is not possible to use parts of it by their own. Therefore, J2EE and EJB as part of it are not modular enough to allow high flexibility.

3.3.5 Products

Products developed under the J2EE platform or which allow to use Enterprise JavaBeans include [58]

- BEA WebLogic (<http://www.bea.com>)
- IBM Websphere (<http://www-01.ibm.com/software/websphere/>)
- JBoss (<http://www.jboss.com/products/jbossas>)
- Sun Microsystems Sun Java System Application Server including glassfish (<http://developers.sun.com/appserver/>)
- Apple WebObjects (<http://www.apple.com/ca/webobjects/>)
- Borland Enterprise Server (<http://www.borland.com/us/products/appserver/index.html>)
- Fujitsu Siemens Computers Bean Transactions (<http://www.fujitsu-siemens.com/products/software/openseas/beantransactions.html>)
- Macromedia JRun Server (<http://www.adobe.com/products/jrun/>)
- SAP Web Application Server (http://www.thespot4sap.com/Articles/SAP_WAS_Overview.asp)

3.3.6 Conclusion

EJB is a component architecture for the J2EE platform developed in Java that allows many functionalities characteristic for component frameworks.

It is possible to develop independent components working together through the container which provides proxy interfaces for remote access and therefore allows the use of EJBs developed by other companies. It additionally provides basic life-cycle and management functionalities.

EJB is a good solution for composing web applications but it requires the J2EE platform which may binds components to vendor specific implementations and to the Java language, but also provides the technologies to communicate with components or applications over JMS or web services.

3.4 Service Component Architecture (SCA)

Service Component Architecture ¹ (SCA) was first designed by a group of vendors including BEA, IBM, Oracle, SAP and others. As seen in figure 12, SCA is now owned by OASIS, although the OSOA group does further specifications on the standard. Figure 12 also presents the time line of the development of the SCA standard.

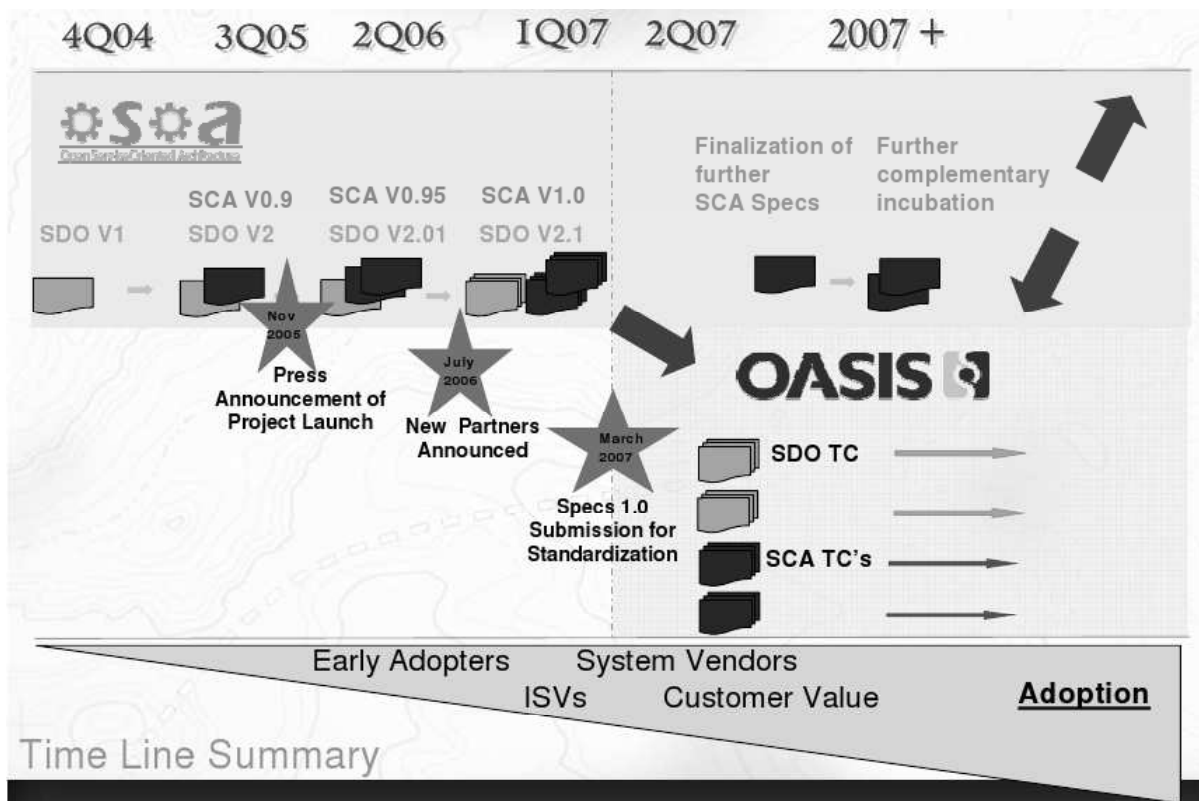


Fig. 12: Timeline of the development of the Service Component Architecture (SCA) standard taken from [60].

SCA provides a programming model for building applications and solutions based on a Service Oriented Architecture (SOA). The approach allows applications to run on the same machine or on two or

¹Within the SCA definition also a definition for Service Data Objects (SDO) was created to allow a unified access to data. But since JSR-235 [59] it is a standalone product. An SCA environment is not required to run SDO. The idea is equal to JPA for J2EE which now also can be used in non J2EE applications. It does not really belong any longer to SCA and is therefore out of the scope of this paper.

more machines. Components of an SCA could be binded by different technologies as Webservices, Messaging systems, Remote Procedure Call (RPC) and many others. Further more SCA allows to write different components in different languages and techniques as C++, Business Process Execution Language (BPEL), the Springframework or plain old Java (POJ). This system allows applications to use new business components as well as old services wrapped by SCA [61, 62].

3.4.1 Architecture

The SCA Model consists of a series of artifacts about which an overview should be given in the next chapter, referencing to the chapters describing the artifacts and concepts in greater depth.

SCA provides a programming model for building applications and solutions based on a Service Orientated Architecture (SOA) (see chapter 2). This means that the core tasks of the SCA standard are based on providing services, consuming them and setting properties in them.

The most basic artifact in this architecture is the composite shown in figure 13. The code contained in an component could be implemented by many different technologies, including "traditional" programming languages such as Java, C++ or BPEL, but also scripting languages such as PHP or JavaScript and declarative languages such as Xquery or SQL are supported. Services as well as references are services defined in the system. This could be done by Interfaces (e.g. in Java) or via WSDL. Properties could be set to services as well as to references. The concepts of the components themselves are further described at section 3.4.2.

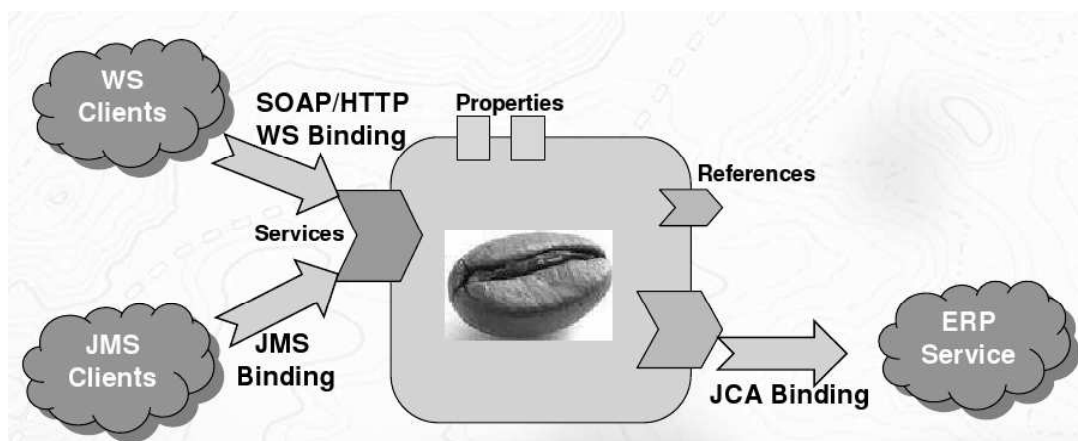


Fig. 13: Basic component model of SCA taken from [60].

Components are composed by the services required and provided by them. Which techniques used to bind them are described by SCA but is not limited. This global option allows also non SCA components as Java Server Pages (JSPs) or webservice clients to consume the services of a component and access it. Components could also use regular Data Access Technologies as Object Relational Mapping (ORM) via JPA, EclipseLink, Hibernate and similar implementations, Relational Object Mapping (ROM) as Ibatis or Java Data Objects (JDO). Nevertheless all these technologies are out of the scope of this paper, but they could be used by an SCA component as well as by an "old" application. A service could be bound to as many endpoints as wanted. These concepts are further explained in section 3.4.2.

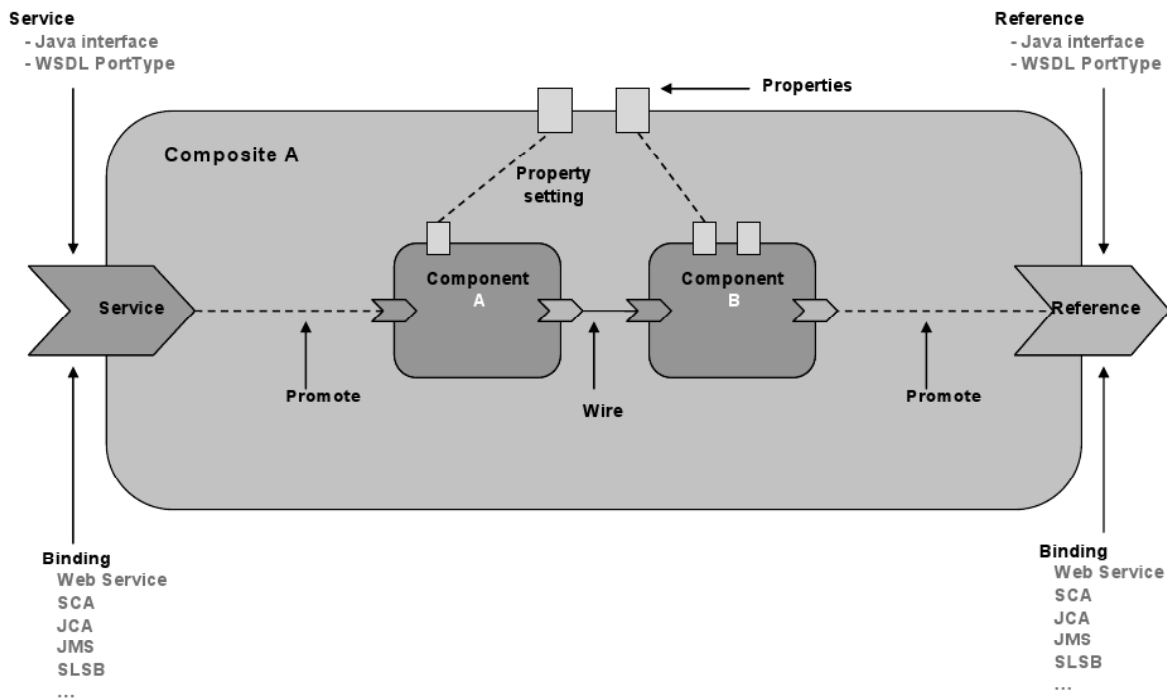


Fig. 14: Basic composition model of SCA taken from [61].

Components can be composed in composites. Composites could be seen as components containing components as seen in figure 14. References from the composite could be promoted to many components and any number of services provided by the components could be promoted to other composites/components. In turn, composites can be used as complete component implementations: providing services, depending on references and with settable property values. Such composite implementations can be used in components within other composites, allowing for a hierarchical construction of business solutions, where high-level services are implemented internally by sets of lower-level services. The content of composites can also be used as groupings of elements which are contributed by inclusion into higher-level compositions. A more detailed description of components could be found at section 3.4.2.

Composites could contain entire applications and are deployed to SCA domains. Typically these domains represent a set of services providing an area of business functionality that is controlled by a single organization. As an example a domain could be spread via the entire organization or just a part of it with closed functionality as the customer service containing all administrative services and address based functions. Although SCA does not provide a defined way to communicate between the SCA domains provided by two different vendors it's possible via standard bindings as webservices. In figure 15, a complete overview of the domain, composites and component model could be seen. Furthermore it's shown how external service provider and consumer communicate SCA domains. Further information about SCA domains could be found at section 3.4.4. Last but not least, SCA provides a model to control and restrict the bindings very strongly via policies. Policies can be used to guarantee the quality of service, the security between services and so on. Further information would be provided at section 3.4.5 [62, 61].

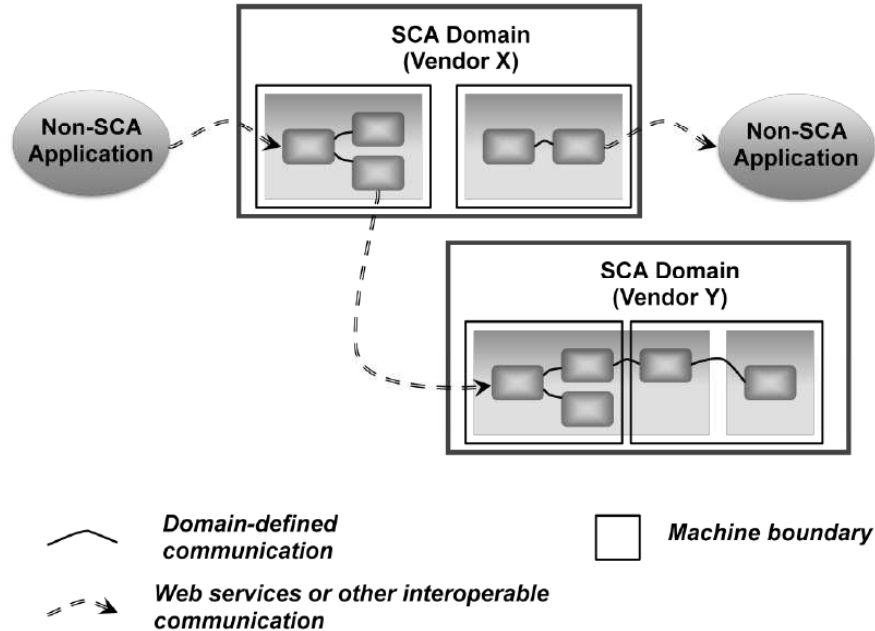


Fig. 15: Basic domain model of SCA taken from [62].

3.4.2 Components

Components are the atoms an SCA application is build of. Like atoms, SCA components behave in consistent ways and can be assembled into different configurations. According to David Chappell [62] a component is an instance of an implementation that has been appropriately configured. The code, written in Java or BPEL or one of many other languages, is called the implementation. The configuration defines how a component interacts with the outside world and is expressed in the Service Component Description Language (SCDL). According to SCA assembly model specification [61], the current specification does not mandate the implementation technologies to be supported by an SCA run-time; vendors may choose to support the ones that are important for them.

Components are made up of Properties, Services, References and Bindings which should be explained in the following descriptions.

Properties: Properties are simple values which are injected to the components the time the code is instantiated. Properties can be configured and their types are defined by the implementation. An implementation can also declare a property as multi-valued, in which case, multiple property values can be present for a given property [61].

Services and References: Services as well as References can be seen as interfaces describing one or more business functions. These functions are described as Java interfaces or as WSDL port. Nevertheless SCA provides extension points to offer more interface types than those could be offered. If these interfaces are provided by a component they are called services. If a component requires and consumes a services this function is described as a reference to the service for the

component.

Services can be described remotable or local. Local services are designed to be only used "locally" by other implementations that are deployed concurrently in a tightly-coupled architecture within the same operation system process. Normally local services rely on by-reference calling conventions and could assume a very fine-grained interaction style. Remoteable interfaces are constructed to be published remotely in a loosely-coupled SOA architecture. This type of service uses pass-by-value semantics and no technology-specific datatypes could be transferred [61].

Binding: Bindings are used by references as well as by services. References use bindings to define where the service to consume can be found. Services use bindings to provide their functionality to specific endpoints.

Two possibilities of binding have to be provided by each vendor implementing SCA. The SCA default-binding on one hand and webservices on the other. The SCA default-binding is not specified in the standard and it could be freely chosen by the vendor how components providing their service via this binding are connected to each other. This binding does not work if services are linked to be provided to non SCA components or to components in other SCA domains. Also services could not be consumed (references) via SCA default-binding if they residence in different domains or non SCA environments.

Beside the SCA default-binding and web services, many other bindings are possible to be implemented. Message Systems (JMS in Java), IIOP (EJB in Java) and the binary protocol (RMI in Java) are only some of the possibilities also defined by the SCA standard.

Additional it has to be said that a service could provide its methods to as many bindings as wanted. Nevertheless the order how these bindings have to be processed and some other details are not defined by the standard and are opened to be chosen by the vendors.

To describe how useful this way is, bindings are defined in SCA the way applications in Java use different protocols. As shown in figure 16 there are defined interfaces for each protocol abstracting the specific implementation of the technology.

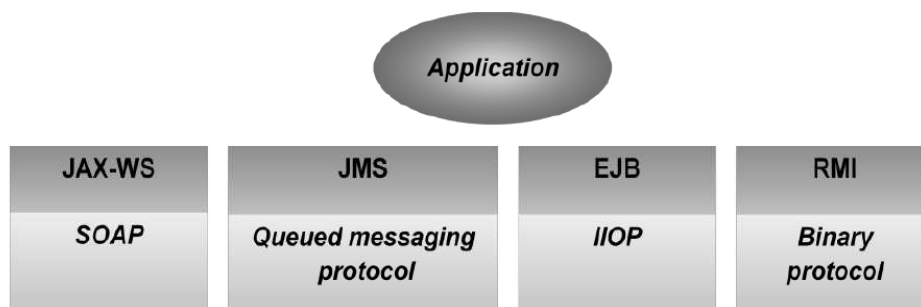


Fig. 16: The java binding approach in SCA taken from [62].

The disadvantage of such a technique is clear. The developer have to learn the interfaces for each technology. Furthermore, this way is quite invasive. In SCA the protocols are defined in a unified

way which provides the different implementations to the consumer. This way has two advantages. The developer does not have to learn different interfaces for each implementation in each language and environment. Secondly it allows a non invasive option for connecting implementations to others. Furthermore, its the only possible way for SCA vendors to work with different languages for implementation [62, 61].

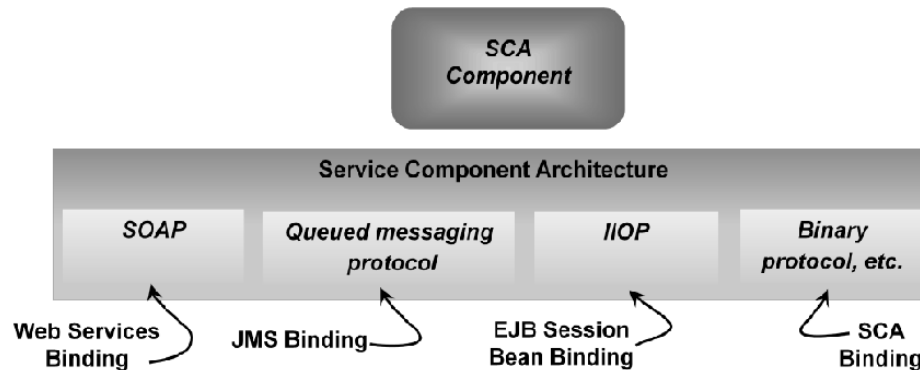


Fig. 17: The sca binding approach in SCA taken from [62].

3.4.3 Composites

To stay at the analogy of components with atoms, composites could be seen as molecules. Composites group components into useful combinations, which could be further combined themselves. Composites could be seen as components but working with components instead of an implementation in code (never mind the language of implementation).

Composites can define references, provide services, which are provided by components and can request references to external services which are provided as references to one or more components contained by the composite. This system allows to compose many lower level services to higher level ones and only provide the high level services to other composites/components while hiding the lower level.

Components deployed via one composite could be deployed only in one domain but the components in a composite might run entirely within a single process, across processes on a single machine, or be spread across processes on different machines.

The SCA assembly model specification [61] summarizes these ideas to the following characteristics of a composite:

- It may be used as a component implementation. When used in this way, it defines a boundary for component visibility. Components may not be directly referenced from outside of the composite in which they are declared.
- It can be used to define a unit of deployment. Composites are used to contribute business logic artifacts to an SCA domain.

One composite can be used to provide part of the definition of another composite, through the process of inclusion. This is intended to make team development of large composites easier. Included composites are merged together into the using composite at deployment time to form a single logical composite.

3.4.4 Domains

An assumption by the creators of SCA was that a given environment would install a group of SCA runtimes from a single vendor. For example it should be assumed that a division of a company would chose a particular implementor as its SCA vendor. This division is likely to install their chosen runtimes on any number of machines belonging to them. This expectation mirrors how organizations are typically purchased and installed J2EE products.

These SCA runtimes will likely be managed by the same group of people, and this set of systems with a common vendor's runtime technology and common management provides the primary example of a domain [62].

Domains are one of the most important concepts in SCA. This results from the fact that even though SCA allows creating distributed applications, it does not fully specify how components installed on different machines work together. The communication among these components/composites can be implemented differently and optimized by each vendor. As described in section 3.4.2, a default-binding between components/composites can be implemented. This default-binding only works between components in the same domain and is also not fully specified by SCA. Therefore this binding does not work between different domains. This does not mean that composites installed in different domains are not able to communicate. All they have to do is to take one of the standardized bindings as webs- or message services to bind to an external domain. This approach is also used to provide service to non SCA-applications and consume them inside of a SCA-domain. The external applications shall never see that they are bound to an SCA-domain [62].

As already stated, domains are the specific implementation of the SCA-definition by a vendor. Therefore many other details are specified by SCA in [61], as packaging, distribution, packaging, artifact resolution and so on. All these concepts would go beyond the scope of this paper.

3.4.5 Policy Framework

The capture and expression of non-functional requirements is an important aspect of service definition and has an impact on SCA throughout the lifecycle of components and composites. SCA provides a framework to support specification of constraints, capabilities and Quality of Service (QoS) expectations from component design through to concrete deployment [63]. It allows developers to use policies to specify their intents and let the runtime figure out how to achieve this intent.

The policy framework defines two broad categories of policies:

- **Interaction policies:** These kind of policies define how components have to interact with other components (how the services have to be used). This includes the requirements for security or for reliable message transfer. Quality of Service (QoS) is often used as a buzzword in this context.
- **Implementation policies:** These policies define the local behavior or components. For example

they describe that components may have to run inside of transactions.

Policies are described, as everything else in SCA, in SCDL configuration files directly at the bindings of the services/references they belong to. Policies are always defined directly to bindings.

The architecture of policies is quite simple in SCA by using intents and policySets. Intents are mapped directly to bindings whereas policySets are collections of intents which could be set to bindings.

An example for this is provided by David Chappell [62]:

"For example, a binding for a service can have an associated policySet describing its interaction policies, while a binding for a reference can have another policySet describing its interaction policies. When a wire is created between them, these policySets are matched, and their intersection determines the set of policies used for this communication."

However, SCA does not define a particular policy language, so each vendor is free to implement this in the way he prefers. Resulting of this fact policies could only be directly merged by the runtime of components/composites in the same domain. If communicating with external applications or SCA domains, the policies are used which are provided by the used technique itself. So WS-Policy would be used to define the restrictions of using published web services.

David Chappell [62] does also provide another example for the need of different policies in a company using the same SCA vendor through the entire system:

"[...] within this environment, different parts of an organization might require different policies. Suppose, for example, that two departments in the same company use the same vendors SCA product but have different security requirements. To address this, the firm might choose to create two separate SCA domains, each with distinct security policies [...] and bound them together via domain-external (non-default) bindings."

3.4.6 Implementation

Theoretically, SCA allows any language to be used for implementation and any protocol for binding services. Nevertheless only a subset of them is already specified. This chapter should shortly present the already existing specifications. It has to be mentioned that vendors of SCA runtimes are allowed to extend the supported implementations and bindings, but they have to be documented.

The base papers specifying SCA on which the entire chapter is heavily based on are the SCA assembly model specification [61] and the article by David Chappell [62]. Because web service binding is required, the web service binding specification [64] is also quite important.

The SCA specifications further offers specifications for C++ [65], BPEL [66], COBOL [67], C [68], EJB-session beans [69], Java [70][71], JavaEE [72] and spring-components [73].

Bindings are specified for web services [74], JCA [75] and JMS [76].

The policy framework is specified through the base framework specification [63] and the transaction policy specification [77].

3.4.7 Problems

Although SCA provides many advantages it also provides some disadvantages to the implementor. First of all the typical SOA problem of adding another layer of complexity to the software. Another problem is that an SCA user cannot expect what exactly an SCA implementation provides. Meaning SCA does not mean that talking about exactly the same. Some vendors provide C++, other COBOL but no C++ and so on. This problem is further discussed at section 3.4.9. The last problem is that SCA adds an inversive programming model for some programming languages as Java. Binding to the SCA definition files is provided via annotations which bind the code to the an explicit distributor.

Some of those problems could be handled via Spring or EJB as described at section 5.4 and 5.5. Others as different extensions implemented by different vendors will never be solved.

3.4.8 Products

SCA is a quite accepted specification as there are many implementations available for it [60]:

- Tuscany (<http://tuscany.apache.org>)
- Fabric3 (<http://fabric3.codehaus.org/>)
- IBM WebSphere (<http://www.ibm.com/software/websphere/>)
- BEA (<http://www.bea.com>)
- RogueWave (<http://www.roguewave.com/>)
- TIBCO (<http://www.tibco.com/>)

The Apache Tuscany project is published under the Apache license and provide an implementation of the latest SCA specification and Service Data Objects to provide a unified way to work with components and data. Fabrica3 is also an Apache licensed infrastructure supporting Java, Spring, Groovy, Ruby, BPEL and EJB. IBM, BEA, RogueWave and TIBCO offer commercial solutions and support for SCA, also containing developer suites which contain graphical tool support with the intend to speed up the development process.

3.4.9 Conclusion

SCA offers great possibility of creating SOA base applications without using invasiv techniques or let objects know about the way they are bound to other components and services. Nevertheless, there is also another side of the medal. The specification offers plenty of options to implement SCA and so implementations of different vendors can be completley different. Some runtimes may only support Java and C++ programming models where other vendors may also offer BPEL and COBOL. This means that componets created for one SCA-domain do not have to run properly at another domain although they are completely implemented according to the standard.

Although many implementation show that the concept of SCA works quite well, there are many questions remaining, e.g. what will conformance mean when SCA offers so many options? Nevertheless

the middleware convinces with a clear structure and unlimited possibilities and should not be lost out of sight if SOA is required for future application and integration projects.

3.5 Java Business Integration (JBI)

At the integration (Enterprise application integration (EAI) and business-to-business integration (B2B)) of different applications, problems occur very often and the complexity of the topic is often underscored. Integration should not be confound with distributed systems. In distributed systems, there are only a fixed number of layers and the communication between those layers is done with synchronous calls against a fixed interface. At the integration of different applications, most calls are asynchronous and the use of the services is only coordinated. As an example a ordering- and a mail system can be taken. If a customer makes an order, he or she should only receive one email from the system, but in most cases, the mail system runs on a different machine than the ordering system and it can happen that no connection can be established as one example [78].

This problem is not a new one. Formerly EAI and B2B solutions require the use of non-standard technologies to create functional systems. Nevertheless no implementor can create support for all of the standards available, which brings the software architect in the uncomfortable position of selecting less than ideal solutions to their integration problems, and paying dearly for them [8].

Some of the base requirements for a complete business solution can be found in the article by Krueger et al. [79]:

"(1) a high demand for flexibility with respect to the configuration and support of business processes to anticipate and cater to changing threat and mitigation scenarios, (2) high agility demands during both development and production to address legacy and emergent capabilities, processes, applications and technologies, (3) wide variety of trust relationships among and across stakeholders and their organizations."

As said by the Greek philosopher Plato [80]: *"Necessity is the mother of invention."* Sun Microsystem Inc. released the Java Business Integration (JBI) within the JCR 108 [81, 8] to create a standard based architecture for integration. The infrastructure allows third-parties to produce components pluggable to a standard infrastructure. Further more these components would be interoperate in a predictable, reliable fashion despite being produced by separate vendors. With the JBI, Sun anticipate that a multi-vendor "ecosystem" will be created rising a large pool of integration-related technologies which could be freely chosen by the user. Further more this system fosters new innovation in integration technologies because the creators of the components could concentrate on a particular area and have not to worry about providing all the details needed to build a complete integration platform [8] (see section 2.3).

The introduction of the official JBI specification [8] describes best further advantages of JBI:

"Every integration problem is unique; an appropriate combination of JBI-compliant components will provide a solution that is sized appropriately to the problem at hand. By avoiding lock-in to a particular vendor of integration technologies, the user is free to choose components that provide the particular functions that he or she needs, and be assured that a functional integration solution can be assembled from those pieces. In the past, attempts to compose third-party components into systems that have the attributes required of enter-

prise systems have not been very successful. JBI addresses this by adopting a service-oriented architecture (SOA), which maximizes the decoupling between components, and creates well-defined interoperation semantics founded on standards-based messaging. The SOA approach creates many other benefits that are applicable to enterprise integration solutions.”

3.5.1 Architecture

According to the JBI specification [8]

”JBI defines an architecture that allows the construction of integration systems from plug-in components, that interoperate through the method of mediated message exchange. The message exchange model is based on the web services description language 2.0 [WSDL 2.0] or 1.1 [WSDL 1.1].”

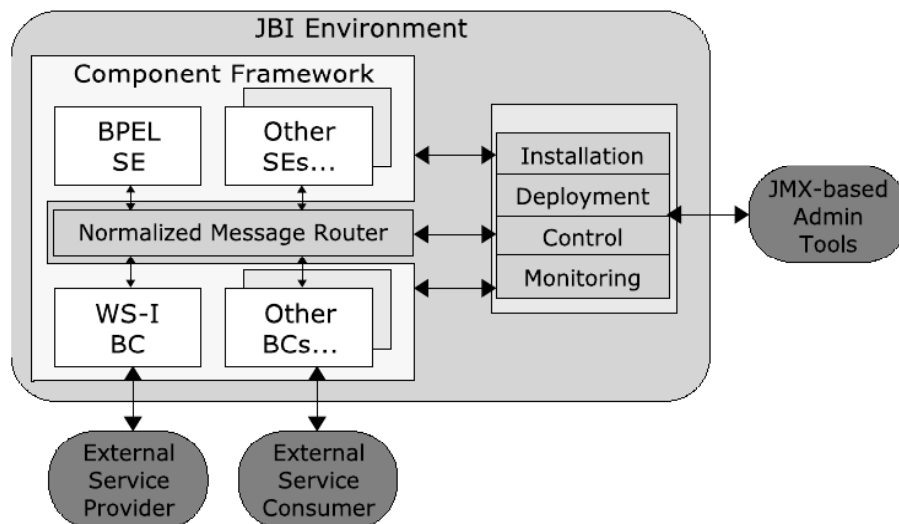


Fig. 18: Overall concept of the JBI architecture taken from [8].

The overall architecture of JBI can be split into three parts. A component framework (see section 3.5.2), a Enterprise Service Bus (Normalised Message Router) (see section 2.3 and section 3.5.3) and a management part based on Java Management eXtensions (JMX) (see section 3.5.4). The overall concept of the the JBI architecture is illustrated in figure 18 at an abstract level. JBI provides specific interfaces for the use by plugins while the plugin-in provides specific interfaces for the use by JBI. Because JBI functions work as an intermediary to route messages from one component to another, the plugins do not interact with each other directly. This separation is the key-concepts of decoupling service providers from consumers, which is highly desirable for service orientated architecture as well as for integration of systems. A very important system in JBI is that every processing is inherently asynchronous, which means that service provider and consumer never share the same thread.

The service model in JBI is quite simple. Service provider could provide functions to the system which are modelled as WSDL 2.0 operations. Every operation involves the exchange of one ore more mes-

sages between the partners (service provider and consumer) as defined. Every operation is done via one of four basic, WSDL-defined message exchange patterns, which should further explained in section 3.5.3. WSDL has the additional advantage of adding metadata to services which can be queried by consumer to request the correct service from the JBI runtime.

The component framework is split in two distinct and separated main types of components.

- **Service Engine (SE):** The service engine modules provide the business transformation logic for other components of the system and consume services. Service engines can also integrate Java-based applications and other resources.
- **Binding Component (BC):** Binding components are responsible for providing connectivity to services external to JBI. Therefore binding components can integrate applications and components which require or provide technology not available in java.

Service engines as well as Binding Components can consume and/or provide services. The distinction between SEs and BCs is purely pragmatic, but is based on architectural principles. The separation of business (and processing) logic from communication logic reduces implementation complexity and increases flexibility.

In addition, JBI defines a management structure based on JMX which provides standard mechanism for [8]

- Installing components.
- Managing a components life cycle (stop/start etc.)
- Deploying service artifacts to components.

3.5.2 Component Framework

One of the base components in the JBI is its component model. The middleware is made up of four component types, a fixed defined lifecycle and standardized interfaces provided by JBI to allow the management of components in the system. The base concept of the plugin model provided by JBI is demonstrated in figure 19.

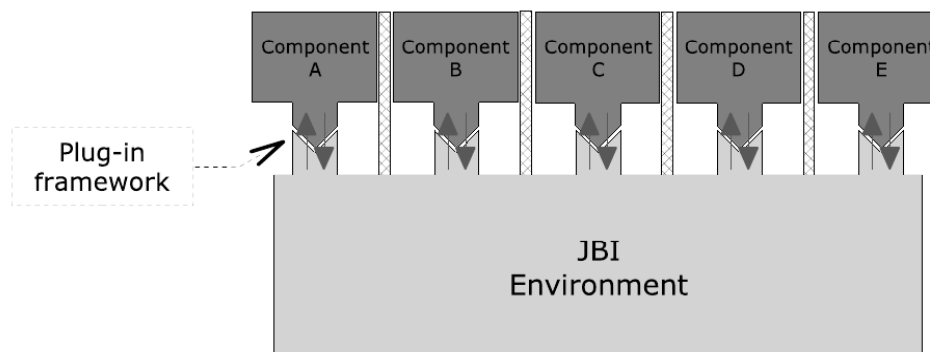


Fig. 19: Base concept of the JBI plug-in model taken from [8].

Service Engine In contrast to the Binding Component, the Service Engine itself provides business and transformation logic. The implementation of these components is mostly done commonly and provides an execution environment. Typical representatives of this branch are XML Transformation (XSLT) or the Business Process Execution Language (BPEL). Service Engines have the possibility to use (deployment) so-called Service Units (SU). Service Units are, in the case of BPEL, the concrete BPEL-XML-files and at XSLT the transformation files. This brings the advantage, that the required logic has to be written just once. The Service Units just have to receive the right configuration parameters finally [78]. Service engines can also be used to orchestrate service consumption and provision or provide sophisticated routing or EDI services such as message collation / de-collation facilities [8].

Binding Components A binding component connects an external service with the JBI infrastructure. It converts external message protocols to use their functionality. This serves to isolate the JBI environment from particular external protocols by providing normalization and denormalization from and to the specific protocol format. Normally a binding component should not provide any business logic, which is beyond the scope of using the protocol. This paradigm is completely in the scope of the developer and is not checked, as already mentioned (see section 3.5.1), by the JBI. Binding components can be compared to J2CA according to their adaptability. Typical protocols implemented in binding components are SOAP, HTTP, FTP, SMTP, REST and ebMS. Nevertheless the possible implementations are not limited to these protocols [78, 8].

Shared Library This kind of component provides functionality for many JBI components. This is very beneficial at very huge components which need the same libraries for many binding components or service engines [78].

Service Assemblies A Service Assembly is an aggregation of more than one service engine, binding component or service library. In few words, a service assembly is a composed component. Not all kinds of components have to exist in it. The components contained by a service assembly provide the required functionality for a JBI application. This package simplifies the distribution of components a lot. In some cases it can happen that a sequence has to be followed which could also be defined in a service assembly [78].

Component Distribution Additional to the specification of the implementation of Java classes, it has to be defined, for the sake of reusability, how components have to be packaged, described and distributed. Therefore JBI abstains from choosing an own file extension and takes the extension zip/jar as typical for Java. An archive has to contain a folder with the name META-INF which has to contain a XML file with the name *jbi.xml*. This file contains information about the type of the component, what has to be done by the framework with the component, which class contains the bootloader and so on [78]. This behaviour is shown in figure 20.

Programming Model Although the programming model is out of the scope of this paper, some basics shall be provided. The exact contract between the framework and the components can be found in the JBI specification [8].

- The Component Installation context is provided to the component during installation and uninstallation to get access to the JBI-provided resources.

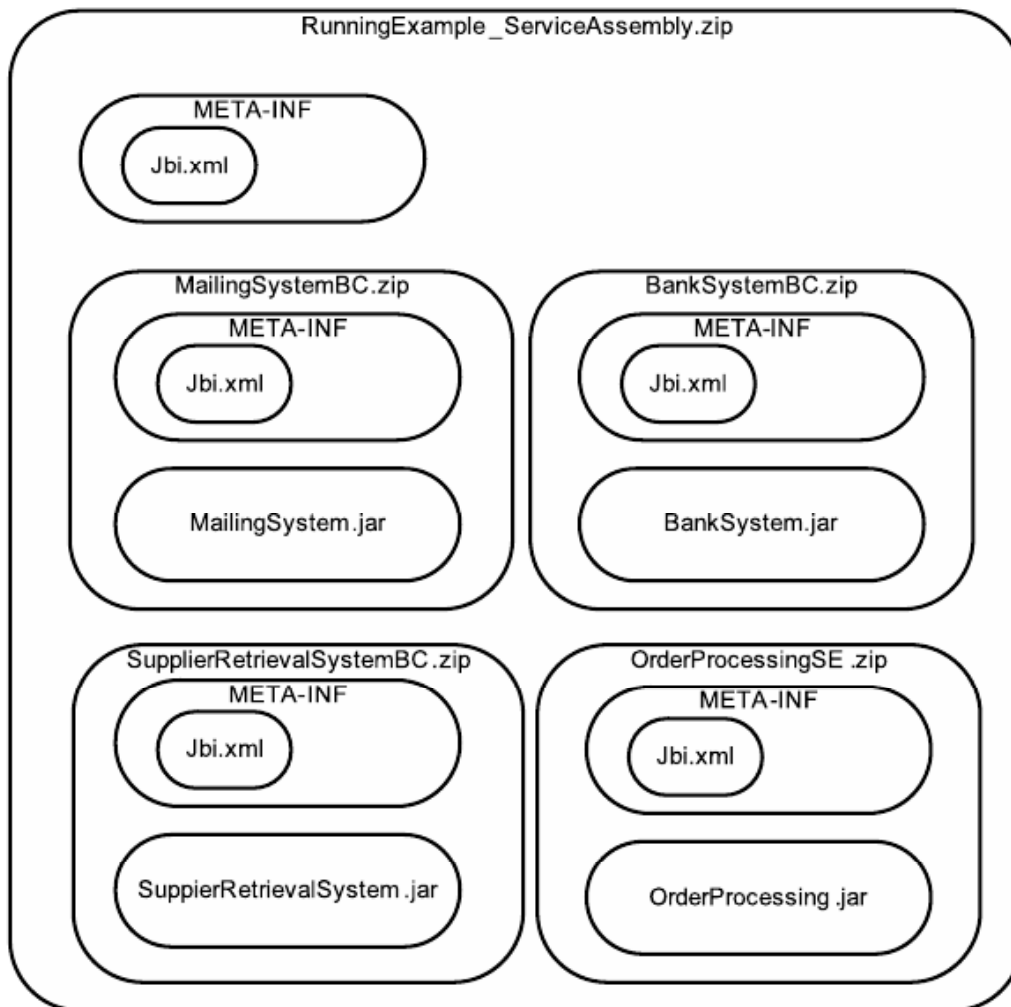


Fig. 20: Component Distribution model of JBI taken from [78].

- The Component context is provided by the JBI to the component during the starting and stopping of the module to add and remove services by the module.
- This *must* be supplied to the component during component initialization. This provides access to JBI-provided resources during normal execution of the component.
- Class loading and error indication. JBI has to ensure that the class loader used to create instances of the component-supplied Bootstrap and Component classes is set appropriately. This gives components access to a standardized class loader hierarchy, and therefore to component, shared and JBI libraries. Exception classes are provided by JBI to allow the component to distinguish various error types.

Additional interfaces for managing the module have to be provided by the component. These systems should further be touched at the section management (3.5.4) [8].

3.5.3 Messaging

The messaging with the Normalized Message Router (Enterprise Service Bus) (see section 2.3), the way messages are routed and the normalized messages is the second part of JBI beside the component framework.

Normalized Message Router (ESB) The normalized message router (NMR) or enterprise service bus receives message from the JBI components and routes them to the appropriate component. This model allows the NMR to perform additional processing during the lifetime of the message exchange.

All concepts of the NMR are based on the WSDL concepts which structure how services are modeled. The WSDL contains all information of how JBI components have to interact. It abstracts the service model based on operations which are defined as message exchange patterns (see section 3.5.3). A collection of related functions is called an interface. A service implements such an interface. Furthermore a service has one or more endpoints which provide access to the service over a specific binding (protocol). All these concepts are provided to allow components acting as service producers and consumers to interoperate with one another in a predictable fashion whereas all of the coupling done between them is described in WSDL [8].

Message Routing JBI exactly supports four types of Message Routing Patterns which should be described in the following chapter in more detail. The patterns are quite important for JBI because they define how the components communicate with each other.

- **One-Way (In-Only):** A One-Way message pattern is a simple request by a consumer to a provider with no error (fault) path provided.
- **Reliable One-Way (Robust In-Only):** A Reliable One-Way message pattern describes a request of a consumer to provider, where the provider may respond with a fault if it fails to process the request.
- **Request-Response (In-Out):** A message pattern where a request is issued by a consumer to a provider with expectations of response. The provider has to respond with a fault (if it fails) or the expected response.
- **Request Optional-Response (In-Optional Out):** This pattern describes the same behaviour as the Request-Response but with the difference that a response of the provider is optional [8].

Normalized Message Exchange The key concept of the JBI/ESB is to route normalized message exchange from one component to another. Every external message protocol has to be normalized (by the binding components) before it can be sent to another component. Binding components and Service Engines communicate with the ESB via a DeliveryChannel Object, retrieved by the ComponentContext retrieved during the initialization of the module. This way provides a bidirectional delivery contract for message reception and delivery, which is presented in figure 21.

Figure 22 describes the way of a message from an external Service via WS-I to a Binding Component, sending the message via the NMR to a Service Engine. The external service consumer

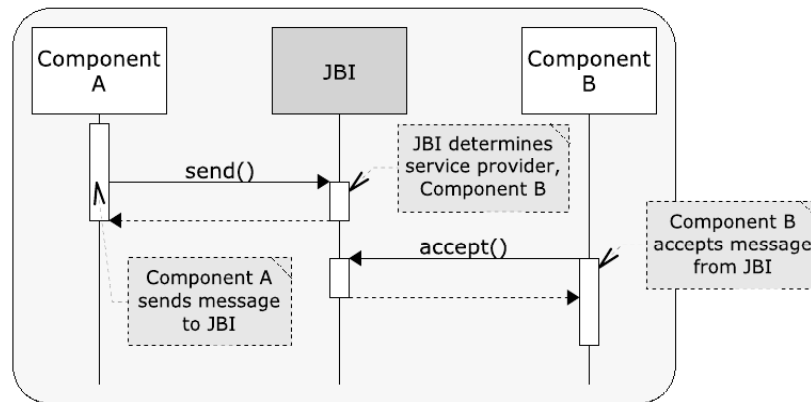


Fig. 21: High-level message sequence chart of the normalized message exchange system of JBI taken from [8].

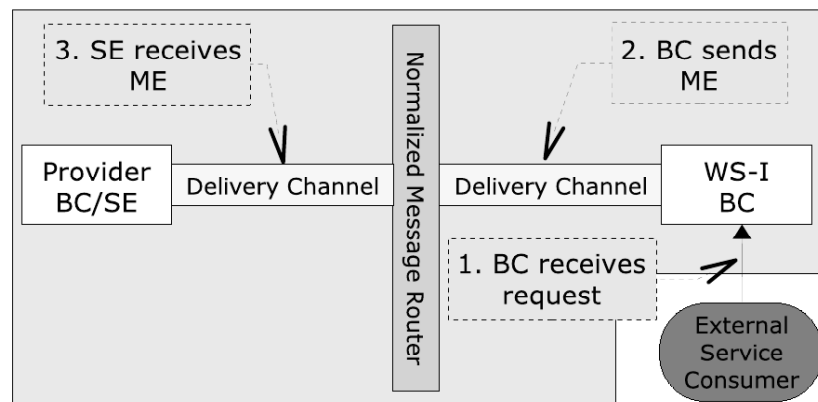


Fig. 22: Routing of external messages taken from [8].

sends a service request to a Binding Component in a specific protocol and via a specific transport. The Binding Component could convert the request to normalized form, as described in a WSDL model that describes the service exposed by the endpoint exposed by the Binding Component to the consumer. After the conversion, the Binding Component creates, according to a Message Routing Pattern, a Message Exchange (ME). Via the DeliveryChannel, this message was sent to the NMR. The Normalized Message Router selects a suitable receiver for the request and pulls the ME into the DeliveryChannel of the receiver. The Service Engine must accept (pull) the ME from the DeliveryChannel.

Reversing the flow of the ME, as shown in figure 23, changes nothing, except that the request to the Binding Component is denormalized instead of normalized [8].

A normalized message consists of three main parts as described in the JBI specification [8]:

- **Payload:** a XML document that conforms to an abstract WSDL message type, without any protocol encoding or formatting.

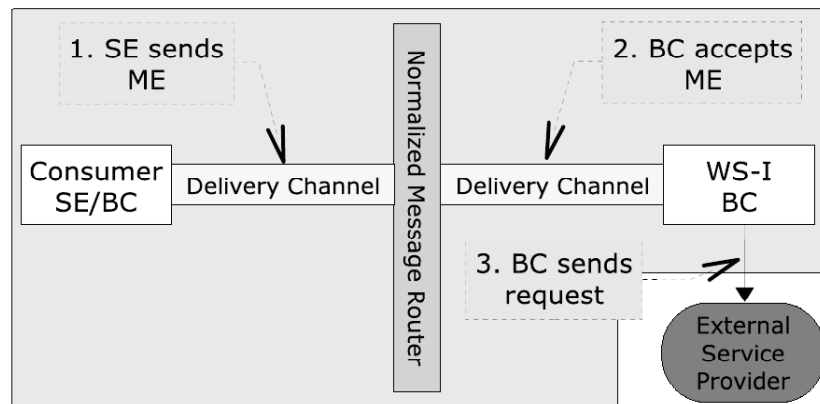


Fig. 23: Routing of internal messages to external receivers taken from [8].

- **Message properties (Metadata):** holding the extra data associated with the message, gained during the processing of the message. Such properties can include security information, transaction context information, and component-specific information.
- **Message attachments:** a portion of the content of the "payload" (content) of the message can be made up of attachments, referenced by the payload, and contained within a data handler that is used to manipulate the contents of the attachment itself. Such attachments can be non-XML data. The normalized message provides interoperability between components, using abstract WSDL message type definitions .

Programming Model As described in section 3.5.3, the programming model of the Message Exchange shall be not described in detail, but shall only give a feeling about how things are done with JBI. A DeliveryChannel represents the bidirectional communication pipe used by Service Engines and Binding Components to communicate with the NMR. The DeliveryChannel can be retrieved via the ComponentContext (see section 3.5.2) during the initialization of the component. The DeliveryChannel form the API contract between consumers, providers and the NMR. Every component is only provided with a single delivery channel, wherefore it has to be supported the concurrent use of a given channel from multiple threads [8].

3.5.4 Management Component(JMX)

A JBI implementation offers the possibility to administrate its components. The specification for the JMX (JMX itself is not in the scope of the paper, but can be found in the book by Steven Perry [82]) component interfaces exist to manage these needs. They include the ManagementBeans, Installation-ServiceMBean, InstallerMBean, DeploymentServiceMBean, LifecycleMBean and the ComponentLife-CycleMBean. With the help of these interfaces, a UI could be presented to the administrator which strongly reduces the complexity for administrating the components. In a nutshell, the tasks of the Management Component can be summarized as the following:

- Installation of engines and bindings (components)

- Life cycle management of components (start/stop controls)
- Deployment of component artifacts to engines and bindings that support dynamic additions to their internal execution environments. For instance, the SE is an XSLT engine, and the artifacts are new XSLT style sheets to be installed in the container.
- Monitoring and control.

As already said, all management tasks were realized with the help of JMX. This is very common in Java environments. Nevertheless the publishing of own management functions is disputable, because a JBI environment can not recognize the generic semantic according to the added function [78].

3.5.5 Integration Standardization Beyond JSR 208

It is a well known problem that it is not enough to standardize the interfaces between the communication layer and the components. If the interfaces for standard components are not standardized, a vendor lock-in to the artifacts provided by third parties would happen. Some of the examples of these artifacts are named by the JBI Vision Technical Whitepaper [83] and are cited in the following paragraph:

- **Process Engine:** The Process Engine consumes business process maps. An example is WS-BPEL, a proposed standard that has been submitted to OASIS for consideration as a potential business process standard. WS-BPEL is a web service sequencing language that defines the "conversation" flow of Web Services Description Language (WSDL)-described message exchanges. Additional work will be required to define broader business process management standards.
- **Business Rules Engine:** The Business Rules Engine would consume business rules as defined by enterprise business analysts. At present, there are no standards in this space.
- **Document Transform Engine:** The Document Transform Engine consumes document transform map definitions that tell the engine how to go about transforming a document from one format to another. eXtensible Stylesheet Language Transformations (XSLT) and XQuery are standard ways of transforming XML documents. Similar types of standards are required for other types of documents.
- **Partner Management:** The Partner Management Engine would consume artifacts that define the document type, communication protocols, authentication, and other key items for the interchange of documents between trading partners.
- **"Other" Engine:** This could be other types of engines (such as EDI transform, EII, etc.) not yet envisioned, or legacy engines plugged into JSR 208 for backward compatibility and customer migration. To realize the full Java Business Integration vision, the market will need standards for each of these artifacts that will ensure portability of integration solutions and people skills across different vendors integration implementations.

3.5.6 Problems

JBI does not come with new problems in its specification except the already known problems:

- JBI is only defined for Java which reduces its usability for Service Orientated Architecture (SOA) a little bit, because SOA services are distributed within many different languages and systems.
- Also, JBI comes with all advantages of an Enterprise Service Bus (ESB), but also with its disadvantages as overhead, new knowledge which is required and the higher administration effort.
- The last problem is that JBI is quite invasive to the code. This means that modules written for JBI have to hold references to specific libraries. This makes it impossible to use the code, without any changes, in other, non JBI, environments. This problem occurs, because the complete life-cycle model is managed through code. Further messages (service calls) are also directly done at the code and have to be handled by the code.

Apart of these minor problems, where some of them can be handled, as described at chapter 5, it is very simple to create modules for JBI and to work with the specification.

3.5.7 Products

As an open standard there are a lot of implementations available for JBI. The following list contains some of the best known and most commonly used enterprise service busses implementing the JBI standard:

- Open ESB (<https://open-esb.dev.java.net>)
- Apache ServiceMix (<http://servicemix.apache.org>)
- Fuse ESB (<http://open.ionac.com/products/fuse-esb/>)
- PetALSh (<http://petals.objectweb.org/>)

Open ESB is a JBI implementation directly developed by SUN with tool support from the NetBeans IDE. This ESB is directly implemented into Glassfish and comes with the standard distribution available by Sun. Apache ServiceMix is a JBI implementation by Apache with a lot of components already existing. Fuse ESB developed and published by Iona. This is an Open Source ESB based on Apache ServiceMix and enhancing it. PetALSh is another JBI based ESB, that is hosted by OW2 (former ObjectWeb).

3.5.8 Conclusion

JBI has two very huge advantage. First of all it is designed asynchronous and perfectly fits the typical architecture requirements for a SOA respectively for EIA. On the other hand, the more important advantage of JBI is that it is defined as an opened standard. This allows implementer no to lock into any vendor specific technology and develop their components for every JBI ESB the same way. Furthermore, JBI defines a complete component and management system allowing implementors relaying on a stable and opened definition. With JBI a big step in the right direction was done by Sun for a unified services providing model in the EAI/SOA world.

4 Comparison of Component Frameworks

This chapter tries to characterize component frameworks from the view of software developers based on the criteria formalized in section 2.4. This shall enable developers to find the framework most suitable for given specifications without knowing much about the different frameworks in advance.

4.1 Language Support and Platform Compatibility

The exact definition of this criterion can be found in section 2.4.1.

The Springframework itself is developed for the Java developing language for the full Java Runtime Environment (JRE). Therefore it can only be used, without modifications, in this environment.

The Springframework is, instead of many other frameworks and middlewares, including those discussed in this paper, more an attitude than a real model. Therefore, the port to any programming language and environment, supporting some basic possibilities as XML/XSD verification and tools, object orientation, a meta model, reflection and proxies can be used. As first thought, a Springframework for the .NET framework comes into mind, which does really exist. Also a framework for python is at the moment in development. Nevertheless, all of these frameworks only run on the full platforms at the moment. No implementation for the .NET Compact Framework for mobile devices running on Windows CE exists at the moment or is planned for the future.

In comparison to Spring, EJB is open in the same way. Nevertheless, EJB could not be compared to Spring. EJB is a specified programming model, which is based on the Internet Inter-ORB Protocol (IIOP) which is open for other languages too. EJB is not bound to a full blown J2EE platform. There are also supports for running EJB based applications in OSGi and in standalone applications. Furthermore, there are implementations for Java ME. Therefore, EJB is surely the component container framework supporting Dependency Injection which can be run on the most different platforms.

OSGi was designed for running in single Java Virtual Machines (JVMs). Therefore, it can run on all Java platforms — for example on Java ME as well as on Java or J2EE. Although it is propagated, the specification of OSGi could not be used for any other language than Java at the moment. First of all, the specification is directly made for Java. This means, that special functionality of Java is used which may not be available in other programming languages. The only language which could become close to the approach of Java is the .NET Common Language Runtime (CLR), comparable to the Java Runtime Environment (JRE). The concept why the CLR can not be used for implementing the OSGi specification should not be explained here in detail, because it would go beyond the scope of this paper. In few words reasons for this are the way the CLR loads the context of different assemblies and how these Application Domains work and communicate with each other.

JBI and its concept of an Enterprise Service Bus allow many different components on many different JVMs to communicate with each other. Nevertheless, it has two big disadvantages. First of all, it requires a full blown J2EE server to be able to run. On the other hand, it can only work in combination with Java. No other languages can be used to write components for JBI and no other platform than J2EE can be used.

SCA is the most liberal platform of all presented component frameworks. Unlike all other frameworks,

it is explicitly designed to be used from different language implementations and from different protocol bindings between them. It is not defined by the specification which language has to be used to implement the SCA runtime. Therefore it can run theoretically on every platform and on every device. It only depends on the implementation.

4.2 Learning Curve

The exact definition of this criterion can be found in section 2.4.2.

Spring, started as a community open source project and is quite easy to learn. It comes with a very rich, task orientated documentation. Knowing Java means to be able to use the base concepts of Spring within minutes. Every new part of Spring can be used within the same time, because all of them are based on the same concept of the IoC-container, which builds the heart and soul of Spring.

To master Spring, quite more knowledge is required. The problem on mastering Spring is not based on any difficulties to use it. Its only based on the problem of the huge size of the framework. With all its derivatives as for example Springframework, Spring-Web, Spring-Webflow, Spring-DM, Spring-JSF, Spring Integration, Spring Batch and many more, Spring offers such a rich support for programming that simply knowing all the possibilities is the problem, not to use the functionality once it was learned.

EJB itself, which means only its four primary bean types (entity beans could be count to JPA and not have to be count to EJB), is quite easy to learn and to use. The model is very simple to read and to write and has a rich support in most IDEs. Therefore, mastering EJB is not too hard because the top of EJB is JNDI which is quite easy to use. Every further communication between EJB and other components has to be searched in other technology branches of the J2EE environments.

In a nutshell, EJB is a technology which is quite easy to learn and still not very hard to master. The concepts can be learned within a view hours and very good development environments like for example netbeans help the developer to master EJB without having to know it in all details.

In the OSGi component framework, it is not too hard to learn the basics of how to run a component in an OSGi runtime. Nevertheless it is very hard to master OSGi and understand all its concepts. Because of its own way of working with context class loader and restricting which module is allowed to see which libraries of which module, things as reflection, context class loading and so on are quite complex and complicated. The core problem is that many libraries are based on context class loading as Hibernate, Log4J, Jakarta Common Logging (JCL) and so on.

JBI is really simple to learn. To write the first Hello World program can be done, in an environment as netbeans, in about five minutes, and should not need any longer in any other IDE. But JBI is not easy to administrate and can be complicated when writing huge projects. To administrate all possible message types, the interfaces, the binding and the service engines can become quickly a very complex task. To understand all details of the fine grained system would be another burden. JBI is an option for Java exclusive environments and has its warranties to be used. So, developers or project leaders have to calculate some time before JBI could be mastered at a corresponding level for a big business application with many integration tasks.

SCA is a great tool for building integrated applications in service orientated architectures very quickly.

Because of the very simple notations and the rich definition, not more than 10 minutes are required to create the first project. IDEs as Eclipse fully support SCA in their SOA environments and provide a rich toolset for working with the specification. Because of the definition only implemented in XML, it is possible to develop in an iterative way, meaning that it could be started quite quickly with the development without having problems with all details of SCA. Anyway it is also not too hard to master SCA because the possibilities provided by the SCDL (see section 3.4.2) are quite rich and allow everything but there are not too many tasks which can be reduced to only those really required for the project.

4.3 Security

The exact definition of this criterion can be found in section 2.4.3.

Spring can be used for J2EE development as well as for Rich Client Applications (RCA). For RCA, Spring offers neither a policy framework nor a security system. Bean containers can not be protected (by the framework itself without any customized extensions) against beans from packages not signed by a vendor, or any other kind of identification. For J2EE applications, SpringSource provides the Spring-Security framework [84]. The framework is based on Acegi Security and extends it in many ways. Spring-Security offers support for OpenID, Windows NTLM, support for JSR 250 [85] and many others. Standards such as role-definition and so on are also included.

EJB offers an annotation based support for security and role definitions specified by JSR 250 [85].

The OSGi framework provides a very rich security model based on the Java 2 security architecture [86] and extends it further [28]. It provides the infrastructure to manage and deploy applications which have to run in fine grained environments. Bundles can be signed to only run with other signed bundles, resources can be fine grained restricted in their usage and code restrictions can be defined.

The JBI specification itself [8] defines no security model for components and modules. The entire security in JBI is granted via the Normalized Message Context. Messages can be signed and/or encrypted. This means that modules working together can secure their provided functionality. Furthermore, definitions as WS-security are supported by JBI to defend the system. Messages allow to contain a security subject containing all the relevant information for a secured message. Vendor implementations as OpenESB or ServiceMix balancing the weakness of a missing component security system with their own proprietary implementation introducing the big disadvantage of losing cross vendor compatibility.

SCA defines a very fine grained policy structure in its specification through [63]. It defines how to encrypt or sign the communication between two components via a binding. Further more it defines how to use different security standards as WS-security. Message and transport level security support are provided by the Policy Framework specified at SCA.

4.4 Performance and Stability

The exact definition of this criterion can be found in section 2.4.4.

To evaluate and describe performance and stability of the presented frameworks is quite a non trivial task. Most of the frameworks offers more than one part which could be measured. Furthermore all of these middleware specifications/implementations work in different ways, so there is no standard a

comparison could be made against. It is tried to describe the way the framework works and what it means for performance. Simpler questions as "do smaller errors disable a whole system", or "how resistant is a system based on a framework to components that do not work as intended" are tried to be answered directly for each framework.

Spring consists of many components as could be seen at section 3.2. Nevertheless, every part of the framework is based on Spring's IoC container. Benchmarks for comparing the IoC container against a regular new statement in code are not quite fair because an IoC container manages much more than just the creation. Nevertheless an interesting, but not very scientific benchmark is given in [87].

The IoC container of Spring is built in a very robust way. Fails of beans do not affect the container in any way, because exceptions are caught and returned to the callers. If beans are directly requested from a container, an exception would be thrown if any error occurs during the creation process. In the case of an error during Dependency Injection, the exceptions are caught and logged and null is inserted to the requestor instead of the final bean.

Beside of the IoC and AOP support, Spring is not really a framework, providing functionality (could not be said cross-the-board but for the sake of simplicity it should be taken as if) itself. Spring provides the glue to connect code independently from the implementation of the modules. Therefore, measurements of performance and stability have to be done at the third party implementation more than at Spring itself.

For EJB, as usual, the same rules can be applied as for Spring. Although EJB uses another EJB container, the creation of the objects does not differ very much in time. Also failure of components do not affect the entire runtime.

OSGi, as a real component framework with a full-blown life-cycle and service management system, can be analyzed better than Spring or EJB on stability and performance. The specification (see [28]) defines components shall be handled that failed to load, updated or during any other life-cycle step. It is not as critical as in other frameworks as Spring or EJB, because the way is defined how components should be handled if they fail. The more important question is how components requiring on failed components shall be handled. Here helps the fine grained life-cycle model of OSGi. Components, as long all of their references can be resolved, can be resolved by the environment. So their published classes can be accessed. If a module cannot be started, the services are simply removed (or even not deployed) to the registry and are simply not available.

Performance could not be measured as it can be done with Spring or EJB, both coming from one main vendor (SpringSource for Spring and Sun Microsystems for EJB). OSGi is only a specification which can be implemented by any vendor. In few words, the service registry is extremely good performing. The class loading system of OSGi directly uses options of the virtual machine which works also quite well. Furthermore, the specification is created for restricted environments (to run on Java ME) and therefore the performance loss through OSGi can be ignored on full blown desktop environments. Nevertheless, all these statements cannot be said globally because of different vendors implementing the OSGi specification.

JBI and SCA have the same troubles as OSGi according to performance. Neither the JBI nor the SCA specification mentions performance in any word, wherefore it is left completely to the vendors to implement a well performing environment. It is clear that an Enterprise Service Bus cannot be as fast

as doing the same task directly. E.g. sending a message via the ESB could never be as fast as sending it directly via the Java Message Service (JMS).

Although JBI provides, in comparison to SCA, a complete life-cycle model for its components, it cannot be analyzed according to stability of the component deployment cycle. It will be out of the scope of this paper to compare different products about how stable their vendors implemented reliability and performance in their systems.

Finally it can be said that reliability and stability can be seen as a cost-benefit calculation. Using an IoC container (either Spring or EJB) would always cost performance and some reliability because the developer give up some of the type information stored at the object (explicitly if xml configurations are used). The benefit will always be to compose software which is very resistant against changes of requirements. Also full-blown life-cycle systems as OSGi or ESBs will cost performance but decrease the work required as well as the configuration work and management work for the software.

4.5 Life-Cycle Management and "Plugability"

The exact definition of this criterion can be found in section 2.4.5.

As described on the homepage of JavaBeat [88], Spring provides a life-cycle model for Spring-beans. The process is started in the moment a bean is requested by the IoC-container and ends if the container is disposed. Nevertheless, the life-cycle of the bean can not be affected at runtime and is very limited for the creation and destruction process of a bean.

Spring itself does not allow any kind of plugability. For example, a jar file can not simply be taken and placed in any special folder so that spring will load it automatically. Of course, beans can be deployed at runtime via coding. This means that a module can merge application-contexts (see section 3.2) at runtime, but no default mechanism is provided by Spring.

Both, life-cycle management and plugability is neither the way Spring works, nor anything it was build for. Spring can be seen for assembling components at development time and increase reusability but not as an framework for managing components at runtime or change their structure when required.

Ejb, an IoC-container as Spring, provides exactly the same advantages and disadvantages as Spring-beans according to life-cycle management and plugability.

OSGi can be seen as the definition for plugability and life-cycle management in the Java world. Most of the specification is about how to manage different bundles (see section 3.1). A complete lifecycle, from deploying, enabling, starting, stopping and undeploying is offered. By default, the OSGi specification defines no automatical loading of bundles and delegates this decision to the vendor providing the implementation. Nevertheless, bundles can be started via the management console, which has to be provided by each OSGi implementation or by bundles themselves. Bootstrapping is not included into the OSGi specification. This step of installation is delegated to vendors themselves.

For Bootstrapping as well as for automatical loading, the Equinox implementation of OSGi, provided by the Eclipse Foundation, provides solutions for both problems.

The specification of JBI describes a complete life-cycle model including bootstrapping, deploying, starting, stopping, undeploying and uninstalling. The life-cycle has to be completely controlled by the imple-

mentation. The user has the possibility to add and remove components and to change their life-cycle state at runtime, as at OSGi via a management console. In contrast to the way management is specified in OSGi, as common in J2EE environments, JBI supports Management Beans which have to be supported by every application server.

Plugability itself is not provided by JBI — not in the common way as add a package and it will be automatically installed. Here work the same laws as OSGi. Such tasks can only be done by the management console, but vendors can implement solutions for this problem.

The SCA specification does simple neither provide a life-cycle model nor any kind of plugability. The specification simply describes which components exists, how features are described and how modules have to be handled. Any further implementations is open to the vendor. Therefore, any vendor can and has a different implementation. The exact model of implementation shall not be described here.

4.6 Reusability and Meta-Data

The exact definition of this criterion can be found in section 2.4.6.

Spring provide a complete Meta-Data model for components described in XML. No invasive code, as abstract classes, interfaces or annotations is required to bind the code to the Spring-Containers. Therefore, the code can be reused in completely other environments that even do not provide an IoC-Container. In those cases only the binding between the classes have to be rewritten, but the classes themselves do not have to be changed. Anyway, Spring provides the opportunity to program with annotations. This is not recommended if the code should be untouched, because the code is then bound to the Springframework.

Basically, EJB has a very invasive programming model. The regular way of writing EJBs is to use annotations as `@EJB`, `@Resource`, `@Stateless` and so on. Normally this does not matter because the EJB annotations are in the `javax` namespace which means that every distribution (except Java ME) has them included. Therefore, the code can be reused in any other programm, independently if it supports EJB or not. Only the binding code, replacing the EJB-IoC-Container, has to be rewritten. Furthermore, EJB provides a way of defining all Beans at the `web.xml` file of an webapplication. This avoids to bind the code directly to any framework and load any information required for EJB from the Meta-Data of the `web.xml` and the reflected class.

OSGi provides a invasiv programming model. Bundles have to be started at the code, services have to be used from the framework and have to be provided to the framework. Furthermore, services have to be activated and deactivated, befor and after they are used. This will create OSGi-references everywhere in the code. With some design decisions, these references can be minimized, but never completely removed (with OSGi techniques alone). If the service technology of OSGi is not used, the code is quite reusable. Every bundle is a `.jar` file which also can be used in any other Java application and the `manifest.mf` at the `META-INF` folder does not bother other runtimes than OSGi.

JBI has almost the same problems with code reusability as OSGi. Comparing the models, they are very similar. Access for the framework to the code is provided with a very small peace of meta-data described in a file located at a `META-INF` folder and interfaces describing methods needed by the middleware. The complete communication with the ESB is described in code and not in meta-data,

which makes the code completely coupled to JBI. Nevertheless, it is the same as with EJB. Because the interfaces for JBI are provided in the javax namespace, each non Java ME implementation provides the classes and the code can run even in non JBI environments.

SCA provides a very rich meta-model. From security to bindings and implementations everything is described as meta-model. Therefore, there is a very low binding to the code, but there can be a binding to the code, depending to the programming language used. In Java, a direct binding happens, because annotations are used. In C++, the annotations are provided as comments which means that the code is not directly bound to SCA. These bindings can not be reworked (with SCA specifications).

4.7 Adaptability and Control

The exact definition of this criterion can be found in section 2.4.7.

Spring offers a very mighty control mechanism for the developer. As already described at section 3.2, the IoC-container is the core of the framework. That is the place where a developer wants to listen and act as required. Beside of a very good and detailed log against the Jakarta Commons Logging API [89] Spring allows the plugin of processors at various places of the creation process of beans and beanfactories. This method allows to control and listen the creation process of each bean very deeply and manipulate it whenever required.

Spring in its entire architecture is build for flexibility and scalability [39]. A developer can chose from a rich selection of first class supporting tools starting from the IoC-container alone with 200kb to a collection of different extensions to the container.

Spring can entice developers to use it as the Golden Hammer as described in the book "Antipatterns: Refactoring Software, Architectures, and Projects in Crisis" [90] which nevertheless may fit in this situation because of the very fine grained scalability and mighty mechanisms to control most steps of the framework.

EJB is a very scalable platform according to many points described by the authors of [91]. First of all, it can be scaled according to the choice of implementor. There are many open source and commercial implementations available for the EJB3 container listed at section 3.3. Secondly, the EJB specification offers remote as well as local interfaces. With this differentiation, it allows the developer to choose where to run his beans. Remote beans can reside in different Java Virtual Machines (JVM) but the calls between, because of the additional network communication layer, is much slower than to beans marked with the local interface meaning that they reside in the same JVM as the caller and the network communication can be passed. Therefore, EJB alone at specification level (independent from an explicit implementation) provides, with the right architecture and mixture of local and remote beans, a very high level of scalability right out of the box. Different parts of business logic (EJB Session Beans) can be located on different servers and are connected via JNDI. According to this, EJB allows the developer full control about how and where EJBs are deployed. The combination of scalability and the five base types of beans allow developers a very high level of adaptability.

To make a specific decision on the control of the beans is much harder. That's because EJB is a specification and does not exactly specify where developers are allowed to hook the flow of control and where to change it. Therefore, this question has to be done per implementation of the EJB-container

which is quite far beyond the scope of this paper. Nevertheless, many implementations as JBoss allow to hook quite every process of the life-cycle of EJBs. JBoss as an MBean Server, which means that every unit is deployed as an Management Bean, allows a very controllable infrastructure.

The OSGi specification allows something called the Extender-Model [92] and a complete listener model [28]. The listener model is well known from other systems as asynchronous events could be registered to the framework allowing to get a deep insight into the communication. Started at the Life-Cycle model and ended by the Service-Registry, mostly any action can be hooked. All these actions are fine grained in their security requirements allowing to give full insight to the developer without risking the overall security of the system.

The Extender-Model is something the regular Windows or Unix user are familiar with. For Mac users its nothing new. The model could be seen as externalized control modules to the life-cycle of the OSGi specification. Extender are synchronized listener to the life-cycle allowing them to do actions to modules when they are required, e.g. before a module is installed or before a module is started. An example of such an extender for servlet registration is given by Peter Kriens [93]. The most popular example for an extender could be found at SpringSource at the Spring Dynamic Module (formerly known as Spring-OSGi) project [46].

Furthermore, as could be seen in the OSGi core specification [28], OSGi is built for scalability and adaptability. Only some core bundles are really required to run OSGi. These bundles can be fully adapted by the extender-model without ever touching the original implementor code. From this point on, bundles and services can be freely chosen by the developer, building a platform exactly according to his/her needs.

The specified component-model, mixed with the extender and listener model make OSGi to one of the most adaptable platforms described in this paper.

JBI has a completely different model than the other frameworks investigated till now on adaptability and control, which can be seen in combination with the Enterprise Service Bus. The idea behind an ESB is to provide all technologies so that the user has not to adapt it. Therefore, everything which could be needed by a developer should be already in the bus.

For the point of control, JBI completely relies on MBeans, which can be already seen at section 3.5. According to the specification [8], management beans should be used to control the deployment and life-cycle of the modules, but it does neither provide a way to control the implementation of the ESB directly nor to extend modules anyway as OSGi do for example.

JBI is a well defined platform providing everything to the user they need to create their system. Freedom to adapt the system as required by developers is not given and the control is minimized to the control of the life-cycle of components. Nevertheless, this should be more than enough for every project requiring an ESB. The customization is done by the modules installed to the platform.

The SCA definition allow absolute no option to control the way the implementation works, nor provides it insight into the communication process. All these tasks are delegated to the vendors providing implementations of the specification. Therefore, no specific decision about this could be made by this paper.

5 Interconnection of different Component Frameworks

In the following ten sub sections, all component frameworks, presented in this work shall be analyzed in respect to ways how they can be combined and which synergy effects can be achieved. Furthermore, it shall be analyzed why someone would like to combine these frameworks in such a combination.

5.1 Spring – Enterprise JavaBeans

Spring as well as EJB are container frameworks for DI/loC (see section 2.3). Although they are competitive, there are possibilities to combine them. There are many possible ways to do so. Nevertheless all options work via JNDI (see section 2.3).

To inject EJBs into Spring beans and the other way round, internally the JNDI is used. The Springframework simplifies the use quite well. Beans to convert Spring to EJB are directly provided by the framework [39].

EJB does not provide any possibility to merge Spring and EJB. Nevertheless there are some workarounds to merge them. This happens via AOP and the direct injection of the registration of the beans to the JNDI [94].

Why should someone try to combine the EJB and Spring frameworks? There are many reasons for combining these middlewares. Some of the arguments are tried to be given by an article of Ramanujam A. Rao [95]:

1. Use Spring as a general loC-AOP container, and then integrate EJB components as regular beans to implement business functionality.
2. Use Spring as a provider of EntityManager for the underlying JPA implementation. In this case, Spring goes beyond basic container support to manage the state of the EntityManager and ensure compatibility with third-party JPA providers, automatic participation in transactions, exception handling, and so on.

Other reasons can be that legacy code of either Spring or EJB applications shall be brought together to reuse code written one time. In most cases, it may be better to do such claims as wrapping the already written components and bind them to an Enterprise Service Bus framework as JBI or SCA, but surly there are cases where an combination of EJB and Spring are proved for entitlement.

5.2 OSGi – Spring

For Spring, the OSGi definition is one of the future frameworks for lifecycle management and service administration. Therefore, OSGi is quite well supported by Spring. SpringSource offers a complete middleware to reduce the complexity of OSGi and its classloading system. Because of the special classloading of OSGi (see section 3.1) and many frameworks as Hibernate, JAX-WS, Ibatis and others using context classloading via reflection not knowing about the asseblies required, developing with OSGi is such a complex task, especially because OSGi does not define how context classloader should handle during service calls. Another problem with OSGi is that working with services is a quite complex task. To register services they have to be registered at the startup of bundles binding the code quite

strongly to OSGi. To call services, they have to be retrieved and activated when they are needed and deactivated and unregistered when they are not needed any longer. All these tasks are handled by Spring-DM (Dynamic Modules, which was formerly known as Spring-OSGi). Via the framework, beans could be simply registered as services and imported by other bundles as beans. Therefore services could be used, through the entire system, as regular Springbeans. The exact description of Spring-DM could be found at the official specification [96].

OSGi itself does not provide any mechanisms to load springbeans as services. Nevertheless it provides many interfaces to do such work, as the Springframework proves.

The cooperation of Spring and OSGi has a very rosy future because SpringSource releases a new Application Server (Spring Application Plattform) which could be seen as a combination of the OSGi framework Equinox, the different Springmodules, described at section 3.2, and the Apache Tomcat server [97].

Combining Spring and OSGi provides the best models of both worlds. Spring has nothing in common with OSGi. Spring is a framework building around an IoC-container not trying to control the life-cycle of modules and the class loading of them (see section 3.2). OSGi controls the life-cycle, class loading, security and many more aspects of modules (bundles), as described by section 3.1. Therefore Spring simplifies the OSGi model for developers a lot. It allows Dependency Injection of beans into modules, where beans could be services or entire sets of services. The entire starting and stopping of modules can be done by Spring according to definitions in plain SML. All those and many more advantages could be found at official specification [96].

5.3 OSGi – Enterprise JavaBeans

EJB can be used in OSGi via bundles as OpenEJB [98] or EasyBeans [99]. But nevertheless these techniques just allow to use EJB technologies in an OSGi environment but does nothing comparable to Spring-DM. This is simply based on the fact that OSGi and EJB simply work on completely different systems.

Using EJBs in OSGi means to use only the life-cycle management capabilities of OSGi but completely ignores its servicesregistry. Although OpenEJB as well as EasyBeans are deployed as OSGi-bundles, which mean that they will run in an OSGi enabled environment as Equinox or Apache Felix, they still use JNDI to deploy and retrieve the beans. Although it seems to be annoyingly not to be able to combine OSGi and EJB, it is not too worse, because you can use both in each other. This means that the advantages of both worlds could be used; the deployment/life-cycle model of OSGi and the JVM crossing concept of JNDI used by EJB.

The advantages of OSGi and EJB are clear to see. OSGi is used to control the life-cycle and EJB is used to control the services. Nevertheless, one big disadvantage occurs in this combination. Compared to the Spring-OSGi combination at section 5.2, a EJB-OSGi combination would not use and wrap the service implementation of the OSGi implementation. It would override it and use JNDI instead to combine its beans. Therefore, surely some advantages of both worlds can be used but also some advantages will be lost.

5.4 Service Component Architecture – Spring

Using SCA in Spring is quite a simple task. The SCA specification comes with a definition of how to use the Spring-Framework in an SCA environment [73]. Via SCA, Spring-beans can be directly provided as services to other components. References can be provided as beans to the Springframework. The Spring container does all the injection work as can be seen in the Spring Component Implementation Specification [73].

SCA provides, with the help of the Springframework, a very elegant option to connect applications with a completely not-inversive approach. The Spring framework adminstrate the beans in its container and works the completely DI approach while SCA does the connection between the components.

Spring could not do anything directly to SCA which is reasoned by the fact, that Spring and SCA are build for complete differntly tasks. Therefore the best parts of both frameworks can be used; the concept of binding software components in an very simple, declarative way and Springs ability to do DI in an completely non-inversive way.

A combination of Spring and SCA bring huge enhancements to the code developed for SCA. The code is no longer bound to the annotation definitions of the vendor. Dependency Injection can be done to Plain Old Java Objects (POJOs) allowing to let the code know nothing about SCA. This has two further advantages. First of all, components not developed for SCA but with Spring can be used in an SCA environment without any changes to the code. Finally, it allows also components developed with the Springframework for SCA to be used in any other Spring-powered environment or with any other component-frameworks supporting IoC of POJOs.

5.5 Service Component Architecture – Enterprise JavaBeans

Also EJB can be combined quite easily via SCA as described in the Java EE Integration Specification [72]. Because of the existence of the specification directly available at the SCA specification, many vendors have already implemented EJB bindings for SCA.

SCA offers very good possibilities for EJB. Enterprise JavaBeans offered services can be directly bound to any binding. Via SCA, references can be defined, which are directly imported to missing EJB references.

Therefore, SCA is a perfect partner for EJB. It offers many possibilities to bind already existing EJB modules to the integration environment.

As with Spring, EJB does nothing to SCA which is based on the same reason as at Spring.

SCA brings a lot of enhancements to EJB. In some points, SCA does the same to EJB as to Spring. Nevertheless, it has to be mentioned again that EJB does not work against POJOs as Spring does. Therefore it does not bring advantages at the same level as SCA brings to Spring, but it allows a very easy option to build standard EJB applications and enrich them with different components created in different programming languages in a very simple way.

5.6 Service Component Architecture – OSGi

Neither OSGi nor SCA provide a direct way to use each other. Nevertheless, this combination is one of the most powerful: the simple way of SCA of providing services to other components in different JVM or other implementations combined with the service registry of OSGi and its lifecycle control.

Although neither of each specifications provide a worktogether, neither of them prohibit it. Therefore, some projects, as for example the Newton project [100] provides exactly this behaviour. The Newton runtime is implemented as an OSGi bundle and can be run in any OSGi runtime. This allows OSGi services to be deployed to other JVMs at which the Newton middleware is deployed in an OSGi framework and allow access to EJB, web services and many other technologies provided by SCA.

It turns out that OSGi becomes more and more the standard programming model for components. Many very popular applications as Eclipse and the Spring Application Server as described in section 3.1 and 5.2 are using OSGi. Its very easy to enhance libraries written for an “plain” Java SE, J2EE or Java ME platform to integrate to OSGi, because the only work to do is to extend the manifest-header already describing jars. Therefore, OSGi and SCA are a perfekt match.

OSGi already describes most of the things required by SCA to export or import services. Furthermore, the OSGi platform has also already a broad policy framework which can be easily integrated to SCA. Defining the bindings can be such an easy task as extending the headers of the manifest files.

5.7 Java Business Integration – Enterprise JavaBeans

EJB can be integrated very easily into the JBI runtime as a Service Engine. Tools as JBI4ejb allow such an integration. This is quite an advantage for servers which does not support JBI integration. Therefore, direct integration to JBI is allowed via this tool.

Combining JBI and EJB adds many advatages to the programming model of JBI. As good as the idea of a normalized message system is, it has the big disadvantage of beeing quite complex at huge projects. EJB simplifies this programming model very much.

JBI does not offer a very simple programming model. To reduce all services to messages is very advantageous as seen of a SCA sight, but very disadvantageous for developers trying to keep an overview over their components. Therefore, EJB can simplify the model quite a lot because for developers it is still the old way of programming and JBI does the rest. Finally, it can be summerized that there is a very high motivation to combine those two concepts to increase the usability for the developer and furthermore to use legacy code in new applications.

5.8 Java Business Integration – Spring

Combining Spring and JBI is not a simple task. As can be seen in section 3.2 and section 3.5, Spring and JBI does not have interfaces between them. There is no place to deploy Springbeans at runtime nor to define JBI to load spring beans.

Theoretically, it is possible to integrate Spring to JBI as an Service Engine, but there are no implementations or papers about this at the moment. Furthermore it can not be estimated how much work have

to be done to allow this.

Nevertheless, some implementations of the JBI standard as Apache Service Mix allow to deploy Spring beans into the runtime and retrieve elements of the runtime (other JBI services) as beans. This way of coupling has one big disadvantage: your implementation would be bound to the vendor offering Spring-Support, because the specification of JBI does not provide any way to do so. For most scenarios, this is still a good option, because JBI domains can be coupled very easily together as described in the JBI specification [8].

Spring would be very interesting to be used in an JBI environment because it can simplify many tasks actually delegated to the developer. Other motivations for combining these two frameworks are the same as for SCA-Spring (section 5.4). Otherwise it would provide the same advantages as described for JBI-EJB at section 5.7.

5.9 Java Business Integration – OSGi

OSGi can do much for JBI as it does for every other component framework too. By deploying three bundles in the OSGi runtime, JBI-packages build as OSGi bundles are analysed by the runtime and can be loaded to JBI in the moment they are activated. The plug-ins handling the runtime do everything to provide the deployed packages to the JBI runtime.

Of course, the advantages for combining those frameworks are the same as for SCA and OSGi as described in section 5.6, because neither of them provide an own way for component life-cycle management as detailed as OSGi does.

5.10 Java Business Integration – Service Component Architecture

As described by the OSOA on their homepage [101], combining JBI and SCA is a very useful task, although it is not required. As described by this source the relationships could be summed up in two statements:

1. Supporters of SCA view JBI as a Java Platform standard that can be helpful in implementing SCA on the Java platform.
2. Supporters of JBI view SCA as providing additional service metadata that can help simplify and standardize service composition within the Java Platform and between the Java Platform and other platforms.

In few words, JBI Service Engines can be compared to SCA implementation types and JBI Binding Components to SCA bindings.

More exactly, the possible bindings between JBI and SCA are described by Brian O'Neill [102]. According to this article, there are three different relationships that make sense, which should be described in the following list.

1. Already existing SCA runtimes such as Tuscany, already described at section 3.4, can be packaged as Service Engines and deployed to a JBI runtime. In this respect, JBI provides the backbone to the SCA runtime and make SCA contributions by making them available to networks,

capabilities and systems for which there is no SCA component implementation as SIP and XMPP. This approach would attach SCA to an Enterprise Service Bus with all the benefits that come from this.

2. The second possibility is about to add an JBI Component Implementation to SCA. This would introduce the advantages of allowing SCA to contribute to declare a JBI service unit as a component implementation within an SCA contribution. However such an implementation does not exist at the moment.
3. The most interesting solution according to Brian O'Neills article [102] would be to make JBI the runtime for SCA contributions. A translation layer could translate an SCA contribution into a set of service units deployed to binding components and service engines. This solution does not exist at the moment, but in such an idea, every SCA composite would be split as JBI Service Engine and Binding Component and likewise each service implementation in JBI composite application, a service unit would be deployed to a service engine as bpel, POJO and so on.

Although many of the possible implementations are not done at the moment, this is the way it shows that JBI and SCA are made for each other, although they do not really know about this at the moment.

Combining SCA and JBI would bring many advantages to a SOA. Services from both worlds could be used. Some parts are already designed in one implementation but not in the other. Furthermore, JBI provides, compared to SCA, a standardized ESB which could be used by SCA in combining them. More methods to combine those two worlds and more advantages are described by Dmitri Ilkaev [103].

5.11 Further Combinations

Although its quite logical that when two of the frameworks can work together, it would also work that more then two are combined. Nevertheless this fact should be discussed a little bit in more detail by this paper, because theres a very high motivation to combine those frameworks to create a better, more stable and richer platform for developers.

One of the most interesting projects in this sector is the Swordfish project provided by Eclipse. The core of the framework is up and running at the Deutsche Post, one of the largest logistic companies in the world, since 2002 [104]. Three years after that, in 2005, the SCA enviroment, developed by the Deutsche Post was offered to the Eclipse foundation with the idea to make it open source and integrate it with OSGi and JBI. Therefore, the Swordfish project was born which was introduced to the community 2007 [105].

The base layout can be seen in figure 24. Showing how the platform is going to be built. Swordfish will run at the client side and on the server side and should be ready about 2009 [105].

Because of the special architecture of Swordfish it would not be too heavy to also integrate EJB and/or Spring to the platform allowing the advantages of all five frameworks, described in this paper, togehter.

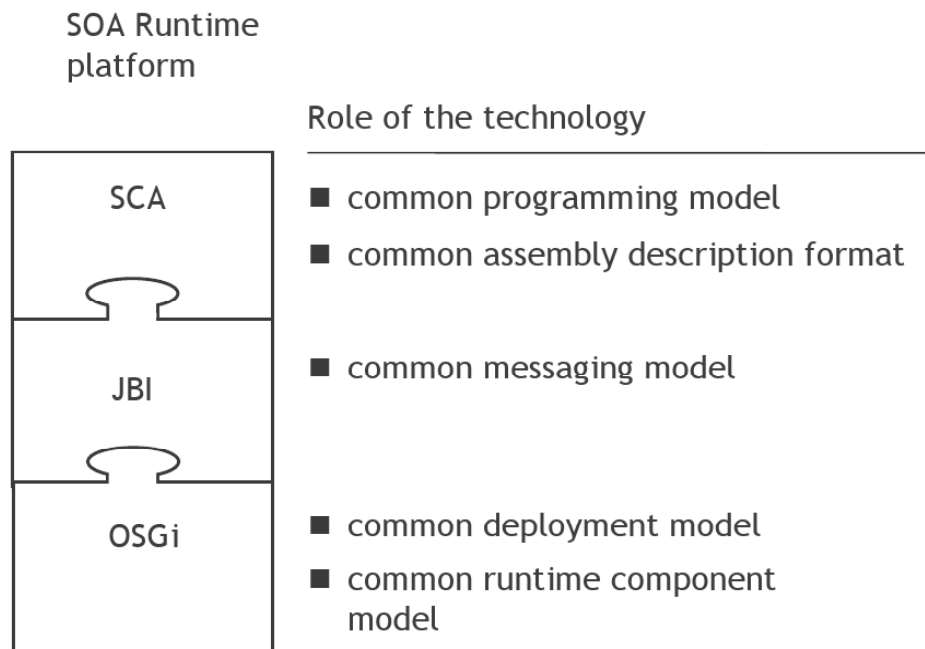


Fig. 24: Figure describing the base architecture of the Swordfish framework, the base model of the Eclipse SOA platform. As seen it consists of OSGi, JBI and SCA building on each other taken from [105].

6 Conclusion

The aim of this paper is to make a comparative analysis of the state-of-the-art Java component frameworks. To fulfill this, Spring (see section 3.2), J2EE/EJB (see section 3.3), OSGi (see section 3.1), JBI (see section 3.5) and SCA (see section 3.4) were discussed and it can be concluded that they look in most cases like real component frameworks. The following list should give a sum up of the results found about the different, presented frameworks and standards.

- EJB and Spring are very hard to compare and sum up, simply because they base exactly on the same concept. They are IoC-container and not really component frameworks according to section 2.2. They work in another way than regular component frameworks. Although it is only seen as a word, Spring or EJB beans cannot directly be compared with components. The IoC-container does not fulfill the work a real component framework would do. They are more a glue for different frameworks than a standalone framework.
- OSGi is something completely different. It can be stated that it is the only real component framework of all frameworks presented in this paper. It provides a complete life-cycle management as well as a way to secure components and automatically manage them. Furthermore, it also provides a way to expose services between components and to manage them.
- SCA and JBI are the third kind of frameworks presented and discussed in this paper. Although JBI and SCA provide some kind of life-cycle, they contain not even a quarter of the functionality OSGi provides. JBI and SCA are service and not component centric. They focus on how components can provide their services to the system and how these services can be extended by policy

frameworks and such things which were discussed in this paper.

Three different types of frameworks were compared in this paper. Nevertheless, they can be compared, apparently, against each other. This works because general enough comparators were chosen. It could be seen as if this paper tries to compare hats and shoes. Of course, they could be compared against factors as warmth, durability, simplicity and so on, but the question is not to choose either hat or shoes. The real question should be which shoes could be worn with which hat according to factors as look, aesthetics and so on. This fact could be seen very extremely at chapter 5 where most of the frameworks work quite well together.

This problem occurs because of the very ambiguity definition of the component concept. As seen, according to chapter 2, all frameworks match the concept of a component framework, allowing to compare them quite well. Nevertheless, all of them have very different specialism in the broad field of the concept of components, as life-cycle management, service management and so on. Therefore, it is possible to choose the five most popular Java component frameworks without being able to find a competition between them, except for Spring and EJB.

The five frameworks were chosen because they all have factors in them similar to component frameworks as defined in section 2.2. This paper does a great job explaining the different frameworks, comparing them against each other and discussing about possible ways of combining them. Exactly here, further work can be done and this point should be studied at a greater depth than this work can provide.

Nevertheless, all these frameworks add a great value to programming itself. Component oriented programming, including life-cycle management, dependency injection and component and service binding management reduces the dependencies between components, increases the manageability of components and adds simplicity to code. Whatever middleware is chosen, some of these values are added to the software. Combinations would also increase the complexity, but the effort can be seen multiply times as value.

Finally, it can be said that all of the described frameworks are a blessing for software itself and for the developers at specific, but there is no rose without a thorn. The development of component frameworks is still at the beginning. Many of the specifications such as SCA or JBI are still at version 1.x. OSGi is already at release 4.1, but only a very small amount of requests find their way into each final release. EJB is matured while Spring is only a de-facto standard with not even a specification. The frameworks are not different enough which leads to the misinterpretation that all of the frameworks can be used for the same tasks and therefore be replaced with each other.

All frameworks and their specifications are enhanced in a continuous process, bringing them closer together each day and make the use of them easier. At the moment, it cannot be decided if all of them will survive the hype around frameworks such as Spring or OSGi. Maybe, all these technologies lead to the final target of choosing software components completely freely from different vendors and combine them in any way as dreamed by software engineers for more than 20 years, or all of them will be forgotten at the next major step in technology evolution. It will be seen what is made out of all the ideas and technologies and which will survive.

List of Figures

1	Simple graphical representation of a software component: A single and mobile entity with "plugs" where other entities can bind to the component or where it can bind itself to other components taken from [1].	6
2	The architecture of the OSGi component framework taken from [28].	15
3	The layer interaction model of the OSGi component framework taken from [28].	16
4	The class importing scheme of the OSGi component framework taken from [23].	18
5	The life-cycle model of the OSGi component framework taken from [28].	19
6	Illustration of the Hibernate problem occurring in OSGi environments. A bundle A loading Hibernate (C) has the Problem that Hibernate (B) do not know about the contextloader of A and other bundles referenced by A (B).	22
7	Modules composing the Spring framework taken from [35].	24
8	Assembly model of spring to configure Plain Old Java Objects (POJOs) taken form [39].	24
9	Logical architecture of the J2EE 5 Platform taken from [54].	32
10	Life-cycle of a stateless session bean handled exclusivley by the container taken from [56].	35
11	Life-cycle of a stateful session bean taken from [56].	36
12	Timeline of the development of the Service Component Architecture (SCA) standard taken from [60].	39
13	Basic component model of SCA taken from [60].	40
14	Basic composition model of SCA taken from [61].	41
15	Basic domain model of SCA taken from [62].	42
16	The java binding approach in SCA taken from [62].	43
17	The sca binding approach in SCA taken from [62].	44
18	Overall concept of the JBI architecture taken from [8].	49
19	Base concept of the JBI plug-in model taken from [8].	50
20	Component Distribution model of JBI taken from [78].	52
21	High-level messagee sequence chart of the normalized message exchange system of JBI taken from [8].	54
22	Routing of external messages taken from [8].	54
23	Routing of internal messages to external receivers taken from [8].	55
24	Figure describing the base architecture of the Swordfish framework, the base model of the Eclipse SOA platform. As seen it consits of OSGi, JBI and SCA building on each other taken from [105].	72

List of Tables

1	Differences between objects and components [1, 5].	7
---	--	---

References

- [1] Oscar Nierstrasz and Dennis Tsichritzis. *Object-Oriented Software Composition*. Prentice Hall International (UK) Ltd, Hertfordshire, Great Britain, 1995.
- [2] E. Bruneton Y, T. Coupaye Y, and J. B. Stefani Z. Recursive and dynamic software composition with sharing, 2002.
- [3] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and generative programming. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 2–19. Springer-Verlag, 1999.
- [4] Wolfgang Weck. Independently extensible component frameworks. In *Special Issues in Object-Oriented Programming*, pages 177–183. Dpunkt Verlag, 1997.
- [5] Luigia Petre. Components vs. objects. *TUCS Technical Reports*, (370), 2000.
- [6] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, 2000.
- [7] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, 1992.
- [8] Sun Microsystems. Java business integration (jbi) 1.0, 2005.
- [9] David Sprott and Lawrence Wilkes. Understanding service-oriented architecture. *TDBDI Forum*, 2004.
- [10] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.
- [11] Anurag Goel. Enterprise integration - eai vs. soa vs. esb.
- [12] Martin Fowler. Inversion of control containers and the dependency injection pattern, 2004. <http://martinfowler.com/articles/injection.html>.
- [13] Moisés Daniel Díaz Toledano. The architecture of enterprise information systems, 2002.
- [14] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.
- [15] Inc. Sun Microsystems. Jndi overview, 2008. <http://java.sun.com/products/jndi/overview.html>.
- [16] Richard S. Hall, 2007. OSGi R4 Service Platform: Java Modularity and Beyond.
- [17] Timo Honkanen. Osgi - open service gateway initiative.
- [18] OSGi Alliance, 2007. Leveraging OSGi Tecnology.
- [19] OSGi Alliance. About / homepage, 2008. <http://www.osgi.org/About/HomePage>.
- [20] The Eclipse Foundation. Eclipse.org home, 2008. <http://www.eclipse.org/>.

-
- [21] ObjectWeb. Jonas opensource java ee application server - main - webhome, 2008. <http://wiki.jonas.objectweb.org/xwiki/bin/view/Main/>.
- [22] SpringSource. Springsource application platform — springsource, 2008. <http://www.springsource.com/products/suite/applicationplatform>.
- [23] OSGi Alliance. About the osgi service platform - technical whitepaper, 2007.
- [24] Jan S. Rellermeyer and Gustavo Alonso. Concierge: A service platform for resource-constrained devices. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.
- [25] Jan S. Rellermeyer and Gustavo Alonso. Services everywhere: Osgi in distributed environments. *EclipseCon*, 2007.
- [26] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The eclipse 3.0 platform: Adopting osgi technology. *IBM Systems Journal*, 44(2), 2005.
- [27] Bryon Winkler. An implementation of an ultrasonic indoor tracking system supporting the osgi architecture of the icta lab, 2002.
- [28] OSGi Alliance. Osgi service platform - core specification, 2007.
- [29] Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison Wesley, 2003.
- [30] OSGi Alliance. Osgi service platform - service compendium, 2007.
- [31] OSGi Alliance. Listeners considered harmful: The.
- [32] JSR-232 Expert Group. Mobile operational management - for java 2 platform, micro edition - jsr-232, 2006.
- [33] JSR-211 Expert Group. Java 2 platform, micro edition - content handler api specification (chapi), 2005.
- [34] OSGi Alliance. Osgi service platform - mobile specification, 2006.
- [35] Ernest Hill. Overview of the spring framework.
- [36] Jan C. Jasik. Spring framework, 2008.
- [37] SpringSource. Company overview - springsource, 2008. <http://www.springsource.com/aboutus>.
- [38] SpringSource. Springframework.org, 2008. <http://www.springframework.org/about>.
- [39] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, and Ramnivas Laddad. Spring - java/j2ee application framework - reference documentation, 2008.
- [40] SpringSource. Spring phyton - spring phyton site, 2008. <https://springpython.webfactional.com/>.
- [41] Mark Pollack, Rick Evans, Aleksandar Seovic, Bruno Baia, Federico Spinazzi, Rob Harrop, Griffin Caprio, Ruben Bartelink, Choy Rim, and The Spring Java Team. Spring.net - reference documentation - version 1.2.0 m1, 2008.
-

- [42] SpringSource. Spring portfolio, 2008. <http://www.springsource.com/products/springportfolio>.
- [43] SpringSource. Spring security, 2008. <http://www.springsource.com/products/springsecurity>.
- [44] SpringSource. Springframework.org, 2008. <http://www.springframework.org/webflow>.
- [45] SpringSource. Spring web services, 2008. <http://www.springsource.com/products/springwebservices>.
- [46] SpringSource. Spring dynamic modules for osgi (tm) service platforms, 2008. <http://www.springsource.com/products/springdynamicmodules>.
- [47] SpringSource. Spring batch, 2008. <http://www.springsource.com/products/springbatch>.
- [48] SpringSource. Pitchfork, 2008. <http://www.springsource.com/pitchfork>.
- [49] SpringSource. Spring ide, 2008. <http://www.springsource.com/products/springide>.
- [50] SpringSource. Springframework.org - spring ldap, 2008. <http://www.springframework.org/ldap>.
- [51] SpringSource. Springframework.org - the spring rich client project, 2008. <http://www.springframework.org/spring-rcp>.
- [52] SpringSource. Spring integration, 2008. <http://www.springsource.com/products/springintegration>.
- [53] EJB 3.0 Expert Group. Jsr 220: Enterprise javabeans, verion 3.0 - ejb 3.0 simplified api, 2006.
- [54] Sun Microsystems. Java platform, enterprise edition (java ee) specification, v5, 2006.
- [55] Inc. Sun Microsystems. Enterprise javabeans technology, 2008. <http://java.sun.com/products/ejb/>.
- [56] EJB 3.0 Expert Group. Jsr 220: Enterprise javabeans, verion 3.0 - ejb core contracts and requirements, 2006.
- [57] EJB 3.0 Expert Group. Jsr 220: Enterprise javabeans, verion 3.0 - ejb persistence api, 2006.
- [58] TheServerSide. Enterprise java community: Application server matrix, 2007. <http://www.theserverside.com/tt/reviews/matrix.tss>.
- [59] Matthew Adams, Cezar Andrei, Ron Barack, Henning Blohm, Christophe Boutard, Stephen Brodsky, Frank Budinsky, Stefan Buennig, Michael Carey, Blaise Doughan, Kelvin Goodson, Andy Grove, Omar Halaseh, Larry Harris, Ulf von Mersewsky, Shawn Moe, Martin Nally, Radu Preotiuc-Pietro, Mike Rowley, Eric Samson, James Taylor, and Arnaud Thiefaine. Service data objects for java specification, 2008.
- [60] WebSphere User Group. Composing business solutions using sca, 2008.
- [61] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, Oisin Hurley, Sabin Ielceanu, Alex Miller, Anish Karmarkar, Ashok Malhotra, Jim Marino, Martin Nally, Eric Newcomer, Sanjay Patil, Greg Pavlik, Martin Raeppe, Michael Rowley, Ken Tam, Scott Vorthmann, Peter Walker, and Lance Waterman. Sca service component architecture - assembly model specification, 2007.
- [62] David Chappell. Introducing sca, 2007.

- [63] Michael Beisiegel, Dave Booz, Ching-Yun Chao, Mike Edwards, Sabin Ielceanu, Anish Karmarkar, Ashok Malhotra, Eric Newcomer, Sanjay Patil, Michael Rowley, Chris Sharp, and Uemit Yalcinalp. Sca policy framework, 2007.
- [64] Simon Holdsworth, Sabin Ielceanu, Anish Karmarkar, Mark Little, Sanjay Patil, and Michael Rowley. Sca service component architecture - web service binding specification, 2007.
- [65] Andrew Borley, David Haney, Oisin Hurley, Todd Little, Brian Lorenz, Conor Patten, Pete Robbins, and Colin Thorne. Sca service component architecture - client and implementation model specification for c++, 2007.
- [66] Martin Chapman, Sabin Ielceanu, Dieter Koenig, Michael Rowley, Ivana Trickovic, and Alex Yiu. Sca service component architecture - client and implementation model specification for ws-bpel, 2007.
- [67] Bryan Aupperle, Patrick Donnelly, and Terry Warren. Sca service component architecture - cobol client and implementation specification, 2007.
- [68] Bryan Aupperle, David Haney, Oisin Hurley, Todd Little, Conor Patten, Pete Robbins, and Terry Warren. Sca service component architecture - c client and implementation specification, 2007.
- [69] Ron Barack, Henning Blohm, Dave Booz, Rashmi Hunt, Michael Keith, and Michael Rowley. Sca service component architecture - ejb session bean binding, 2007.
- [70] Ron Barack, Michael Beisiegel, Henning Blohm, Dave Booz, Jeremy Boynes, Ching-Yun Chao, Adrian Colyer, Mike Edwards, Hal Hildebrand, Sabin Ielceanu, Anish Karmarkar, Daniel Kulp, Ashok Malhotra, Jim Marino, Michael Rowley, Ken Tam, Scott Vorthmann, and Lance Waterman. Sca service component architecture - java common annotations and apis, 2007.
- [71] Ron Barack, Michael Beisiegel, Henning Blohm, Dave Booz, Jeremy Boynes, Ching-Yun Chao, Adrian Colyer, Mike Edwards, Hal Hildebrand, Sabin Ielceanu, Anish Karmarkar, Daniel Kulp, Ashok Malhotra, Jim Marino, Michael Rowley, Ken Tam, Scott Vorthmann, and Lance Waterman. Sca service component architecture - java component implementation specification, 2007.
- [72] Ron Barack, Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, Anish Karmarkar, Michael Keith, Ashok Malhotra, Sanjay Patil, Prasad Peddada, Peter Peshev, Matthew Peters, and Michael Rowley. Sca service component architecture - java ee integration specification, 2007.
- [73] Michael Beisiegel, Dave Booz, Adrian Colyer, Hal Hildebrand, Jim Marino, and Ken Tam. Sca service component architecture - spring component implementation specification, 2007.
- [74] Simon Holdsworth, Sabin Ielceanu, Anish Karmarkar, Mark Little, Sanjay Patil, and Michael Rowley. Sca service component architecture - web service binding specification, 2007.
- [75] Bryan Aupperle, Martin Chapman, Codanda Chinnappa, Eric Johnson, Peter Peshev, Piotr Przybylski, and Michael Rowley. Sca service component architecture - jca binding specification, 2007.
- [76] Rajith Attapattu, Henning Blohm, Simon Holdsworth, Eric Johnson, Anish Karmarkar, and Michael Rowley. Sca service component architecture - jms binding specification, 2007.

- [77] Dave Booz, Mike Edwards, Mike Kanaley, Mark Little, Ashok Malhotra, Eric Newcomer, Sanjay Patil, Ian Robinson, and Michael Rowley. Sca service component architecture - acid transaction policy in sca, 2007.
- [78] Christoph Hartmann. Einfuehrung in java business integration. 2006.
- [79] Ingolf H. Krueger, Michael Meisinger, Massimiliano Menarini, and Stephen Pasco. Rapid systems of systems integration - combining an architecture-centric approach with enterprise service bus infrastructure. *IEEE Internation Conference on Information Reuse and Integration*, pages 51–56, 2006.
- [80] David A. Chappell. *Theory in Practice - Enterprise Service Bus*. O'Reilly, 2004.
- [81] Sun Microsystems. Company info, 2008. <http://www.sun.com/aboutsun/company/index.jsp>.
- [82] J. Steven Perry. *Java Management Extension*. O'Reilly, 2002.
- [83] Sun Microsystems. Java business integration vision - a technical white paper, 2004.
- [84] Ben Alex and Luke Taylor. Spring security - reference documentation 2.0.x, 2007.
- [85] Inc. Sun Microsystems. Common annotations for the java platform, 2006.
- [86] Sun Microsystems Inc. Jdk 6 security-related apis and developer guides – from sun microsystems, 2006. <http://java.sun.com/javase/6/docs/technotes/guides/security/>.
- [87] Torkel degaard. Coding instinct: loc container benchmark - unity, windsor, structuremap and spring.net, 2008. <http://www.codinginstinct.com/2008/04/ioc-container-benchmark-unity-windsor.html>.
- [88] JavaBeat. Life cycle management of a spring bean, 2008. <http://www.javabeat.net/articles/49-life-cycle-management-of-a-spring-bean-1.html>.
- [89] Apache Software Foundation. Easybeans - webhome, 2008. <http://commons.apache.org/logging/>.
- [90] William J. Brown, Raphael C. Malveau, and Thomas J. Mowbray. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, Inc., 1998.
- [91] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. *ACM SIGPLAN Notice*, 37(11):246–261, 2002.
- [92] OSGi Alliance. Osgi alliance blog: The osgi extender model, 2008. <http://www.osgi.org/blog/2007/02/osgi-extender-model.html>.
- [93] Peter Kriens. Extender pattern with automatic servlet registration, 2006. <http://www.aqute.biz/Snippets/Extender>.
- [94] SYS-COM Publications. Spring and ejb 3.0 in harmony, 2008. <http://java.sys-con.com/node/180386?page=1>.
- [95] Ramanujam A. Rao. The java ee application as an ejb/spring/hibernate hybrid, 2007. <http://www.devx.com/Java/Article/35722>.

-
- [96] Adrian M. Colyer, Hal Hildebrand, Costin Leau, and Andy Piper. Spring dynamic modules reference guide - 1.1.0, 2008.
- [97] SpringSource. Springsource dm server, 2008. <http://www.springsource.com/products/suite/dmserver>.
- [98] Apache Software Foundation. Openejb home, 2008. <http://openejb.apache.org/>.
- [99] EasyBeans and ObjectWeb consortium. Easybeans - webhome, 2006. <http://wiki.easybeans.org/xwiki/bin/view/Main/WebHome>.
- [100] Paremmus Limited. Newton 22: Developer guide, 2008.
- [101] Mike Edwards. Relationship of sca and jbi - open soa collaboration, 2007. <http://www.osoa.org/display/Main/Relationship+of+SCA+and+JBI>.
- [102] Brian O'Neill. Sca and jbi: Made for each other. (soa gone wild), 2007. http://weblogs.java.net/blog/boneill42/archive/2007/06/sca_jbi_made_fo.html.
- [103] Dmitri Ilkaev. Review of sca and jbi in soa world, 2007.
- [104] Paul Krill. Eclipse prepares open source soa framework, 2008. <http://www.cio.co.uk/technology/soa/news/index.cfm?articleid=2465>.
- [105] Oliver Wolf. Eclipse summit on runtime technologies and platforms - swordfish project introduction, 2007.