University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master Thesis

Determining and publishing component compatibility information

Pilsen 2014 Jakub Daněk

Statement

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Pilsen, 05. 05. 2014

Jakub Danek

Abstract

Component substitutability verification can be rather resource-demanding for small or mobile devices. Component Repository supporting Compatibility Evaluation (CRCE) developed at the Department of Computer Science and Engineering at the University of West Bohemia aims to move the burden of compatibility-related meta-data computation from its clients. The goal of this master thesis was to design and implement a suitable storage for the compatibility-related meta-data. The storage allows the repository to compute the meta-data in advance, at the time of component upload, and therefore significantly shortens the response times needed to provide component compatibility information. The meta-data are acquired using existing tools developed at the Department. Access to the meta-data is realized via RESTful web-service API using XML data format. As such, the information is processable by both client applications and their maintainers.

Abstrakt

Proces ověření kompatibility komponent je náročný na zdroje, což je problém především u malých či mobilních zařízení. Component Repository supporting Compatibility Evaluation (CRCE) vyvíjené na Katedře informatiky a výpočetní techniky na Západočeské univerzitě v Plzni má za úkol provést výpočet potřebných metadat za své klienty. Cílem této diplomové práce bylo navrhnout a implementovat vhodné úložiště metadat vztahujících se ke kompatibilitě komponent. Úložiště umožňuje vytvoření metadat již při ukládání komponent. Důsledkem je výrazné zkrácení doby potřebné pro rozhodnutí o kompatibilitě komponent. Metadata jsou získávána pomocí nástrojů již dříve vyvinutých na Katedře. Úložiště poskytuje veřejné webové služby založené na principech REST. Výstup je díky využití technologie XML čitelný pro klientské aplikace i lidi.

Contents

1 Introduction **1**

2 Component-based Software Engineering **3**

2.1 Overview 3

2.2 OSGi Component Model 5

2.2.1 OSGi Implementations 6

2.2.2 OSGi Component 6

2.2.3 OSGi Service Layer 8

3 Component Substitutability **10**

3.1 Verification of Component Substitutability 11

3.1.1 Semantic Versioning 11

3.1.2 Substitution Checks for Typed Specifications 12

3.1.3 Component Subtyping 13

3.1.4 Subtyping as Reliable Compatibility Check 14

3.2 OSGi Bundle Compatibility Checker 15

3.2.1 OBCC Workflow 15

3.2.2 OBCC Architecture 16

3.2.3 OBCC-based tools 16

3.2.4 Java Class Comparator 17

3.2.5 Comparator Results 17

3.2.6 Project Status 19

3.2.7 Related Work 19

4 Component Repository supporting Compatibility Evaluation **21**

4.1 CRCE Architecture 21

4.2 Component's Lifecycle 22

4.3 Data Store 23

4.3.1 Store Types 23

4.3.2 Meta-Data Structure 24

4.4 Client Access 26

4.5 Modules 27

4.6 Project Status 28

5 Difference Meta-Data Repository **29**
5.1 Analysis 29
5.2 Use Case Summary 30
5.3 Data Model 31
5.4 Persistence Layer 33
5.4.1 Analysis 33
5.4.2 Analysis Conclusion 36
5.4.3 MongoDB 37
5.4.4 Implementation 38
5.5 Service Layer 40
5.5.1 Implementation 40
5.6 Summary 42

6 CRCE Extensions **43**
6.1 CRCE Concurrency Plugin 43
6.2 CRCE Compatibility Plugins 44
6.2.1 Automated Versioning Plugin 44
6.2.2 Compatibility Plugin 45
6.3 Web User Interface 46

7 Web Service Module **47**
7.1 RESTful Web Services 47
7.2 CRCE Web Service API 49
7.2.1 Original API 49
7.2.2 API Extensions 51
7.3 Meta-Data XML Representation 53
7.3.1 Core Meta-Data 53
7.3.2 Compatibility Meta-Data 55
7.4 Implementation 56

8 Testing **57**
8.1 Correctness 57
8.2 Performance 58

9 Conclusion **63**

Bibliography **65**

List of Abbreviations **68**

List of Examples 68

List of Figures 69

List of Tables 70

A Meta-Data XML Representation 73

B Compatibility Meta-Data JSON Representation 75

C Performance Test Results 78

1 Introduction

Component-based development promotes the idea of building modular software from reusable components. Each component is an independent unit existing on its own and capable of communication with others via strictly defined contract (i.e. interface). Such application design enables separation of concerns in the matter of functionality into small building blocks and increases re-usability of code. A component, once written, can be integrated into any future project.

Modularity of applications also decreases maintenance costs. Well-structured code is easier to read and understand. Assuming the communication contract remains the same, the encapsulation of functionality within component eliminates the need to test the whole system when a single component changes.

However, due to possible interface modifications it is necessary to test compatibility of new components with the rest of the application. Therefore it is considered generally unsafe to replace components at runtime as potential incompatibility might lead to breakdown of the whole system. Difficulty of the task grows with the application size and if done manually, it is rather error-prone.

Researchers at the Department of Computer Science at the University of West Bohemia in Pilsen, Czech Republic, are working on tools for computational verification of component compatibility. The goal is to provide more reliable way of component compatibility testing.

The method is based on comparison of public interface changes using component's binaries. Target platform of the research is Open Services Gateway initiative (OSGi) - component framework for Java applications. The method itself is, however, suitable for any other component framework.

The research group presented Component Repository supporting Compatibility Evaluation (CRCE) which will provide the compatibility information for stored components on its service layer, thus taking the burden of computation from its clients. The process of extracting required information from component's binaries and the following comparison requires rather high amount of computational resources. Therefore it is unsuitable for e.g. mobile devices which usually aren't equipped with a lot of memory and powerful computing units, and which also suffer from limited power supply. The aim of this paper is to deliver a CRCE module for working with compatibility information.

This task is divided into two main parts. The first focuses on acquiring the data using tools developed by the research group and storing the results inside CRCE. Proper storage format must be designed and a suitable persistence technology selected. Design of the module must take into consideration time requirements of the component comparison. It is required that the module has minimal impact on responsiveness of the application. Realization of the task is described in chapters 5 and 6 and chapter 8 contains overview of test results of the implemented module.

The other goal of this this work is to expose the compatibility data to clients via both web services and user interface. While the compatibility information returned via web services need to be processable by machines, it is also important to maintain some level of human readability. That should allow clients to make sound decisions about component replacements. Web Service Module and its extension supporting compatibility meta-data are described in chapter 7.

The initial part of this thesis introduces principles of component-based software engineering (chapter 2) and explains issues related to component substitutability (chapter 3). Chapter 4 provides detailed description of CRCE and its status at the beginning of the project.

2 Component-based Software Engineering

2.1 Overview

The concept of component-based software engineering (CBSE) promotes the idea that applications should be created as an assembly of independent components. The design pattern of component-based systems is defined by the types of components in the system and their interaction rules [2]. Graphical representation of the pattern is in Figure 2.1.

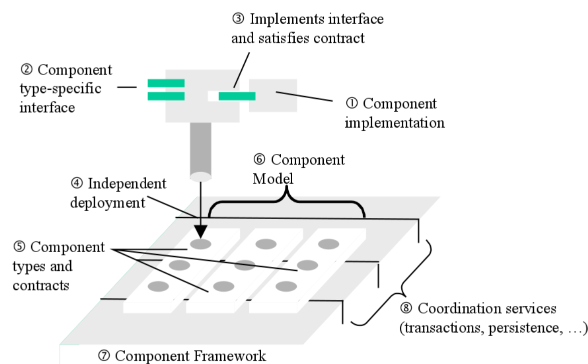

Figure 2.1: The component-based design pattern [2].

A component is an encapsulated implementation of related functionality. It usually implements one or more public interfaces and to function properly, it requires one or more services provided by other components in the system.

For its surroundings, a component is defined by its public interface. Therefore any component can be easily tested on its own by simply providing own test implementations of required services. Any component can be replaced by a different one, under the condition the replacement implements the very same interface. Therefore it is important to keep in mind that no component should rely on a particular service implementation.

Each component also needs to fulfil specific requirements given by chosen component model. That allows the component framework to properly manage the system, all its components and their communication.

Component Model

A component model is set of rules and assumptions components must fulfil in order to be able to co-operate. There is no definite answer to what should or must be defined in a component model as discussed in [2]. The paper further reveals the reasons of imposing standards and requirements on components' developers:

Two components can interact with each other if they have correct assumptions about what each of them provides and requires of the other. While some of the assumptions will always be unique to the particular component, many of them are common and hence possible to standardize. These standards might specify the means of components' location, communication protocols, control flow synchronisation etc.

Except for standardized approach to building components it is important to provide stable runtime environment and simple deployment options. Due to the composite nature of the applications and independence of components, the infrastructure for deployment must be standardized enough to allow smooth transition from development environment to testing or customer environment. This has been the main motivation for component frameworks.

Based on these requirements, a component model is most likely to impose the following standards[2]:

- Component types - the type may be defined by the interfaces a component implements. If a component implements more than one interface, it can fulfil the role of any of the interfaces - types (hence it is polymorphic with respect to these types). A component model requires components to implement one or more interfaces and in therefore it defines one or more component types.

- Interaction schemes - component model will specify two classes of interaction schemes. The first one covers component lifecycle management. The second one focuses on quality aspects of communication - security, transactions, number of parallel communications, etc.

- Resource binding - A resource is either a service provided by the component framework or by one of the other components. Component model describes which services are available to which components and the means of their binding.

Component Framework

> A good way to think of a component framework is as a mini-operating system. In this analogy, components are to frameworks what processes are to operating systems[2].

Component framework is a runtime environment for components. It is implementation of a component model. The framework manages shared resources, components' lifecycle, handles inter-component communication, etc. All components in the system must fulfil specification of the component model, otherwise the framework would be unable to work with them.

2.2 OSGi Component Model

The Open Services Gateway initiative (OSGi) technology is a set of specifications that define a dynamic component model for Java [1]. It is created and maintained by OSGi Alliance. The purpose of the framework is to allow developers to assemble applications from reusable and independent components. Creators of these components shouldn't be required to have thorough knowledge of the implementation of remaining components.

Figure 2.2 depicts model of the framework, which consists of the following layers [1]:

- Bundles - Bundles are the OSGi components made by the developers.
- Services - The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
- Life-Cycle - The API to install, start, stop, update, and uninstall bundles.
- Modules - The layer that defines how a bundle can import and export code.
- Security - The layer that handles the security aspects.
- Execution Environment - Defines what methods and classes are available in a specific platform.

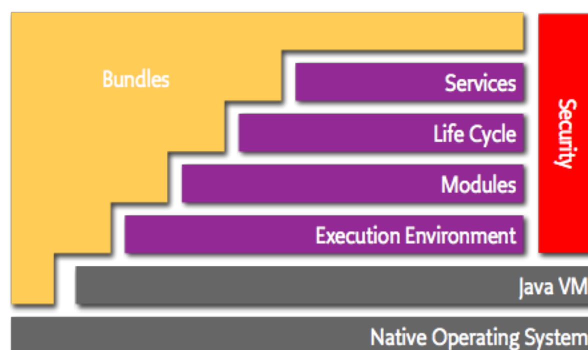

Figure 2.2: OSGi Model [1].

2.2.1 OSGi Implementations

There are several implementations of the OSGi component model. They differ in amount of specification implemented, user interface and support tools. Here is the list of some of the most commonly used OSGi implementations:

Eclipse Equinox[5] Open-source OSGi implementation developed as part of the Eclipse project.

Apache Felix[6] Open-source, community-driven implementation of OSGi component model.

Knoplerfish[7] Open-source implementation of OSGi component model strongly supported by Swedish company Makewave[1].

2.2.2 OSGi Component

In terminology of OSGi is component a bundle. Bundle is a regular java archive (JAR) file with a manifest file `META-INF/MANIFEST.MF`. The file contains specification of the component represented as simple key-value pairs. The following list contains some of the most common keys:

Bundle-Name Optional key. Human-readable name of the bundle.

Bundle-SymbolicName A required key. Its value is the unique name of the bundle within system. It is quite common for its value to follow the same naming conventions as fully qualified Java package names.

Bundle-Version Optional key. Contains version of the particular bundle in the format `major.minor.micro.qualifier`, conform with OSGi semantic versioning (as described in section 3.1.1).

Bundle-Activator Optional key. Fully qualified name of a class implementing the `BundleActivator` interface. The interface allows developers of the bundle to provide own actions within particular events during the component's lifecycle (e.g. when starting or shutting down the component). Methods of the interface are automatically called by the framework.

Export-Package Comma-separated list of packages the bundle exposes. After starting a bundle, classes within these packages are available to the other components within the framework. It is possible to specify version of the exposed package.

[1]Makewave - http://www.makewave.com/site.en/about/

Replace-Bundle Comma-separated list of bundle which must be found in the system in order to start the component. As a consequence the component is dependent on these particular bundles. Therefore the use of this header is not recommended in favour of the `Import-Package` header.

Import-Package Comma-separated list of packages the bundle needs to be present within the bundle context in order to function properly. It is possible to specify particular version (or version range) of a package to import. If the framework cannot find imported packages within the system, it refuses to start the component. Main advantage of this header over the `Require-Bundle` is the fact it doesn't impose dependency on any particular bundle.

The Example 2.1 shows a manifest file:

```
Bundle-Name: Example bundle
Bundle-SymbolicName: cz.zcu.kiv.obcc.exampleBundle
Bundle-Version: 1.0.0-R123
Bundle-Activator: cz.zcu.kiv.obcc.exampleBundle.Activator
Export-Package: cz.zcu.kiv.obcc.exampleBundle
Import-Package: cz.zcu.kiv.obcc, org.slf4j.impl;version=”[1.3.1, 1.4.0)”
```

Example 2.1: Sample of a manifest file. [3]

The example represents a bundle `Example bundle`, which is identified as `cz.zcu.kiv.obcc.exampleBundle` within the system. Its version is `1.0.0-R123`, has a `BundleActivator` implementation and exports `cz.zcu.kiv.obcc.exampleBundle`. The bundle requires any version of `cz.zcu.kiv.obcc` package and `org.slf4j.impl` within version range `<1.3.1, 1.4.0`).

OSGi Component Lifecycle

As one can see in Figure 2.3, a bundle can be in one of the following states [4]:

INSTALLED The bundle has been successfully stored in the persistent storage of the framework.

RESOLVED All required classes are present in the bundle context. Therefore the bundle is ready to be started.

STARTING The bundle is being started and its `BundleActivator.start` method is called. This state indicates that the method has not returned yet.

ACTIVE The bundle is running.

STOPPING The bundle is being stopped and its `BundleActivator.stop` method is called. This state indicates that the method has not returned yet.

UNINSTALLED The bundle has been removed from the framework and can no longer change its state.

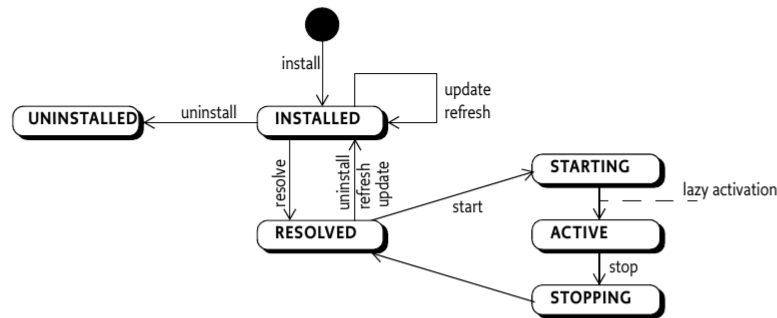

Figure 2.3: OSGi Bundle Lifecycle [4].

OSGi Resolving

During the resolving phase the framework checks that all required resources are present in the system and therefore the installed bundle can be started. In the context of the OSGi it means that all the packages stated in the `Import-Package` and all bundles specified by the `Require-Bundle` headers of the component's manifest file must be found in the application.

Resolving is a complicated task which must deal with several complex situations. It is possible that there are more bundles satisfying an `Import-Package` requirement. Another case is when a package implementation is spread across several components.

Due to the dynamic nature of the environment it is possible for multiple versions of the same bundle co-existing in the framework at the same time. This situation occurs when a new bundle version is loaded into the system while there are still components with connections to the old version. In such case the runtime is not updated instantly and may cause compatibility issues.

2.2.3 OSGi Service Layer

Bundles have two main ways of cooperation. The first is direct reference of classes exported by another bundle. The other way is to use services.

An OSGi service is a Java object registered under one or more interfaces. The interface should reveal none or minimum amount of information about its implementation.

> The service object is owned by, and runs within, a bundle. This bundle must register the service object with the Framework service registry so that

the service's functionality is available to other bundles under control of the Framework [4].

When a bundle registers a service with the framework's service registry, any other component can request the service under its interface name. The framework contains query mechanism which provides the bundle with means to search and ask for services it requires. Due to the framework's event system it is possible for the bundles to receive messages when new services are registered or status of an existing service has changed [4].

Clients of an OSGi service must, however, accept the fact that a service may or may not be available when needed. This is caused by the dynamic nature of OSGi environment, due to which it is impossible to ensure that a service won't be unregistered just moments before it is called.

3 Component Substitutability

While it is easier to find and fix bugs and provide upgrades with the component-based development, it also brings higher integration risks. Components must be replaced very carefully due to possibility of introducing an incompatible component into the system (which may cause unexpected behaviour or crash). Therefore there have been desire to provide reliable methods of component compatibility verifications.

These methods can be classified by the level of interface contract on which they operate and by nature of the means (either static or dynamic) used to determine the compatibility[28]:

- **Syntactic compatibility** is based on component interface specification and its type representation (see section 3.1.3 further). Compatibility is resolved through subtype relation checks.
- **Semantic and interaction compatibility** works with behaviour models on both sides of inter-component bindings. These models can be either specified in advance, or retrieved by reflection or run-time observation from component's implementation. The models are either compared by static model-checking to determine compatibility or can be used as basis for a compatibility verification test suite.
- **Extra-functional properties (EFP) compatibility** has been getting more attention because of significant role of performance or security in the architecture and implementation of current systems. However, the means of specification, comparison and verification of the EFP models are in the area of active research and do not belong to the current state-of-art.

In the general case of component substitutability problem there are presumably no relations between the current and replacement component sets. Under such conditions substitutability checks must be done within the particular context [28].

In the specific case of the replacement component being a downstream revision of the current one, the situation is a little easier. In this scenario, the component substitutability means backward compatibility, which can be verified in advance and the information can be stored in compatibility meta-data. Particular means of backward compatibility verification are discussed further in this chapter.

3.1 Verification of Component Substitutability

The need of extensive (and expensive) testing is somewhat reduced by effort of component framework's to lower the risk of loading an incompatible component. One of the common practices is semantic versioning.

3.1.1 Semantic Versioning

In OSGi, the manifest file of each bundle may contain version designation for each imported/exported package. The framework can therefore ensure that compatible versions of components are used. Version compatibility however differs depending on the purpose for which a bundle imports the package.

> Bundles that consume an API package have different backward compatibility rules than a provider of that API. Any semantic change in the API package must be handled by a provider to honour the change in the API contract while many of those changes are backward compatible for consumers [8].

Versions use the format `major.minor.micro.qualifier` with the following semantics [8]:

1. major — Packages with versions that have different major parts are not compatible for both consumers and providers. For example, 1.2.0 and 2.3.0 are completely incompatible. This is usually caused by changing or removal of a method which is part of the bundle's public API.

2. minor — API consumers are compatible with exporters that have the same major number and an equal or higher minor version. For example, 1.2 is backward compatible with 1.1 for consumers. This doesn't apply to providers who are incompatible with exporters with different minor version. Common reason for minor version incrementation is extension of the bundle's API with new methods.

3. micro — A difference in the micro part does not signal any backward compatibility issues. The micro number is used to fix bugs that do not affect consumers of the API.

4. qualifier — The qualifier is usually used to indicate a build identity, for example a time stamp. Different qualifiers do not signal any backward compatibility issues.

The semantic versioning provides the framework with means to decide bundle compatibility. However it puts high responsibility on bundle developers to provide exact and correct information about version of the bundle and the packages they export. Potentially invalid version information renders the automated compatibility checking rather unreliable and as-so doesn't suppress the need of extensive manual compatibility testing.

3.1.2 Substitution Checks for Typed Specifications

The concept of type-based substitutability and its use for component compatibility checking has been described in [9].

Type systems and the subtype relation are used to ensure safe substitutability in object oriented programming languages. Same principle can be used for components: component `A'` can replace the component `A` if $A' <: A$[1].

> To determine whether a structured data type `B` is a subtype of `A`, the subtyping rules for the contained elements (parts) are used recursively until primitive types are reached, for which the rules are defined by enumeration (e.g. *short* <: *long*) [9].

In summary, the substitute `B` must provide at least the same functionality as the original `A`, and in the same manner `B` may require only as much as `A` did.

One exception to the theoretical expectations of subtyping rules represent statically typed languages. Example 3.1 below represents a simple interface type.

```
interface Logging { //ver.1
     void write(String msg);
     long countLogItems();
}
interface Logging { //ver.2
     void write(String msg);
     void writeMany(String msgs[]);
     short countLogItems();
}
```

Example 3.1: Statically typed languages problem [9].

Version 2 of the interface is a subtype of version 1 and therefore should be a suitable replacement. However, invocation of the method `countLogItems()` by a Java client without recompilation would result in `NoSuchMethodError` exception. Reason for this behaviour is static typing of Java and therefore inability of JVM[2] to cast the method signature from the one with return type `short` to the one returning `long`.

[1] <: is operator "subtype of".
[2] Java Virtual Machine

Classes of Type Differences

When evaluating subtype relation between two types, the result can be described by the character of their difference. The results of function $diff(a,b) : Type \times Type \rightarrow Difference$ fit into the the following set of classes [9]:

none if a = b;

insertion (ins) if a is not defined, but b is;

specialization (spec) if b <: a;

deletion (del) if a is defined but b isn't;

generalization (gen) if a <: b;

mutation (mut) if b contains both ins/spec and del/gen differences

unknown if b cannot be compared to a (e.g. due to recursive cycles)

For a structured type, the difference values are computed for each of its parts. Combination of the results then provides the difference value for the whole type. The combination process is recursive where parent element holds combined value of its children. Table 3.1 describes the key for combining the difference values. For algorithm details see chapter 3.1.1 in [9].

	ins	del	spec	gen
ins	ins	mut	spec	mut
del		del	mut	gen
spec			spec	mut
gen				gen

Table 3.1: Combination of difference values [9].

3.1.3 Component Subtyping

Component Representation as a Type

The component type is formed by the component interface. The type consists of individual public interface and attribute types. The elements can be divided into two sets - the provided and the required items.

Specification of the component's interface is spread among multiple places depending on the framework. Parts of it can be found in particular deployment descriptors, XML schema descriptors and via lists of exported/imported packages. Some pieces of information can be found only during source code analysis. Either way it is always possible to reconstruct the whole view of component's interface.

Component Subtyping based on Difference Classification

Component `A` can be replaced by component `A'`, if the component `A'` is a subtype of the component `A`.

As mentioned earlier in section 3.1.2, component `A'` is a subtype of component `A` if it provides as much functionality (fulfils same or stronger non-functional properties etc.) and has at most the same requirements as the component `A`. That is, the provided part $A^{'}_{prov}$ is a subtype of A_{prov} ($A^{'}_{prov} <: A_{prov}$) and the required part $A^{'}_{req}$ is a supertype of $A_{req}(A_{req} <: A^{'}_{req})$.

Described via difference classes, component `A'` is a substitute for the component `A` if:

- $diff(A_{prov}, A^{'}_{prov}) \in \{none, insertion, specialization\}$
- $diff(A_{req}, A^{'}_{req}) \in \{none, deletion, generalization\}$

Because of reasons described in section 3.1.2, OSGi bundles (or any other components written in language with static type binding) the component `A'` is a substitute for the component `A` if:

- $diff(A_{prov}, A^{'}_{prov}) \in \{none, insertion\}$
- $diff(A_{req}, A^{'}_{req}) \in \{none, deletion\}$

3.1.4 Subtyping as Reliable Compatibility Check

The described method can provide means for safer and easier updates of component-based applications. Relationship between difference value and compatibility of compared components is described in table 3.2.

diff(A, B)	None	Specialization	Generalization	Mutation
A is compatible with B	Yes	No	Yes	No
B is compatible with A	Yes	Yes	No	No

Table 3.2: Mapping of difference values to compatibility [10].

This section describes several practical use-cases of the method [10].

Extended Meta-data

Detailed information about differences between two versions of the component (or two components) provide decision maker with enough data to evaluate the possibility of the component's replacement. Such decision can be required in case of skipping several revisions of the component. While version identificator is always result of chain of changes, difference meta-data can be computed pairwise for every previous version of the component.

Automated Updates

The method can also be applied at the deployment time of components to ensure compatibility. In such case, version numbers are not taken into account and the new component is compared to the old one. Such approach eliminates the risk of introducing formally incompatible component into the system.

It is possible to compare new component to the context of the old one within the system. By context we mean subset of component's features that are actually used in the application. The context is retrieved from the application by reflection. Such representation of the old component is most likely going to be incomplete. From compatibility point of view only the parts of the component actually used by the system (and equally only the objects used by the component on the requirements side) are important. Therefore the method can determine component's substitutability in the specific context.

Reliable Versioning

When used to compare two subsequent versions of a component, the described method can be used to reliably determine the new version identifier from consumer's point of view. Versioning rules are described by table 3.3 [9].

diff(A, A')	maj_new	min_new	mic_new
none	maj_old	min_old	mic_old + 1
specialization, insertion	maj_old	min_old + 1	0
deletion, generalization, mutation	maj_old + 1	0	0
unknown	unknown		

Table 3.3: Derivation of new version identifier.

3.2 OSGi Bundle Compatibility Checker

OSGi Bundle Compatibility Checker (OBCC) is a project developed at the Department of Computer Science the University of West Bohemia. The OBCC is a Java implementation of the subtype relation compatibility checking for OSGi bundles. As such it can be used as base for tools focused on automated versioning or safe component updates.

3.2.1 OBCC Workflow

OSGi Bundle Compatibility Checker is a tool with all capabilities required for loading OSGi bundle representations and subsequent component comparison.

Figure 3.1 describes OBCC workflow and connection of individual modules to particular activities. Replacement bundle representation is loaded from its binary archive, whereas original bundle representation can be either acquired from its binary or its footprint within

the current application context. The first approach is likely to be used for e.g. versioning purposes of bundle revisions. The latter is more suitable e.g. for compatibility checks while updating components at runtime.

Bundle comparison is divided into three separate sets - exported packages, exported services and imported packages. Altogether the comparison results form a full view of changes in component's both required and provided interface.

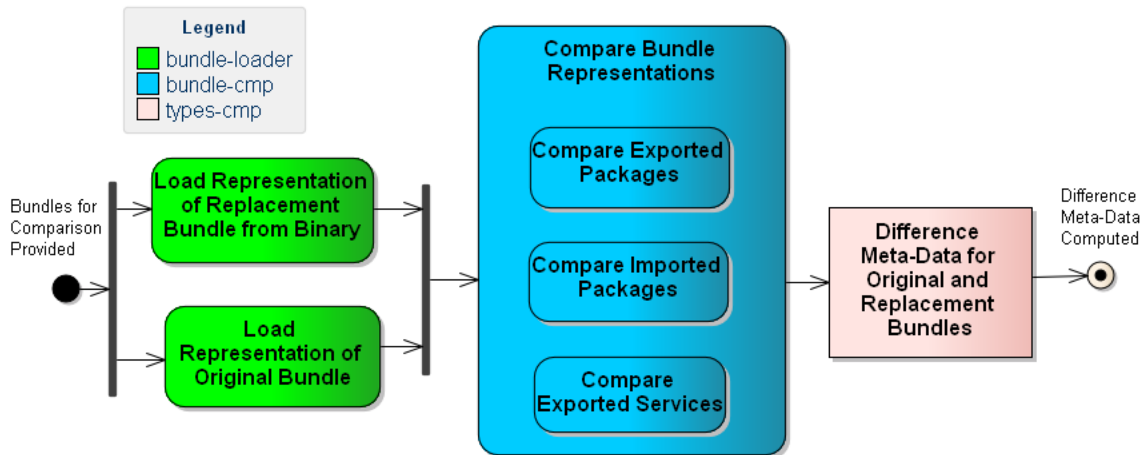

Figure 3.1: OBCC Workflow.

3.2.2 OBCC Architecture

OBCC consists of the following modules:

Bundle Comparator Contains implementation of OSGi bundle comparator.

Bundle Context Loader This module is capable of extracting OSGi bundle representation from its context within a running application.

Bundle Loader Provides functionality for extracting OSGi bundle representation from jar files.

Bundle Types Contains domain classes for representation of OSGi bundles and their elements.

3.2.3 OBCC-based tools

OSGi Bundle Version Generator

OSGi Bundle Version Generator[12] is a tool based on OBCC which can compute component's version based on its differences against the previous revision.

3.2.4 Java Class Comparator

Java Class Comparator is a Java library capable of extracting Java type information representation and which is able to perform type comparison on the extracted data. The type representation is obtained by Java bytecode inspection. The process of reconstruction of component's whole public API from its binary form is fully described in [10].

JaCC Workflow

JaCC proceeds recursively from top level (package) to the lowest (attribute types, method arguments, etc.) for each element (package, class, method, ...). At the lowest levels the resulting difference class[3] of each element is result of direct comparison. On higher levels it is an aggregation[4] of difference values of the element's children.

Implementation

The library presents own Java type system representation, which is very similar to the classes provided by `java.lang.reflect` package. Own representation has been introduced due to particular limitations of the Reflection API. For purposes of compatibility checks, it must be possible to acquire the representation by other means than classloader (e.g. bytecode inspection). Custom implementation is also comparable by subtyping rules.

The library consists of the following modules:

Java Types Contains domain classes for Java type system representation.

Java Types Comparator Contains implementation of the Java type comparator.

Java Types Loader Provides functionality for extracting Java type representation from bytecode.

Comparator Types Provides basic classes and interfaces for comparators.

3.2.5 Comparator Results

Results of the comparison process are stored in a tree structure which by its shape copies the recursive walkthrough of the comparison process through representations of the components.

[3]Representation of type difference classes as described in section 3.1.2.
[4]For aggregation rules see table 3.1.

Comparison result of the two interfaces from the example 3.1 is shown in figure 3.2.

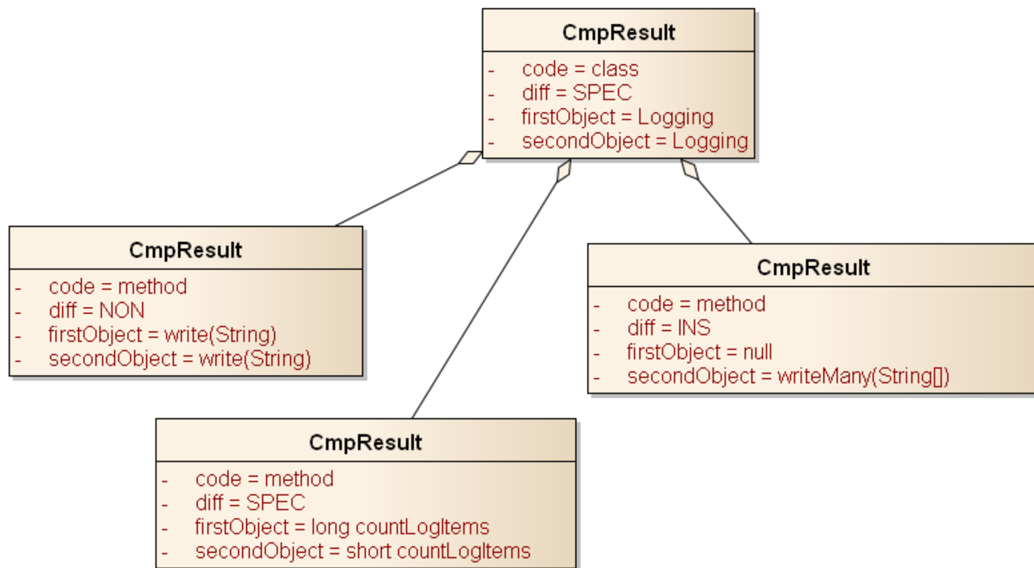

Figure 3.2: Example Comparison Result.

Data Model

The tree structure is built from objects implementing `CmpResult<T>` interface from the `types-cmp` module of JaCC. Each object describes differences between pair of compared items. The following methods are part of the interface:

- `Difference getDiff()` - Returned value is the difference class for the two compared objects. The value is either result of direct comparison (in case of basic elements) or result of aggregation of differences of child elements (in case of complex elements, e.g. classes)

- `T getFirstObject()`, `T getSecondObject()` - Returns reference to the first (second) of the two compared objects. In case an element has been added or removed to the component's interface, one of the objects is null. In the rest of the cases both the objects share properties (e.g. name) as they are representation of the same (possibly modified in one or the other case) items.

- `String getCode()` - Nature of the interface is generic, which makes it difficult to parse without any additional information about node's content. Value of the code identifies particular element type which the result is related to (package, class, method, ...).

- `List<CmpResultInfo> getChildren()` - The method retrieves all child results.

3.2.6 Project Status

Both JaCC and OBCC projects are available for download from their Assembla project pages under open-source Apache License v2.0.

JaCC URL: `https://www.assembla.com/spaces/jacc`

OBCC URL: `https://www.assembla.com/spaces/obcc`

Assembla project spaces contain project documentation in form of wiki pages, issue trackers and source code repository links.

Current release version of JaCC is `1.0.1` and for OBCC it is `1.0.2`.

Known Issues

Duplicate Methods Comparators have trouble with recognition of method implementations. If we had class C implementing interface I, and we added a method to the interface (and implemented it in C), the comparison result of the new version of class C against the original would contain two added methods instead of just one.

Moving Class through Hierarchy This issue has probably same base with the previous one. If we move a method from a class into its parent, the comparator recognizes it as deletion and claims the class (component) is not backwards-compatible.

However, moving methods upwards through class hierarchy doesn't have any impact on binary compatibility[13] and therefore the comparator's result is a false negative.

3.2.7 Related Work

OBCC is not the only project trying to provide developers with tools which would lower the risks and effort bound to component substitutability, versioning and API maintenance.

Eclipse Plugin-Development Environment API Tools [31] is set of utilities integrated into Eclipse SDK to assist developers with API maintenance during Eclipse plug-in development. Main functions provided by the tools are:

- Binary incompatibility between two versions of a component.
- Automated version numbers updates based on Eclipse versioning scheme.
- Identify usage of non-API code in other plug-ins.
- Identify usage of non-API types in API.

bnd [32] is a tool for OSGi that aims on lifting the burden of by-hand maintenance of OSGi manifest from developers. It uses information gained by class file analysis together with instructions provided by developers to create the manifest file automatically. It is commonly used nowadays as an Ant task, an Eclipse plug-in or maven plug-in[33].

Clirr [29, 30] is a tool for compatibility checking of Java libraries with their older releases. Given two sets of java libraries, Clirr returns list of public API changes.

Results are returned as set of log messages with particular code specifying type of change and additional description. In the example 3.2, the new revision introduced a new class.

```
INFO: 8000: demo.parking.statsbase.CountingStatisticsAbstractBaseImpl:
 Class demo.parking.statsbase.CountingStatisticsAbstractBaseImpl added
```

Example 3.2: Clirr result reporting an added class.

The tool can be used either via command-line interface, via Ant or as a Maven plugin. The latter approach provides automated compatibility verification of each build to previous revisions of the project.

Clirr is used e.g. by Apache Commons CLI[5] or Apache Commons Net[6] projects to display comparison of latest releases. However, it seems that the tool development has stopped, since the last commit in its GitHub repository is from 18. 06. 2013 (valid on 04. 05. 2014).

[5] Apache Commons CLI: http://commons.apache.org/proper/commons-cli/clirr-report.html
[6] Apache Commons Net: http://commons.apache.org/proper/commons-net/clirr-report.html

4 Component Repository supporting Compatibility Evaluation

Current small devices (GPS, mobile phones, tablets) provide sufficient performance for running component frameworks [14]. However, resources of such devices are still limited and therefore it is important to make upgrades as efficient as possible. Component Repository supporting Compatibility Evaluation (CRCE) is a a component repository which allows to move demanding computation from the devices. Processed component data are stored inside the repository and accessible via both web user interface and REST API. Client devices can access the component meta-data in form of XML document and use them for component compatibility evaluation.

The required compatibility meta-data will be acquired by comparison of type representations of the particular components using OBCC tool described in the previous chapter. To allow clients to quickly find safe and suitable replacement, components stored in CRCE will be automatically marked with semantic version identifier (see section 3.1.1). Its value will be derived from comparison results according to rules described in 3.1.4.

Concept of CRCE, original design and implementation have been described in [14, 34] and Zuzana Buresova dealt with performance optimizations in her work [3].

This chapter covers the state of CRCE at the beginning of this master thesis project.

4.1 CRCE Architecture

CRCE is a highly modular component storage based on OSGi Bundle Repository (OBR). Modularity is achieved using OSGi framework and well-defined Plugin API. The repository is developed and tested using OSGi bundles. However, it is designed with emphasis on genericity so that it can potentially be used with other component models.

Architecture of the repository is pictured in Figure 4.1.

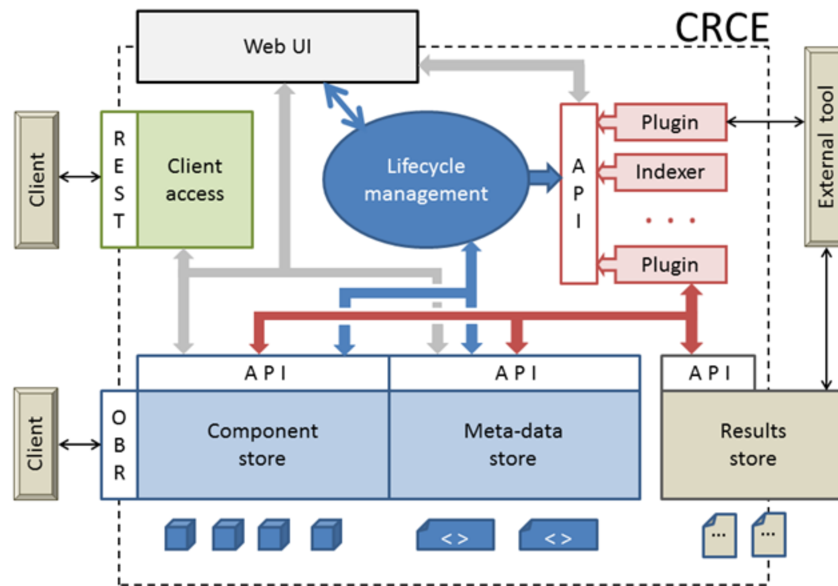

Figure 4.1: CRCE Architecture [16].

CRCE stores three main types of information - the component binaries (Component store), component descriptors (Meta-data store) and other data related to particular components (Results store). All the data are computed by various plugins and indexers during component's lifecycle within the repository. Additional plugins can be provided to extend CRCE's functionality.

Component's data can be accessed via web UI and REST API.

4.2 Component's Lifecycle

During its existence in the system, a component can find itself in one of three states. The lifecycle begins with uploading the component via web user interface or REST API. The component can be in one of the following states:

In Buffer Buffer is a temporary store which is created per user session. During this phase user can see meta-data obtained from the component's manifest and decide whether he really wants to store the component in the repository or, potentially, do some last pre-save adjustments. From this state component can either be persisted into store or deleted from the buffer. If user doesn't commit the component into the store, it is removed with the whole buffer when the session expires.

In Store While in store, all the meta-data and related information are available to clients via web user interface or REST API. Component remains in the store until deleted by user.

Deleted When deleted, all the component's data are removed from the system.

All events during which a component changes state can be intercepted via prepared repository plugin interface `ActionHandler`. Some of the most common interception methods are listed below:

- `beforeUploadToBuffer, onUploadToBuffer, afterUploadToBuffer`
- `beforeBufferCommit, afterBufferCommit`
- `beforePutToStore, afterPutToStore`
- `beforeDeleteFromStore, afterDeleteFromStore`

The first set of events is useful for consistency checks and preprocessing. The second and the third sets are most common for indexing, additional metrics computation and manifest file manipulation. The last set allows plugins to delete all the data they had created earlier. Plugins can make sure no orphaned records remain in the system when a component is removed.

4.3 Data Store

CRCE Data Store has been designed to be a universal component storage facility. Except for component binary store itself, it contains a meta-data database with own inner data representation. Own data model makes CRCE independent on any particular component model.

4.3.1 Store Types

CRCE has three separate stores, each for different type of information.

Component Store

Component store is the persistence layer for the actual components (i. e. OSGi bundle archives). Current implementation is file-based. It's structure is flat, one file per bundle. Each bundle is identified with unique hash.

Meta-Data Store

The meta-data storage contains all the descriptive information about components stored inside, which makes it the key feature of the repository. The meta-data are partly information contained in the bundles' manifest files, partly data computed by the repository itself (or its plugins). One of the goals of this paper is to provide extension to the meta-data store for persisting compatibility-related data.

The store is implemented on top of H2 Database Engine [15]. H2 is open-source relational database written in Java capable of running in either embedded or stand-alone mode. CRCE uses the first option.

Results Store

The detailed data contained in the Results store contain additional information about component's properties. It is meant for additional documents related to meta-data and components stored in CRCE. These data should allow decision-makers to base their verdicts on sound arguments [14].

4.3.2 Meta-Data Structure

Meta-data are organized in a generic structure. Its design allows it to represent components of more than one particular component model. The structure consists of five main entities:

Resource Resource represents a component stored in the repository. Every resource is described by list of its capabilities, requirements and properties. Although the generic structure is rather cumbersome to work with, it allows CRCE to support various component models without data structure modification.

Capability Capability is description of functionality a resource provides. In terms of OSGi capability can be an exported package or service. Basic information about the resource form another capability. The particular type is determined by namespace value. Capabilities can be organized in tree structure.

Requirement Requirements represent need of a capability of the same name. In OSGi they describe imported packages. They are also used as CRCE's component query engine.

Filter LDAP filter model used for component resolving.

Property Properties can be used to provide additional custom data about resources.

Attribute Attributes contain the actual information about Resources, Capabilities, Requirements or Properties. Examples of attributes are resource name, resource version or exported package name.

Product Model of an application composed of components. This entity is a future work and not yet implemented.

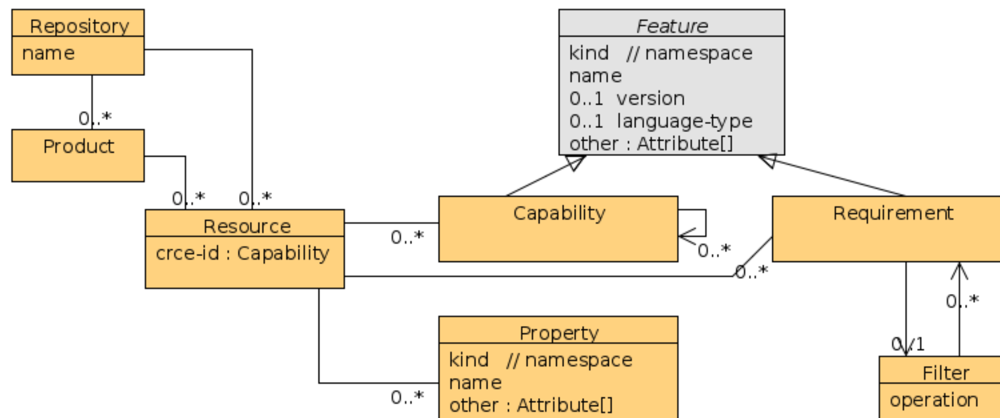

Figure 4.2: Example of CRCE capability hierarchy [39].

This logical meta-data structure can be represented in various ways - internal CRCE Meta-data API or externally accessible via XML (Extensible Markup Language) or JSON (JavaScript Object Notation) serialization.

Both data formats are used for description of structured data. The data formats are both human-readable and machine-readable. XML format organizes data into tree-like structures of elements, where each element can be described by multiple attributes and can contain a value or one or more other elements. See the appendix A for an example of data in XML format.

JSON format uses JavaScript notation for definition of objects. At the cost of lower semantic information the format is much more compact than XML. Data in JSON format are organized as sets of key-value pairs. A value can be an elementary data type, an array or an embedded data structure. Small footprint of the data format makes it a suitable medium for fast data exchange. Example of JSON document can be found in appendix B.

4.4 Client Access

Web User Interface Web interface is written in pure JSP. Users can browse components stored in the repository (as shown in figure 4.3) and view their meta-data. It is possible to upload new components or download or delete the existing ones.

There is also a view of all plugins enabled in the system.

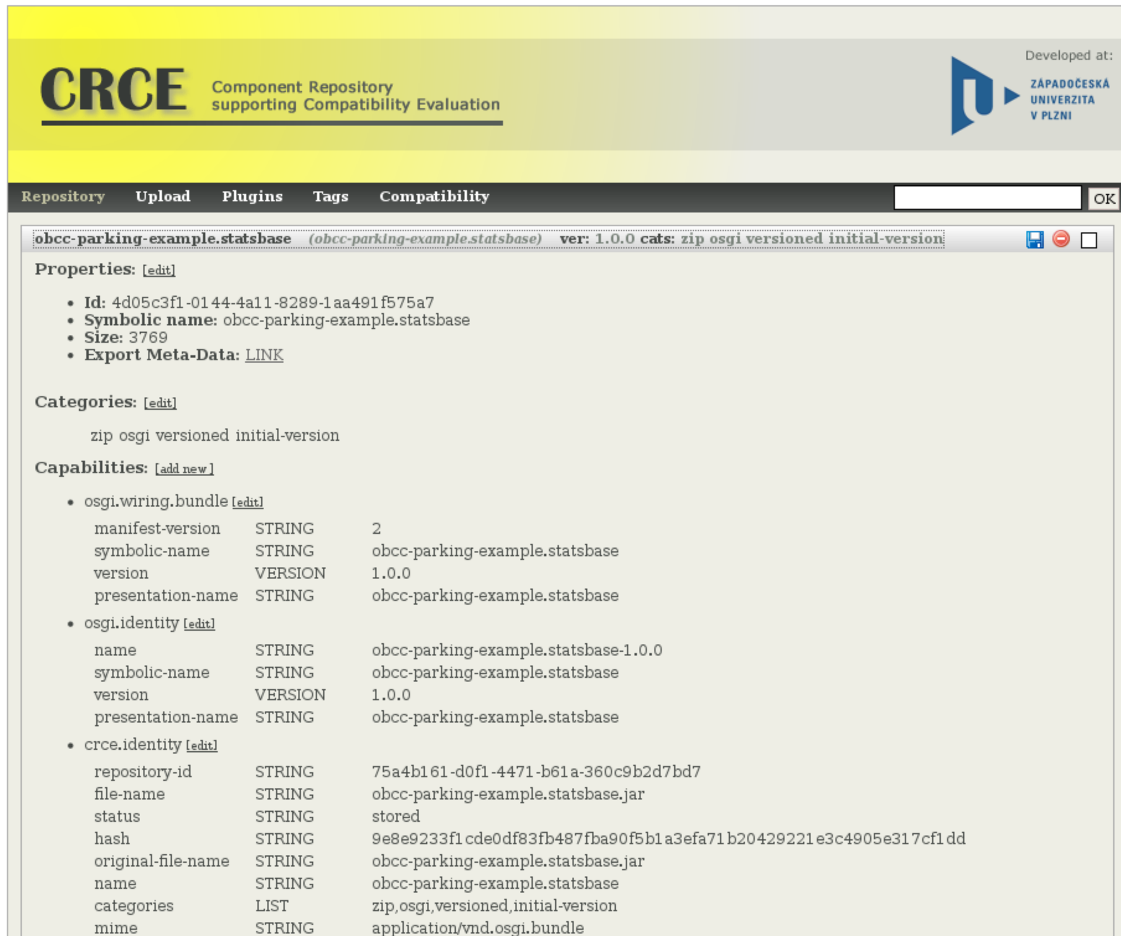

Figure 4.3: CRCE Repository browser view.

Web Services The other option of accessing repository's data are RESTful[1] web services. Output retrieved via services is suitable for computer-processing and allows clients to download and evaluate pre-computed data, thus saving their own resources.

Before this project, the API supported these scenarios:

- Download particular component version.
- Get component's meta-data.

Components could be searched for by both name and/or version.

Functionality and data format are explained in the chapter 7 further in this document.

[1]REST - Representational State Transfer.

4.5 Modules

Structure of the application is highly modular. Relevant parts are encapsulated into OSGi bundles. In most cases, API has been separated from the actual implementation into own component. This approach makes project maintenance easier. Also parts of functionality can be added, reimplemented or removed without high impact on the rest of the application's codebase. The following list brings a short overview of the project's modules at the initial phase of this master thesis:

Versioning Plugin Plugin with the compatibility-related functionality. This initial implementation used OSGi Version Generator (described in section 3.2.3) to compute proper version identificator for new components.

Integration Tests Module with all integration tests. Modular architecture brings certain restrictions on unit and integration testing. Tests of the functionality which depend on other OSGi bundles and therefore requires an OSGi framework runtime must be in a separate project. This approach allows developer to ensure all modules have been rebuild before the tests start.

Meta-Data API Modules contain API and implementation of the basic meta-data structure as described in section 4.3.2.

Meta-Data DAO Modules contain meta-data data access layer API and implementation supporting H2 relational database.

Meta-Data Indexer Indexer plugins are used for retrieving information about artefact from the bundle itself. Typical usage is to scan a newly uploaded component and save the meta-data (name, version, exported packages, etc.) into CRCE's meta-data representation.

Meta-Data JSON Module Module for mapping meta-data into JSON format.

OSGi Meta-Data Module Set of namespace definitions and utilities specific to the OSGi component format only.

Meta-Data Services Service layer for work with meta-data. Provides methods for convenient access to the generic meta-data API.

Plugin API Specifications of CRCE's plugin interface.

Repository API Module integrating resource store into the application. Contains both Buffer and Store implementations.

Resolver This module adds component search capabilities into the repository.

REST API This module starts a Jersey [17] container and handles web service requests, including data (de)serialization using JAXB.

WEB UI Module with web user interface written in JSP.

4.6 Project Status

CRCE project is available for download from its Assembla project pages under open-source Apache License v2.0.

CRCE URL: `https://www.assembla.com/spaces/crce`

Assembla project space contains project documentation in form of wiki pages, issue tracker and source code repository link.

Current release version of CRCE is 2.0.0.

5 Difference Meta-Data Repository

Aim of this work is to design and implement CRCE extension for storing compatibility-related component meta-data. Such extension should realize the main goal of CRCE - move the responsibility and most of the issues related to component compatibility verification from the applications and its developers to a component repository. This chapter describes the design and implementation of the extension, thus forming the core of this master thesis.

5.1 Analysis

As component providers upload their components into CRCE, the repository compares each of the components pair-wise to all of it's predecessors and stores the result. Afterwards, clients may query the repository for the compatibility meta-data and receive the answer quickly, with no computation involved. Clients can therefore quickly search for suitable component updates.

Dependency graph of a large application may become rather complicated - the application can be composed of dozens or even hundreds of components. In such case the developers require solid data to determine a working set of particular component versions. CRCE should be able to provide them with enough information about component revisions and their compatibility to make these decisions.

As described in chapter 3, component compatibility can be resolved from the aggregated difference value of the two components. Computation of the aggregated value from stored difference results would require the repository to do unnecessary computation for each request. Therefore the aggregated value for comparison of each component pair should be stored along with the difference details to allow fast access.

The logical view (see table 5.1) of the resulting store resembles a matrix where both columns and rows are component versions and fields represent difference meta-data between them.

version	1.0.0	1.0.1	1.1.0	2.0.0
1.0.0	X	**NON**	**INS**	**MUT**
1.0.1	NON	X	**INS**	**MUT**
1.1.0	DEL	DEL	X	**MUT**
2.0.0	MUT	MUT	MUT	X

Table 5.1: Logical View Difference Meta-Data Repository

It is enough to store only the upper triangle (printed in bold in the table 5.1) of the matrix since the values in the bottom part of the matrix are inverse to their respective counterparts.

Data stored in this way provide thorough description of differences between components and as such aren't bound to a particular use case or presentation model. The repository can therefore serve developers as knowledge-base for evaluation of component update safety, particular component choice or more. At the same time applications can use the very same pre-computed data (exposed e.g. via web services) to provide automated compatibility checks with almost no increase of requirements in both power and time.

As a result the meta-data repository should provide means to significantly limit integration risks connected to the component-based software development.

5.2 Use Case Summary

As a summary to the introduction of this chapter, the repository has been designed to fulfil the following scenarios:

Provide enough information to determine component substitutability. In special cases it is possible to replace components even if their versions indicate incompatibility. This can be caused either by use in specific context, where the particular incompatible API changes don't matter, or by specific sequence of changes between versions.

The Example 5.1 demonstrates the fact that particular chain of modifications may lead to compatibility between components with different major version identifier. Aggregated difference value between versions `1.0.0` and `2.0.0` is `DEL`, therefore `2.0.0` is not a replacement. However, the difference value between `1.0.0` and `2.1.0` is `INS`, which makes `2.1.0` a suitable upgrade candidate.

```
interface Logging { //version 1.0.0
    void write(String s);
    void writeLn(String s);
}
interface Logging { //version 2.0.0
    void write(String s);
}
interface Logging { //version 2.1.0
    void write(String s);
    void writeLn(String s);
    void writeLn(Level l, String s);
}
```

Example 5.1: Example of version and compatibility inconsistency.

Detailed information about component's differences also allows users to investigate reasons behind component's incompatibility.

Provide suitable replacement for a component. This functionality will allow clients to simple ask for an upgrade/downgrade of their components with additional criteria - nearest, highest, etc. In such case no details about differences between the versions are necessary. The relevant part of the meta-data record is therefore the final aggregated difference value.

5.3 Data Model

The data model of the repository has been designed with two main goals:

1. Provide fast access to the information required in the use-cases mentioned above.
2. Create generic description of the difference meta-data without dependency on any particular component model.

The resulting model consists of two main entities.

Compatibility

`Compatibility` interface represents comparison result of a pair of components. It contains basic information about the compared components, final difference value and detailed list of changes between the versions.

Each implementation must provide these attributes:

id unique identifier of the `Compatibility` instance.

resourceName Name of the newer version of the component. This field is mandatory.

baseResourceName Name of the original version of the component. This field is optional. If not present, it is expected to have value of the `resourceName` field. The name is going to be different e.g. in case of different provider.

resourceVersion Version of the newer component. This field is mandatory.

baseResourceVersion Version of the original component. This field is mandatory.

diffValue Aggregated difference value for this pair of components. Suitable for fast decision about compatibility two components.

diffDetails List of detailed change descriptions.

contract The contract this compatibility information is related to - can be of one of the following values: syntax, semantics, interaction or extra-functional.

The entity, without list of details, holds enough information for finding compatible replacement of a component given.

Diff

`Diff` is an entity containing detailed description of changes between two versions of components. Instances of the interface can form a tree-like structure. Each level of the tree hierarchy has a particular meaning - package, class, method, etc.

Each implementation must provide these attributes:

namespace Namespace attribute provides additional information about semantics of the particular `Diff` instance. This field is optional. Child elements inherit its value from parent unless specified differently.

name Name of the item the `Diff` instance is related to - e.g. name of a method. This field is mandatory.

value Value of the difference. This field is mandatory. The value equals to one of the difference classes described in chapter 3.

role Role of the item this `Diff` instance is related to. Either capability, requirement or property. This field is optional. E.g. for OSGi it is used at package and service level only.

level Level of the construct the `Diff` instance is related to - product, package, type, operation or attribute.

children Further details about the differences on the lower levels.

syntax Syntax this detailed information is bound to - in case of OSGi bundle interface comparison the syntax is Java.

Every `Diff` instance is either a child of another `Diff` instance or belongs to a `Compatibility` instance.

Both interfaces are located in new module `Compatibility API`. See appendix B for example of both entities in JSON format.

5.4 Persistence Layer

The data are very unlikely to ever change once computed. This stable nature of the data enables the repository to compute them once and store for future use.

5.4.1 Analysis

There were several technologies under consideration as persistent storage of the computed difference meta-data - simple file-based storage, relational database management system (RDBMS) or one of nowadays so popular NoSQL databases.

File-Based Storage

File-based storage would be easy to implement in one of the common serialization formats (XML, JSON). As best structure design appears to be a file-per-resource-name approach. Each file would contain all compatibility data related to the resource name. Such solution would allow searching for particular higher/lower version of the resource, thus fulfilling the described use-cases. However, the searching mechanism would have to be implemented as part of the system. It is a reasonable assumption that the implementation would require vast amount of resources without reaching the required quality.

Full list of considered aspects:

- Pros:
  - Simple implementation of basic storage mechanism.
  - No need for external technology apart from (de)serialization library.
  - Human readable.
- Cons:
  - No query engine. Its implementation would hardly reach the quality of query systems current databases have.

- Performance. Searching through large file brings high requirement on either memory (object parsing) or time (stream parsing). Any optimization would have to be self-implemented to suit the needs of CRCE.
- Any potential extension (e.g. compression) would have to be self-implemented or integrated into the system.
- No transaction system.
- No consistency protection.
- Concurrent access protection.

Relational Database Management System

Relational Database Management System [18, 19] is a common way of storing information. Due to fixed structure of the difference meta-data it a suitable candidate for this case as well. RDBMS solve most of the issues mentioned with the file-based storage above - they provide transaction support and lock mechanism to prevent concurrent modifications of data which might result in inconsistencies. Ensured referential integrity limits the possibility of creating orphaned records.

In addition, there is already a RDBMS in use within the repository to store resource meta-data. By using the H2 database system, we would avoid introducing dependency on another piece of technology into the application.

On the other hand, the tree-shaped structure of difference data would not be easy to retrieve. For large trees the operation would consist of large number of recursive queries.

Full list of considered aspects:

- Pros:
  - Built-in query language (SQL).
  - Performance optimization due to maturity of the RDBMS implementation.
  - Support of transactions.
  - Consistency protection.
  - Concurrent access protection.
  - Upstream developers are responsible for bugfixes and updates.
  - Already used in the system.
- Cons:
  - Static data structure.
  - Not suitable for working with whole trees of data.
  - External application required (in case other than embedded database was used).

There is one more fact to consider regarding the implementation of this solution. If the same RDBMS was used, it would seem logical to use the very same database to store data as the meta-data storage uses. Even more, single compatibility items should reference the particular resource they are related to. However, both these actions bring own issues.

Using the same database requires the implementation to handle migrations of schema it doesn't have fully under control. There is e.g. no insurance that some other module of the repository hadn't created a table of the same name earlier. Due to the modular architecture of CRCE and separation of API and implementation modules, it is a reasonable assumption there is a risk of future collision.

Referencing particular resource records from compatibility data on the database level then creates dependency between the modules on the implementation level. Such contract breaks the principles of CBSE as described in chapter 2.1.

NoSQL Databases

NoSQL databases can be divided into several main categories, which differ in their approach to data modelling. These categories are `document databases`, `graph stores`, `key-value stores` and `wide-column stores` [20].

Document Databases Document databases store information in structured documents. One of the used document formats is JSON. Each document can contain key-object pairs, where object can be a typed value, array or whole document. Unlike relational databases, all pieces of information related to a record are usually stored together, thus simplifying data access and reducing need for join operations [21]. Document databases usually provide strong query interface capable of retrieving documents based on value of any of their fields.

Structured data model is easy to map against commonly used object-based data model in current applications. Together with flexible data schema and ability to query on any field, document databases are a suitable choice for a large amount of systems.

Some of the commonly used document databases are MongoDB, CouchDB or Elastic-Search.

Graph Stores Graph stores represent data in graphs. Nodes and edges form a network describing single elements and relationships between them.

Such data model is perfect for cases where relationships between data are the most important issue - e.g. social networks.

Examples of graph databases are Neo4j or HyperGraphDB.

Key-Value Stores Key-value stores don't structure data in any particular way. Every record is a simple key-value pair and it is only possible to query data by the key.

Key-value databases are suitable for applications with large amount of unstructured and possibly polymorphic data. The key-based data access makes key-value stores one of the fastest and most scalable NoSQL databases. However, the lack of structure narrows down the amount of possible use-cases.

Examples of key-value stores are Riak or Redis.

Wide-Column Stores Wide-Column databases store data in tables, similar to the relational databases. However, the approach is completely different. Tables are stored by columns not rows, and the schema is flexible. Columns can be grouped into column families (e.g. First Name and Last Name can form a column family Name). Each row can have different amount of columns.

Wide-Column databases are suitable for large datasets and are used by companies like Facebook (Cassandra), Google (BigTable) or Yahoo (HBase).

NoSQL Choice From the above list it is obvious that not all of the NoSQL technologies are suitable for CRCE Difference Meta-Data Storage use-case. Due to the structured nature of the difference meta-data (as described in the section 5.3), document databases are the only viable choice among NoSQL database engines. They provide similar querying capabilities as RDBMS and remove the need for joins.

Full list of considered aspects of document-based NoSQL stores:

- Pros:
  - Flexible schema.
  - All pieces of information related to a record stored in one document.
    - This enables easier queries for the tree-structured difference data.
  - Fast due to simple operations.
  - Concurrent access protection.
  - Upstream developers are responsible for bugfixes and updates.
- Cons:
  - Requires external application.
  - Potential consistency risks.

5.4.2 Analysis Conclusion

The analysis revealed that the only viable solution is either relational database or document-based NoSQL database. After further consideration, it has been decided to choose the NoSQL path.

Here are the main reasons the decision is based on:

Flexibility Flexible data schema provides space for storing additional meta-data if needed.

Static Data The difference meta-data are a static description of changes between two component versions. These data are unlikely to change and therefore potential inconsistency of the data is not an issue.

No Relationships Each compatibility record is an encapsulated entity with no relationships to the others (except for the resource). Therefore the data would not take advantage of relational capabilities of a RDBMS. Issues related with binding of compatibility meta-data with resources on database level have been discussed in section 5.4.1.

No JOIN Operations The ability to persist documents inside other documents provides easy way of storing tree-structured details of version differences. The tree of details is going to be used as whole and therefore the application can benefit from the possibility to load all the data by a single query.

The only significant disadvantage is need of an external application server running next to the repository.

There were two main candidates for the storage implementation - MongoDB [22] and CouchDB [23]. Both projects offer similar functionality, have large community of users and both are undergoing heavy development. Yet they are mature and production-ready [24, 25]. Eventually, MongoDB has been selected, because the author of this project had more experience with the technology.

5.4.3 MongoDB

MongoDB is a document-based NoSQL database engine written in `C++`. Documents stored in MongoDB are basically JSON documents, however for serialization MongoDB uses Binary JSON (BSON) format. BSON is a binary serialization of JSON. The database supports indexes, document-style queries, map/reduce operations, data replication and is highly scalable.

MongoDB Java Driver

MongoDB Java Driver handles all communication between the application and database server. Base class of the driver is `MongoClient`. During its instantiation client needs to receive hostname and port on which the server is listening.

The driver itself is thread-safe and therefore it is possible and recommended to create only one instance of `MongoClient` per application [27]. Therefore the CRCE module

`CRCE Compatibility DAO - Mongo Implementation` contains class `DbContext`. The class holds singleton instance of `MongoClient` for use in the whole module.

Part of the package are classes for internal representation of JSON. Application needs to map its entities into these classes in order to store them in the database.

Official documentation [26] provides enough examples describing how to use the driver:

```
MongoClient mongoClient = new MongoClient(‘‘localhost’’, 27017);
DB db = mongoClient.getDB("test");
DBCollection coll = db.getCollection("testCollection");
BasicDBObject doc = new BasicDBObject("name", "MongoDB").
                              append("type", "database").
                              append("count", 1).
                              append("info", new BasicDBObject("x", 203).
                              append("y", 102));
coll.insert(doc);
```

Example 5.2: Java driver for MongoDB [26].

The Example 5.2 would insert the following JSON document into the database:

```
{
     "name" : "MongoDB",
     "type" : "database",
     "count" : 1,
     "info" : {
                x : 203,
                y : 102
              }
}
```

Driver version 2.11.3 has been used.

5.4.4 Implementation

Implementation of the CRCE Difference Meta-Data Store persistence layer is divided into two OSGi bundles:

Compatibility DAO API The module contains DAO interface.

Compatibility DAO - Mongo Implementation The module contains database context holder, mapping classes for entity (de)serialization and implementation of the DAO interface.

This architecture minimizes the costs of potential migration to different storage technology.

Interface

Data Access Object (DAO) interface `CompatibilityDao` is located in new module `CRCE Compatibility DAO`. It provides basic Create-Read-Update-Delete (CRUD) operations and simple low-level API for searching. The interface has been designed without any dependency on CRCE Meta-data API. The interface consists of the following methods:

- `Compatibility readCompability(String id)` - returns an instance of `Compatibility` with the given `id`, or `null` if none found.
- `Compatibility saveCompatibility(Compatibility compatibility)` - creates new `Compatibility` record in the database or updates an existing record with matching `id`. Returns saved (updated) instance.
- `void deleteCompatibility(Compatibility compatibility)` - removes the `Compatibility` record from the database.
- `void deleteAllRelatedCompatibilities(String resourceName, Version resourceVersion)` - finds all `Compatibility` records related to the resource with given resource name and version, and removes them from the database. It doesn't matter whether the referenced resource was the "new" or the "base" resource of `Compatibility` record.
- `List<Compatibility> listOwnedCompatibilities(String resourceName, Version resourceVersion)` - returns list of `Compatibility` records in which the resource with given name and version was the "new" one.
- `List<Compatibility> findHigher(String baseName, Version baseVersion, List<Difference> differences)` - returns list of `Compatibility` records which describe changes between the resource with given name and version and it resources with the higher version. Aggregated difference value of the records must be part of the `differences` argument.
- `List<Compatibility> findLower(String resourceName, Version resourceVersion, List<Difference> differences)` - same as the previous method, only records for resources with lower version than the one passed via argument are returned.

Mapping

There were to options to consider regarding data mapping from internal representation into JSON - implementation of own manual mapping or one of many Plain Old Java Object (POJO) mapping tools [26].

Some of the mapping tools have been rejected due to technological reasons (e.g. Spring Data MongoDB - CRCE doesn't use Spring framework). Many of the projects seem immature and it has proven itself difficult to make a choice which would promise existence of the mapping library in long-term. This is caused by rather young age of NoSQL technologies and the ecosystem around them.

Also, there are actually only two entities to persist, and their number is not expected to grow in the near future. Therefore it has been decided to implement own mapping of the entity fields into JSON.

The functionality is implemented as class `MongoCompatibilityMapper` with the following public interface:

- `DBObject mapToDbObject(Compatibility compatibility)` - takes `Compatibility` instance and maps it to `DbObject` interface.
- `Compatibility mapToCompatibility(DBObject source, CompatibilityFactory factory)` - takes `DbObject` received from the database and maps it to a `Compatibility` instance.[1]

Same methods exist for `Diff` and `Version`[2] interfaces.

5.5 Service Layer

Service layer forms a high-level API designed to accomplish use cases described in section 5.2. It is meant to be accessed by both presentation layer modules (web UI, REST API) and other CRCE plugins.

5.5.1 Implementation

Both interface and implementation classes are located in the same modules as entity representations - `CRCE Compatibility API` and `CRCE Compatibility Implementation` respectively.

Compatibility data are computed using OBCC tool and retrieved as CmpResult tree structure (described in section 3.2.5).The structure is parsed into `Diff` tree, which is attached to the related `Compatibility` instance. The `Compatibility` object is saved afterwards.

[1] Factory is used to create new instance of the `Compatibility` interface. This pattern is quite common in OSGi environments where implementation classes of the interface are usually not exported and therefore their constructors are not accessible.

[2] Version is CRCE's internal implementation of semantic versioning format described in section 3.1.1.

The process is pictured by Figure 5.1.

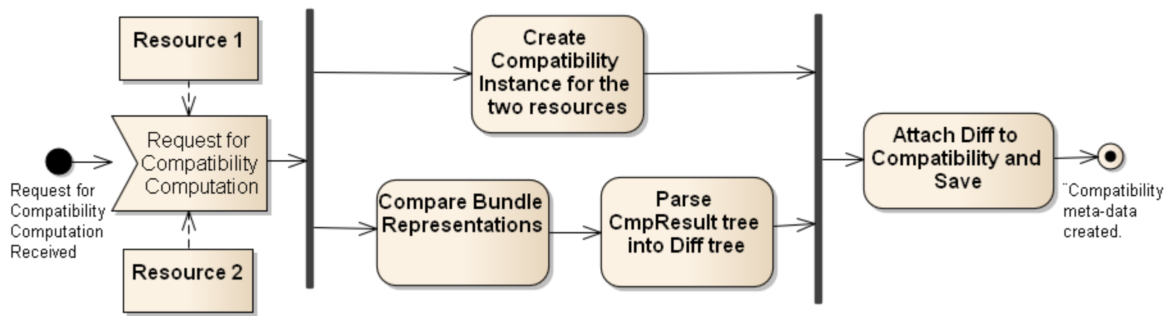

Figure 5.1: Process of creating a `Compatibility` instance.

CmpResult Parser

The parser transforms `CmpResult` tree into equivalent `Diff` tree. Since CRCE use-cases don't require as deep level of detail as OBCC tool provides, only a subset of the tree is mapped. For the purposes of `CompatibilityService`, the following items of `CmpResult` tree can be considered a leaf:

- method
- constructor
- attribute

The resulting `Diff` tree then doesn't contain description of particular low-level changes. E.g. it doesn't contain information about type change of a method's parameter, but only the resulting aggregated difference value for the method.

The parsing algorithm proceeds recursively through the `CmpResult` tree in depth-first manner. When it reaches a leaf or one of the items listed above, it stops and starts to map the particular branch of the tree into `Diff` instances.

It is easily extensible by providing additional recognized elements, if needed.

Interface

Interface methods represent concrete tasks to be accomplished (unlike generic nature of the DAO interface). Two interfaces have been established in order to separate search and management functions. `CompatibilitySearchService` provides ability to search for particular `Compatibility` instances and `CompatibilityService` adds services for compatibility computation or removal.

Interface `CompatibilitySearchService` consists of the following methods:

- `List<Compatibility> listUpperCompatibilities(Resource resource)` - service used for search for all compatibility data between the resource given and its higher versions.
- `List<Compatibility> listLowerCompatibilities(Resource resource)` - service used for search for all compatibility data between the resource given and its previous versions.
- `Resource findNearestUpgrade(Resource resource)` - finds and returns nearest compatible upgrade (resource with higher version) to the resource provided as argument.
- `Resource findHighestUpgrade(Resource resource)` - find and returns highest compatible upgrade to the resource provided as argument.
- `Resource findNearestDowngrade(Resource resource)` - finds and returns nearest compatible downgrade (resource with lower version) to the resource provided as argument.
- `Resource findLowestDowngrade(Resource resource)` - find and returns lowest compatible downgrade to the resource provided as argument.

Interface `CompatibilityService` adds the following management methods:

- `List<Compatibility> calculateCompatibilities(Resource resource)` - calculates compatibility data between the resource given and all of its previous versions stored inside CRCE.
- `List<Compatibility> calculateAdditionalCompatibilities (Resource resource, List<Resource> additionalBaseResources)` - calculates compatibility data between the resource given and the resources in the list provided as the second argument. This methods is part of the API to provide option to calculate additional differences outside of the regular component lifecycle.
- `void removeCompatibilities(Resource resource)` - removes all compatibility information related to the resource (it doesn't matter whether the resource has been the "new" or "base" item of the comparison).

5.6 Summary

This chapter described principles and implementation of CRCE module for creating and storing compatibility meta-data. The next chapter explains how the mechanism is integrated into CRCE processes.

6 CRCE Extensions

Several new plugins have been introduced into the CRCE during the implementation of this thesis. The Concurrency Plugin described in section 6.1 is a support plugin and its functions are usable by any other CRCE module. Compatibility plugins introduced in the section 6.2 bind functionality of Compatibility Data Store presented in the chapter 5 into the system.

6.1 CRCE Concurrency Plugin

CRCE Concurrency Plugin allows running tasks in background outside of regular component's lifecycle process. By default, the component lifecycle is strictly serial. Such approach simplifies solving of synchronisation issues related to the repository processes. However, it is not suitable for potentially time-demanding tasks (like e.g. the difference meta-data computation can be).

CRCE Concurrency module introduces unified interface for running background jobs, which can be used anywhere within the application. Unification of the mechanism is important in respect to maintenance costs and allows the application to control amount of allocated resources.

Individual tasks are ran on separate threads from a single thread pool. Size of the pool is configurable. As a result, application has full control over the amount of threads which can be running in the background (assuming developers use only the API to create background jobs).

Implementation

Implementation of CRCE Concurrency Plugin consists of base class for tasks, task runner implementation and exported service which represents public API to the job-running mechanism. The whole realization is located inside `CRCE Concurrency` module.

Task is a base class implementing Callable interface. Callers can use the class to implement their jobs. To ensure every job can be monitored, it contains basic information about the task (id, job description, caller identification).

User-implemented logic is wrapped by logging and state-monitoring mechanism. As a result, application is capable of asking for the job's status at any time during its existence.

TaskRunner is the executive part of the module. It is responsible for running the scheduled tasks and related resource management. It is a wrapper around `ExecutorService` interface provided by `java.util.concurrent` package, and particularly its implementation - `ThreadPoolExecutor` with fixed thread count size. The `ThreadPoolExecutor` runs the scheduled tasks on a single thread each. If all the threads from its pool are in use, the tasks wait until there is a free thread available.

Usage of thread pool removes the overhead of creating and destroying of thread constantly. The fixed size allows the system to control maximum number of threads running at one moment.

TaskRunnerService is public API of the concurrency plugin to the other modules. The interface allows scheduling of new tasks.

6.2 CRCE Compatibility Plugins

CRCE Compatibility Plugins are extensions responsible for automated component versioning and compatibility meta-data management. Each newly uploaded bundle receives semantically correct version identifier computed using OSGi Bundle Versioning tool described in section 3.2.3.

Difference meta-data are created for all the predecessors of the newly added bundle. The meta-data are acquired and saved using Compatibility Storage described in the chapter 5.

Both the plugins work on the precondition that bundles are uploaded to the repository in ascending order in respect to their revision order.

The plugins currently support only OSGi bundle resources.

6.2.1 Automated Versioning Plugin

Automated Versioning Plugin provides newly uploaded bundles with semantically correct version identifier in respect to the bundle's previous revision. The original version of the plugin had been semi-functional and incompatible with current version of Meta-Data API. Therefore the plugin had been reimplemented by the author of this thesis during the project.

The plugin intercepts component's lifecycle (described in section 4.2) at its `uploadToBuffer` and `beforePutToStore` phases.

Before a previously unversioned[1] component is uploaded to buffer, it is prepared for further versioning process - its name and version are stored into component's meta-data with original-name and original-version keys respectively.

Before the component is persisted into the store, the actual versioning process takes place. The highest version of the bundle with the same symbolic name is found to serve as base resource for the comparison. If none is found, the resource is marked with `initial-version` tag and its original version is kept. If there is a base resource, OBCC tool is used to acquire difference meta-data and the aggregated difference value is used to determine new version identifier (see Table 3.3 for detailed description of `difference value - version identifier` relationship).

Either way, resource is tagged with `versioned` tag and its meta-data are saved. Such timing of the versioning process (in the beforePutToStore phase of component's lifecycle) has two main impacts:

1. User can see the original information in the UI overview before he submits the component into the repository.
2. Version information is changed before the component is saved into the store (and becomes accessible through the repository's API). Therefore it is not possible for clients to get the original, potentially incorrect, meta-data.

The plugin has been implemented `VersioningActionHandler` class within the `CRCE Versioning Handler` module.

6.2.2 Compatibility Plugin

Compatibility Plugin ensures that compatibility meta-data are created between a new component and all its predecessors stored inside the repository, and, on the contrary, that no meta-data remain for a particular bundle in the Compatibility Data Store, when the bundle is removed from the repository. The first task is achieved by intercepting `afterBufferCommit` phase of the component's lifecycle (described in section 4.2), the latter is done within the `beforeDeleteFromStore` phase.

In the `afterBufferCommit` phase the plugin starts computing difference meta-data for every committed resource and its predecessors. The computation is done using the compatibility service layer introduced in chapter 5. The task can be rather time-demanding, therefore it is performed in background using the CRCE Concurrency Plugin described

[1]The component hasn't been marked by CRCE as "versioned" and therefore its version identifier can't be trusted.

in section 6.1. This means that a component can be stored in the repository, but its compatibility meta-data may not be accessible yet. Such state doesn't cause any consistency problems, as the component simply won't show up in compatibility-related search results.

The `afterBufferCommit` phase takes place after all the components in the buffer have been stored into the store and occurs only once for the whole set of committed resources (this is the main difference to the `afterPutToStore` phase which happens for each resource individually as they get committed into the store). The timing eliminates any possible indeterministic problems caused by the order in which the components are submitted into the store from the buffer.

When a component is deleted from the store, all meta-data related to it must be deleted. Compatibility meta-data are removed during the `beforeDeleteFromStore` phase to ensure the bundle doesn't show up in compatibility-related search results when the component binary isn't present in the system anymore.

The plugin has been implemented as `CompatibilityActionHandler` within the `crce-handler-versioning` module. The background job template for concurrency module has been implemented as `CompatibilityComputationTask` inside the same module.

6.3 Web User Interface

Web user interface has been extended by a single page which users can use to search for compatible revisions of a bundle. This allows users to get quick overview of revisions available in the repository. The page displays list of all versions of a particular component and the difference values in respect to the selected version.

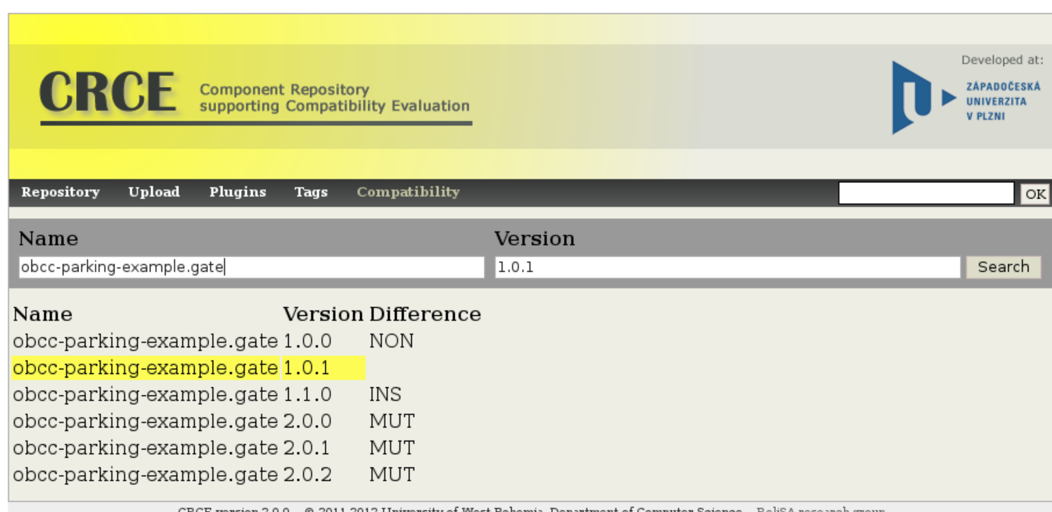

Figure 6.1: CRCE Version compatibility search view.

7 Web Service Module

CRCE web service module allows client applications to access pre-computed component meta-data stored within the repository. The clients using the repository's services don't need to compute the meta-data by themselves. This makes the web service API one of the key features of CRCE.

The meta-data are returned in XML format, which is readable to both client applications and humans. The output is therefore suitable for automated compatibility evaluations or deeper manual analysis of component's capabilities and requirements.

This chapter describes module extension created as part of this master thesis project. The original functionality is listed in chapter 4.4.

7.1 RESTful Web Services

Representational State Transfer (REST) is an architectural pattern for design and implementation of web service interface. It is a resource oriented model, which has become a major API design pattern, especially because it is much simpler to use than its predecessors - SOAP- or WSDL-based interfaces.

RESTful web service interfaces should follow these basic principles [35]:

- Expose directory structure-like URIs.
- Use HTTP methods explicitly.
- Be stateless.

Directory-Like Structure

Structure of the interface determines how user-friendly it is going to be. Each URI should represent a particular resource or collection of resources. Tree-like structure of URIs, resembling directories of a file system, increase intuitiveness of the API significantly.

In the Example 7.1, the first URI represents collection of all meta-data stored in the repository, the other represent a particular resource either identified by name and version or by its id.

```
http://www.crce.com/rest/metadata
http://www.crce.com/rest/metadata/{resourcename}/{version}
http://www.crce.com/rest/metadata/{resourceid}
```

Example 7.1: Resource structure compliant with REST principles.

Explicit Use of HTTP Methods

> One of the key characteristics of a RESTful Web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616 [35].

As a result it is possible to use mapping of basic CRUD operations to particular HTTP methods as a guideline for API design:

- POST, PUT - create or update a resource on the server.
- GET - retrieve a resource.
- DELETE - remove or delete a resource.

Difference between POST and PUT is in idempotence of the PUT method. Result of repeated PUT calls should always be the same. Therefore it is allowed to use PUT to create new resource with particular ID given. However, if want to add a resource and let the server generate its id, POST method should be used instead. See [36] for more details.

Stateless Web Services

Web services should be implemented as stateless in order to increase their scalability. Stateless web service implementation forces clients to send all required data with their requests, which can be subsequently distributed among multiple server instances easily.

7.2 CRCE Web Service API

CRCE Web Service API is designed to fulfil repository's goals in respect to its clients.

7.2.1 Original API

The original API includes functionality connected to basic component meta-data. The API implements several scenarios related to meta-data or component retrieval. List of scenarios and examples have been taken from [16]:

Scenario BO-SB[1]: Obtain a Single Bundle

Clients may ask for particular component binary either by its id or by its name and version. The server responds either with the bundle, or with 404 response status in case no such component has been found.

```
GET /bundle/id
GET /bundle?name=name&version=version
```

Example 7.2: Example requests for scenario BOSB.

Scenarios BO-SM/AM[2]: Obtain a Single/All Bundle Meta-Data

Clients may ask for either particular component's or all meta-data either by its id or by specific filter query. The server responds with the requested meta-data in XML format. Future implementation should also provide JSON serialization.

Each request can be supplied with extra parameters specifying which meta-data the client wants to receive:

- **core** [default] - all *.identity and *.content capabilities (id, provider, name, version, categories, file size, URI)
- **cap** - all "capability" elements
- **cap=name** - all capabilities with the given name
- **req** - all "requirement" elements
- **req=name** - all requirements with the given name
- **prop** - all "crce:property" elements
- **prop=name** - all property elements with the given name

[1]BO = Basic Operation. SB - Single Bundle.
[2]SM = Single Meta-Data. AM = All Meta-Data.

Example 7.3 shows several options of the scenario API usage:

```
GET /metadata        //meta-data for all bundles stored in the repository
GET /metadata/id     //meta-data for a particular bundle
GET /metadata/id?core&prop     //core capabilities and all property elements
                                 of the resource with the given id
GET /metadata?filter=(expr)    //metadata of bundle or bundles, that fit
                                 the filter criteria
```

Example 7.3: Example requests for scenarios BOSM/AM.

Scenario BO-OM[3]: Obtain Meta-Data About "Other" Bundles

To minimize number of requests to the server, clients may cache the meta-data of bundles stored in the repository. In such case, they need to update their information once in a while. This can be achieved by sending list of resources they are aware of and receive list of newly added bundles (and/or list of the resources which have been deleted in the meantime). The returned information can be divided into three categories:

- Newly added bundles. Client doesn't know about them until the update.
- Removed bundles. Client doesn't know they aren't available at the server anymore (this functionality hasn't been implemented yet, the repository doesn't remember deleted bundles).
- Unknown bundles. Bundles which client knows about, but server doesn't recognize them (they have never been stored in the repository).

Client sends the list of bundles in meta-data XML structure described in section 7.3.

```
POST other-bundles-metadata HTTP/1.1
Host: www.crce-repository.kiv.zcu.cz
Content-Type: application/xml;charset=UTF-8
Meta-Data XML Serialization
```

Example 7.4: Example request for scenarios BOOM.

[3] OM = Other Meta-Data.

Scenario SM-FP[4]: Find Providers of Given Capability

Certain requirements may be fulfilled by several components from different vendors. A client may therefore ask the repository for list of bundles which provide a particular set of capabilities.

```
POST provider-of-capability/ HTTP/1.1
Host: www.crce-repository.kiv.zcu.cz
Content-Type: application/xml;charset=UTF-8

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<requirement namespace='osgi.wiring.package'>
    <attribute name='name' value='cz.zcu.kiv.parking.gate' />
    <attribute name='version' value='1.0.0' op='less-than' />
    <directive name='filter'
       value='(&(osgi.wiring.package=cz.zcu.kiv.parking.gate)
       (version&gt;=1.0.0))'
    />
</requirement>
```

Example 7.5: Example request for scenario SMFP.

7.2.2 API Extensions

This master thesis has introduced additional interfaces and puts in use compatibility data discussed in the previous chapters. Implementation of the scenarios is explained in section 7.4.

Scenario SM-RS[5]: Which Bundle Versions Can Be a Replacement

One of the goals of the repository is to provide clients with the information which components updates are compatible with the version they are currently using. When a client asks for compatible bundles by providing identifier of a bundle (which exists in the repository), the server replies with list of meta-data of strictly compatible replacement bundles.

Clients can optionally provide specification of the operation they intend to make:

- **upgrade** [default] = returns the lowest compatible version higher than the original one
- **downgrade** = returns the highest compatible version lower than the original one
- **highest** = returns the highest compatible version higher than the original one
- **lowest** = returns the lowest compatible version lower than the original one

[4]SM = Scenario Meta-Data. FP = Find Provider.
[5]RS = Replace Single.

- **any** = returns any compatible version different from the original one

Compatibility of bundles is decided using data stored in CRCE Difference Meta-Data Store described in chapter 5.

```
//the following calls are equivalent
GET replace-bundle?id=id
GET replace-bundle?id=id&op=upgrade
```

Example 7.6: Example request for scenario SMRS.

Scenario BO-US[6]: Upload Single Bundle into the Repository

While upload of bundles via Web UI gives users additional options of data control, it can be rather cumbersome sometimes. Therefore an interface for bundle upload has been introduced.

In such scenario user doesn't have any options to make final adjustments to the uploaded bundles. The bundle proceeds through the whole lifecycle (including buffer) and is saved into the store.

```
POST /bundle HTTP/1.1

Content-Type: multipart/form-data;
```

Example 7.7: Example request for scenario BOUS.

[6] US - Upload Single.

7.3 Meta-Data XML Representation

CRCE Web Service API returns meta-data in XML representation of the data model described in section 4.3.2.

7.3.1 Core Meta-Data

The original API provided only core meta-data of components. Those included component description, lists of capabilities, requirements and other properties. The meta-data contained no compatibility-related information.

Basic information about the component, like component name or version, are described by `crce.identity` and `<component type>.identity` (e.g. `osgi.identity`) capability namespaces as shown by the Example 7.8.

```
<capability namespace="crce.identity" id="...">

     <attribute name="repository-id" value="1"/>
     <attribute name="file-name" value="obcc-parking-example.gate.jar"/>
     <attribute name="status" value="stored"/>
     <attribute name="hash" value="..."/>
     <attribute name="original-file-name"
                value="obcc-parking-example.gate.jar"/>
     <attribute name="name" value="obcc-parking-example.gate"/>
     <attribute name="categories" value="zip,osgi" type="List"/>
     <attribute name="mime" value="application/vnd.osgi.bundle"/>
     <attribute name="uri" value="file:..." type="URI"/>
     <attribute name="size" value="19892" type="Long"/>
     <attribute name="url"
                value="http://.../obcc-parking-example.gate-1.1.0"/>

</capability>
<capability namespace="osgi.identity" id="...">


     <attribute name="name" value="obcc-parking-example.gate-1.1.0"/>
     <attribute name="symbolic-name" value="obcc-parking-example.gate"/>
     <attribute name="version" value="1.1.0" type="Version"/>

</capability>
```

Example 7.8: Example of core metadata identity.

CRCE identity contains general description of the component stored inside the repository - component name, filename of the binary, mime type, file size, uri of the resource, etc.

Type identity (`osgi.identity` in the example) then contains descriptive information related to the particular component type. In this case version, following the OSGi semantic versioning standard (described in chapter 3.1.1), `symbolic name` of the bundle and `name`

identifier which is unique within the repository and consists of symbolic name and version information.

Provided functionality, like exported packages of OSGi bundles, is described as set of capabilities with particular namespace. Example 7.9 shows description of an exported package:

```
<capability namespace="osgi.wiring.package" id="...">

    <attribute name="name"
               value="cz.zcu.kiv.osgi.demo.parking.gate.statistics"/>
    <attribute name="version" value="1.1.0" type="Version"/>
    <directive name="uses"
               value="cz.zcu.kiv.osgi.demo.parking.carpark.status,
                      org.slf4j,
                      cz.zcu.kiv.osgi.demo.parking.statsbase"
    />

</capability>
```

Example 7.9: Example of core metadata capabilities.

The capability represents a particular exposed package. The XML element contains the name of the package, its version and list of external packages used by the exported package.

Required functionality, like imported packages of OSGi bundles, is described as set of requirements with particular namespace. Example 7.10 shows description of an imported (required) package:

```
<requirement namespace="osgi.wiring.package" id="...">

    <attribute name="name"
               value="cz.zcu.kiv.osgi.demo.parking.carpark.status"/>
    <directive name="text"
               value="Import package
                      cz.zcu.kiv.osgi.demo.parking.carpark.status"
    />

</requirement>
```

Example 7.10: Example of core metadata requirements.

The requirement represents a particular package that must be present in the system in order for the bundle to work.

7.3.2 Compatibility Meta-Data

XML representation of the compatibility meta-data, implemented as part of this master thesis project, has the same structure as the inner representation of the data described in section 5.3 and is connected to the core meta-data via `Property` field with namespace `crce.compatibility`.

In the Example 7.11, one can see version of the base resource, type of contract the difference information is related to, aggregated difference class value and detailed information about changes between the versions.

```
<compatibility base-version="1.0.1" contract="syntax" value="INS">

      <diff level="PACKAGE" name="cz.zcu.kiv.osgi.demo.parking.gate.statistics"
            namespace="osgi.wiring.package" role="CAPABILITY" syntax="java" value="INS">
           <diff level="TYPE"
                 name="cz.zcu.kiv.osgi.demo.parking.gate.statistics.GateStatistics"
                 value="INS">

                <diff level="OPERATION" name="vehiclesArrived" value="INS"/>
                <diff level="OPERATION" name="vehiclesDeparted" value="INS"/>
           </diff>
      </diff>
      <diff level="PACKAGE" name="cz.zcu.kiv.osgi.demo.parking.gate.vehiclesink"
            namespace="osgi.wiring.package" role="CAPABILITY" syntax="java" value="INS"/>
      <diff level="PACKAGE" name="cz.zcu.kiv.osgi.demo.parking.lane.statistics"
            namespace="osgi.wiring.package" role="CAPABILITY" syntax="java" value="INS">
           <diff level="TYPE"
                 name="cz.zcu.kiv.osgi.demo.parking.lane.statistics.LaneStatistics"
                 value="INS">

                <diff level="OPERATION" name="getVehiclesPerInterval" value="INS"/>
           </diff>
      </diff>

</compatibility>
```

Example 7.11: Example of difference metadata.

7.4 Implementation

Implementation of CRCE Web Service module consists of the actual web service endpoints and of the XML (de)serialization mechanism. Both are implemented in the module `CRCE Rest`.

Web service endpoints are implemented according to scenarios specified in the sections 7.2.1 and 7.2.2. Implementation is done using Jersey framework [17]. Jersey is a reference implementation of Java RESTful Web Service API standards (JSR-311 [37] and JSR-339 [38]).

Data serialization is realized by mapping of data from internal representation into the data model described in section 7.3. For serialization has been chosen Java Architecture for XML Binding (JAXB), because it provides easy way to serialize Java model into XML.

To avoid any tight-coupling between inner data representation and the XML output, the XML meta-data representation is specified in separate XML Schema Document (XSD) files. During module build, a maven plugin is used to generate properly annotated Java classes from the schema files. These classes are then used for (de)serialization of data.

API Extensions Implementation

Which Bundle Version can be a Replacement scenario uses service layer of the Difference Meta-Data Repository, described in section 5.5, to acquire suitable resource according to the request parameters. All meta-data of the particular resource are created and returned to the user in XML format.

Upload Single Bundle into the Repository scenario implementation ensures that the bundle proceeds through the same process as when uploaded via Web UI (see chapter 4 for details). Buffer is created for each request, the uploaded bundle is saved into it and the buffer is immediately committed afterwards.

8 Testing

Testing of the solution was done in two respects - correctness and performance. The first ensured the repository computes and returns correct difference meta-data and the latter provided an overview of computation times and confirmed the data had been available after reasonable period of time since upload.

8.1 Correctness

The correctness testing proved that the repository provides valid data and functions properly. The testing was done manually using a set of small components which had been created for component substitutability testing by the thesis supervisor, Doc. Ing. Přemysl Brada, MSc., Ph.D. The set is available as digital attachment to this thesis[1].

The set is a simple model of a parking lot and is divided into the following components: `StatsBase` (base bundle for statistics), `Dashboard`, `Gate`, `CarPark`, `TrafficLane` and `RoadSign`. Each of the components comes in several revisions. The set of revisions contains changes which can be classified by the difference classes described in chapter 3 so that each of the classes occurs at least once. Size of the components' public API enables manual verification of the computed data. All these facts make the set suitable for validation of correctness of the implemented solution.

Test Scenario

Each of the bundles was uploaded into the repository one revision after another. After each upload, the computed difference meta-data were manually compared to the actual changes in the particular component's source code. Afterwards the created version identifier was verified against the aggregated difference value of the component comparison.

The difference meta-data verification was done for results by the both database API and REST API. Thus both Difference Meta-Data repository and REST API extensions were tested.

[1] The set can be also downloaded from https://github.com/pbrada/obcc-parking-example.

Results

The testing showed that the difference meta-data repository works as expected with two exceptions:

1. The OBCC Tool used to compare the bundles is unable to detect internal implementation changes within compared bundles. In consequence, repetitive upload of the same bundle revision results in incrementation of the micro part of the bundle's version identifier (whereas the correct behaviour would be to recognize that the bundle is already present in the repository).
   This problem is difficult to solve as it is troublesome to find a fully reliable method of determining whether two bundles are exactly the same in respect to their implementation. Due to non-existing effect of the micro part of version identifier on component compatibility, it has been decided to leave the responsibility of avoiding duplicate bundle uploads on the users of the repository.

2. The testing confirmed the issue which results in false minor version identifier incrementation due to move of a public API method from a child class into a parent class. This is again a bug of the underlying OBCC Tool, described in section 3.2.6 and reported to the project's issue tracker.

Except for the issues mentioned above, the testing proved that the difference meta-data repository provides correct data and can assist developers and system administrators with component substitutability checks.

8.2 Performance

Performance testing was done in order to evaluate the average time which elapses before the compatibility data are available after component upload.

The testing was done using six sets of components and their updates. The components were bundles selected[2] from the Glassfish Application Server, which were also used in the study [39]. The components had been chosen so that they varied in size (in respect to both file size and number of exported packages) and there were large number of updates available.

[2] http://relisa-dev.kiv.zcu.cz/data/experiments/crce-2013-12/

The table 8.1 shows the details of the component set. The size and number of exported packages are the values of the first revision in the set. The values do not vary greatly in the other revisions.

Name	Number of Updates	Size [kB]	Exported Packages
config-types	139	6.8	1
osgi-adapter	140	54.6	0
dataprovider	90	101.4	2
asm-all-repackaged	141	104.8	4
bean-validator	141	553.6	37
webservices-osgi	44	11863.5	463

Table 8.1: List of components used for performance tests.

Test Scenario

During the tests, a single component and all of its updates were uploaded into an empty repository instance. The components were uploaded via web-service API without any delay, thus putting the instance under heavy load. Computation times were measured in the process.

The times were measured directly inside CompatibilityService interface implementation, in particular in the method `List<Compatibility> calculateCompatibilities(Resource resource)`. The following metrics were measured:

- Duration of single pair computation.
- Duration of comparison of a resource to all its previous revisions.
- Total duration of the scenario.

Measurements were done by logging times (in `ms`) of the beginning and the end of each single pair / revision set computation. Total duration of the scenario was computed as difference between the start of the first set and the end of the last set.

This scenario was repeated five times for each of the components. During the value post-processing, average values and standard deviations for each of the mentioned metrics had been computed. The post-processing was done automatically using a Python script.

Test Machine Specifications

The tests were done on a machine with the following specifications:

CPU Intel Core i7 3612QM Ivy Bridge, 2.1 GHz

RAM 8 GB

OS Gentoo Linux, 64bit, kernel 3.11.6

Java Version 1.7.0_51, Oracle (Sun) JVM

MongoDB Version 2.4.9

Results

The results can be viewed from three angles:

1. Time to compute difference meta-data for a single pair.
2. Time to compute difference meta-data for all previous revisions.
3. Total time to run the scenario.

Single Pair Comparison

Table 8.2 shows the average times to compare single pair of components and the standard deviation of the population of times gathered.

Component	Average Value [ms]	Standard Deviation [ms]
config-types	8.85	64.81
osgi-adapter	42.07	97.02
dataprovider	76.23	151.16
asm-all-repackaged	148.22	130.07
bean-validator	373.18	179.10
webservices-osgi	33113.22	3052.00

Table 8.2: Time to Compute Difference Meta-Data for a Single Pair of Components.

Rather high values of standard deviation for smaller components are most likely caused by synchronization collisions during database access. Lot of components are uploaded and computed quickly, putting heavy load on the database in a multi-threaded environment (by default, the repository used all four available cores for meta-data computation).

Forcing use of a single thread had a positive effect on both average value and standard deviation of the computation times.

Table 8.3 shows comparison of average times and standard deviation for four and single thread respectively:

Component	Avg. Value	Std. Deviation	Avg. Value	Std. Deviation
	4 threads		1 thread	
	[ms]			
osgi-adapter	42.07	97.02	27.99	48.24
asm-all-repackaged	148.22	130.07	103.11	50.45

Table 8.3: Comparison of Computation Times - 1 vs 4 threads.

Comparison of a Component to All its Previous Revisions

Duration of time required to compute difference meta-data between a newly uploaded bundle and all of its previous revisions depends on several factors:

1. Size of the bundle (in respect to amount of exported items).
2. Number of previous revisions.
3. Effect of collisions mentioned above.

The figure 8.1[3] shows clearly that for larger bundles like `bean-validator` the computation time grows quite steadily.

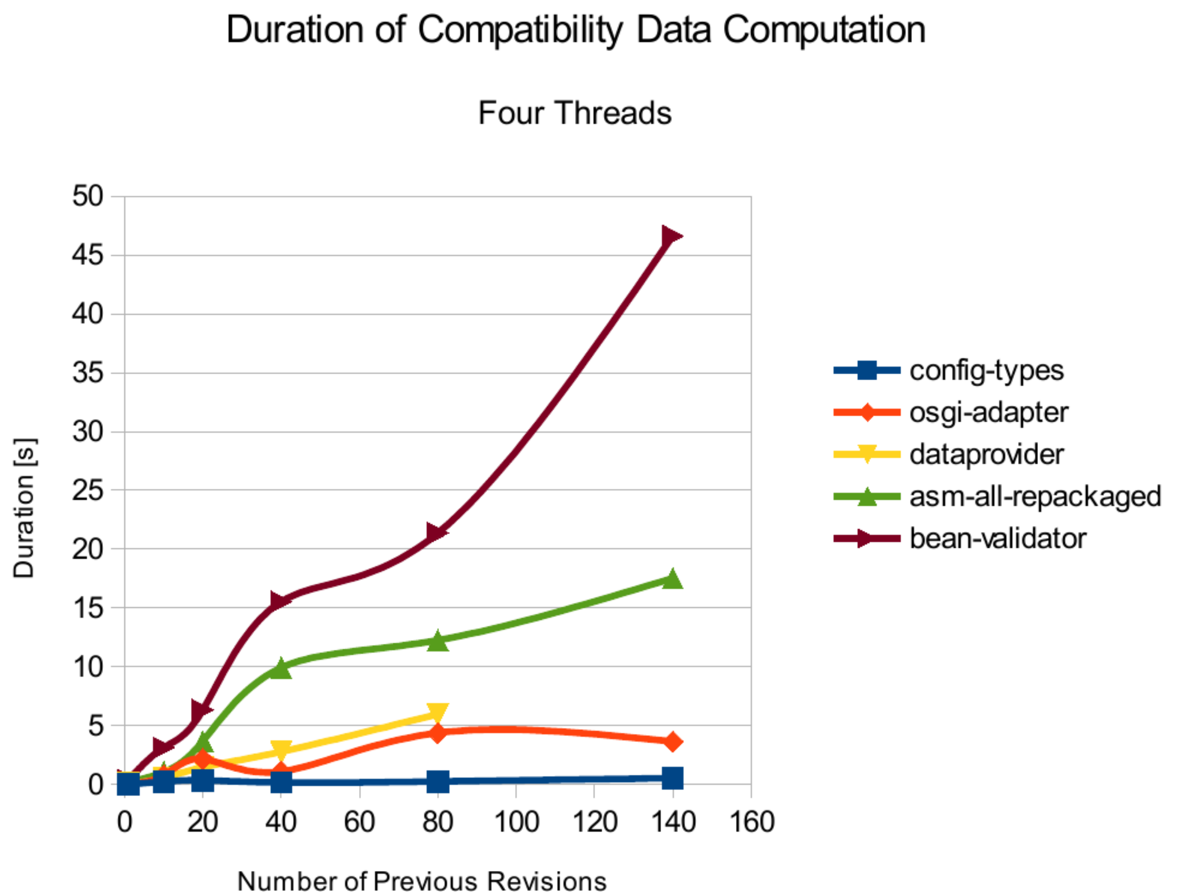

Figure 8.1: Computation Times Based on Number of Previous Revisions - four threads.

For a really like large components like webservices-osgi the growth is basically linear (see table C.1 in Appendix C). However, when uploading large amount of small bundles in a short period of time, the collisions affect total duration of computation to such degree that the difference computation time is almost irrelevant (visible in case of `osgi-adapter`).

[3]See source table C.1 with measured data in Appendix C.

Such behaviour makes the time required to do the computation almost unpredictable (due to extremely high variance values).

Figure 8.2[4] demonstrates how the effect disappears when only a single thread is used to compute the difference meta-data. The times become almost linear for the both tested components.

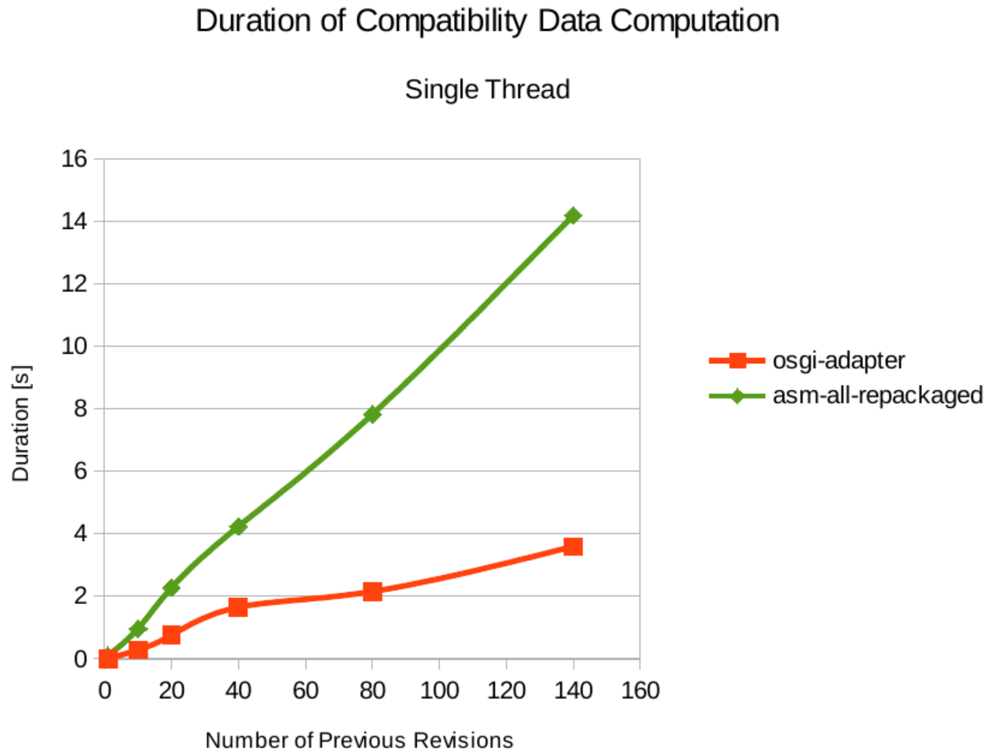

Figure 8.2: Computation Times Based on Number of Previous Revisions - single thread.

Total Duration of Scenario

Despite longer average times to compute single-pair comparison (or comparison of a bundle with all its previous times) in a multi-threaded environment, the overall computation time of the whole test scenario was shorter, as shown in table 8.4.

Component	Avg. Scenario Duration [s]	
	Single Thread	Four Threads
osgi-adapter	396.8	195.1
asm-all-repackaged	1228.4	452.7

Table 8.4: Scenario duration - comparison of single thread vs four threads.

[4]See source table C.2 with measured data in Appendix C.

9 Conclusion

The first part of this thesis introduces basic principles of component-based software engineering with focus on OSGi component model and discusses compatibility-related problems introduced by modular nature of the component-based applications. The author became familiar with general concepts of component substitutability verification based on type comparison and with tools implementing those principles.

The second part of the thesis contains details about the design and implementation of a Component Repository supporting Compatibility Evaluation (CRCE) extension for difference meta-data storage.

The difference meta-data are used to compute semantically correct version identifier of newly uploaded components. In addition, the existing RESTful web-service API was extended to return the difference data along with the rest of the component meta-data.

The solution implementation follows both OSGi principles of component design and existing CRCE project architecture patterns. Data layer of the compatibility meta-data repository was built on top of MongoDB, a document-oriented NoSQL database.

The first goal of the extension was to provide means to create and store meta-data describing differences between component revisions. The second goal was to expose the meta-data to both developers and client applications. All of the goals were achieved, CRCE is now capable of fulfilling its main task: to assist developers and system administrators with resolving component compatibility issues. CRCE can be used as source of information about component compatibility when picking suitable components for an application. Furthermore, clients may ask CRCE for compatibility verification before replacing the current component version with an update. The repository is capable of recommending a compatible component replacement.

The schema-free storage technology suits the deep recursive structure of the data well and is ready for arbitrary extensions in the future. This is especially important due to the fact that while the solution was developed and tested with OSGi framework components, it is designed as generic storage independent of any component model in particular.

The chosen data format, in which a single database record contains complete information about differences between a pair of components, leads to easy data access and provides simple way to create queries.

Additional discussion should be lead about implementation of background job for difference meta-data computation. As shown in chapter 8, the current implementation is focused

on quick processing of data under heavy load at cost of longer and rather unstable times of the process of comparing a single pair of components. Different behaviour might be preferred under particular conditions. However, it should also be noted that the presented results may not be very accurate due to rather low size of the data population the statistics is based on. Larger amount of data should be gathered for a deeper analysis.

Implementation of CRCE Meta-Data API should be extended to allow creating connections between basic component meta-data and additional entities (including the aforementioned difference meta-data). Currently, the inner representation of a component has no reference to the difference meta-data related to the particular component.

The author of this master thesis recognises three main pieces of knowledge gained from work on the project. The author became familiar with principles of component-based software engineering and with implementation of applications using the OSGi framework. Furthermore he got deeper understanding of binary and source compatibility in Java programming language. As the most interesting piece of knowledge the author considers the principles of type-based comparison used for component substitutability verification.

Bibliography

[1] OSGi Alliance. *OSGi Alliance,* 2013
http://www.osgi.org

[2] Felix Bachman et al. *Volume II: Technical concepts of component-based software engineering.* Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[3] Zuzana Burešová. *Optimalizace a rozšíření porovnávání OSGi komponent.* Diplomová práce, Západočeská univerzita v Plzni, 2012.

[4] OSGi Alliance. *OSGi Service Platform Core Specification.* Release 4, Version 4.3, April 2011.
http://www.osgi.org/Download/Release4V43

[5] The Eclipse Foundation. *Eclipse Equinox,* 2014.
http://eclipse.org/equinox/

[6] The Apache Software Foundation. *Apache Felix,* 2013.
http://felix.apache.org/

[7] The Knoplerfish Project. *Knoplerfish,* 2014.
http://www.knopflerfish.org/

[8] OSGi Alliance. *Semantic Versioning.* Revision 1.0, May 6, 2010.
http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf

[9] Přemysl Brada, Lukáš Valenta. *Practical verification of component substitutability using subtype relation.* In Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications, pages 38-45. IEEE Computer Society Press, 2006.

[10] Jaroslav Bauml, Přemysl Brada. *Reconstruction of Type Information From Java Bytecode for Component Compatibility.* Electronic Notes in Theoretical Computer Science 264(4), 2011.

[11] *Java Class Compatibility Checker,* 2013.
https://www.assembla.com/spaces/jacc/wiki/JavaTypes

[12] *OSGi Version Generator,* 2013.
https://www.assembla.com/wiki/show/obcc/OSGi_Version_Generator

[13] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley. *The Java Language Specification, Java SE 7 Edition,* 28.02.2013.
http://docs.oracle.com/javase/specs/jls/se7/html/index.html

[14] Přemysl Brada, Kamil Ježek. *Ensuring Component Application Consistency on Small Devices: A Repository-Based Approach.* Proceedings of 38th Euromicro Conference on Software Engineering and Advanced Applications. IEEE Computer Society, 2012.

[15] *H2 Database Engine,* 2014
http://www.h2database.com/html/main.html

[16] *Component Repository supporting Compatibility Evaluation,* 2013.
https://www.assembla.com/spaces/crce/wiki/

[17] Oracle Corporation. *Jersey,* 2014
https://jersey.java.net/

[18] Oracle Corporation. *A Relational Database Overview,* 2014.
http://docs.oracle.com/javase/tutorial/jdbc/overview/database.html

[19] Scott W. Ambler. *Relational Databases 101: Looking at the Whole Picture,* 2014.
http://www.agiledata.org/essays/relationalDatabases.html

[20] MongoDB, Inc. *NoSQL Databases Explained,* 2014.
http://www.mongodb.com/learn/nosql

[21] MongoDB, Inc. *Top 5 Considerations When Evaluating NoSQL Databases, A MongoDB WhitePaper,* June 2013.
http://www.mongodb.com/lp/whitepaper/nosql-considerations

[22] MongoDB, Inc. *MongoDB,* 2014.
http://www.mongodb.com/

[23] The Apache Software Foundation. *CouchDB,* 2014.
http://couchdb.apache.org/

[24] MongoDB, Inc. *MongoDB Production Deployments,* 2014.
http://www.mongodb.org/about/production-deployments/

[25] CouchDB Wiki. *CouchDB in the Wild,* 2014.
http://wiki.apache.org/couchdb/CouchDB_in_the_wild

[26] MongoDB, Inc. *MongoDB Java Driver,* 2014.
http://docs.mongodb.org/ecosystem/drivers/java/

[27] MongoDB, Inc. *MongoDB Java Driver Concurrency,* 2014.
http://docs.mongodb.org/ecosystem/drivers/java-concurrency/

[28] Přemysl Brada. *Properties and Verification of Component-Based Systems.* Habilitation Thesis, University of West Bohemia in Pilsen, 2011.

[29] *Clirr,* 2014.
http://clirr.sourceforge.net/
https://github.com/ebourg/clirr

[30] *The Clirr Maven Plugin,* 2014.
http://mojo.codehaus.org/clirr-maven-plugin/index.html

[31] The Eclipse Foundation. *PDE API Tools,* 2014.
https://www.eclipse.org/pde/pde-api-tools/

[32] aQute SARL. *Bnd,* 2014.
http://www.aqute.biz/Bnd/Bnd

[33] The Apache Software Foundation. *Maven Bundle Plugin,* 2014.
http://svn.apache.org/repos/asf/felix/releases/maven-bundle-plugin-2.3.7/doc/site/index.html

[34] Jiří Kučera. *Úložiště komponent podporující kontroly kompatibility.* Diplomová práce, Západočeská univerzita v Plzni, 2011.

[35] Alex Rodriguez. *RESTful Web services: The basics,* 06 November 2008.
https://www.ibm.com/developerworks/webservices/library/ws-restful/

[36] Joshua Thijssen and others. *The RESTful Cookbook* - *When should we use PUT and when should we use POST?*, 2014.
http://restcookbook.com/HTTP%20Methods/put-vs-post/

[37] Oracle Corporation. *JSR-311: JAX-RS: The JavaTM API for RESTful Web Services*
https://jcp.org/en/jsr/detail?id=311

[38] Oracle Corporation. *JSR-339: JAX-RS 2.0: The JavaTM API for RESTful Web Services*
https://jcp.org/en/jsr/detail?id=339

[39] Brada, P., Jezek, K., *Repository and Meta-Data Design for Efficient Component Consistency Verification.* Science of Computer Programming - accepted for publication. Elsevier, 2014

List of Abbreviations

API Application Programming Interface

CBSE Component-Based Software Engineering

CRCE Component Repository supporting Compatibility Evaluation

DAO Data Access Object

EFP Extra-Functional Properties

JACC Java Class Comparator

JAXB Java Architecture for XML Binding

JSON JavaScript Object Notation

OBCC OSGi Bundle Compatibility Checker

OBR OSGi Bundle Repository

OSGi Open Services Gateway initiative

POJO Plain Old Java Object

RDBMS Relational Database Management System

REST Representational State Transfer

XML Extensible Markup Language

XSD XML Schema Document

List of Examples

2.1 Sample of a manifest file. [3] 7

3.1 Statically typed languages problem [9]. 12
3.2 Clirr result reporting an added class. 20

5.1 Example of version and compatibility inconsistency. 31
5.2 Java driver for MongoDB [26]. 38

7.1 Resource structure compliant with REST principles. 48
7.2 Example requests for scenario BOSB. 49
7.3 Example requests for scenarios BOSM/AM. 50
7.4 Example request for scenarios BOOM. 50
7.5 Example request for scenario SMFP. 51
7.6 Example request for scenario SMRS. 52
7.7 Example request for scenario BOUS. 52
7.8 Example of core metadata identity. 53
7.9 Example of core metadata capabilities. 54
7.10 Example of core metadata requirements. 54
7.11 Example of difference metadata. 55

List of Figures

2.1 The component-based design pattern [2]. 3
2.2 OSGi Model [1]. 5
2.3 OSGi Bundle Lifecycle [4]. 8

3.1 OBCC Workflow. 16
3.2 Example Comparison Result. 18

4.1 CRCE Architecture [16]. 22
4.2 Example of CRCE capability hierarchy [39]. 25
4.3 CRCE Repository browser view. 26

5.1 Process of creating a `Compatibility` instance. 41

6.1 CRCE Version compatibility search view. 46

8.1 Computation Times Based on Number of Previous Revisions - four threads. 61
8.2 Computation Times Based on Number of Previous Revisions - single thread. 62

List of Tables

3.1 Combination of difference values [9]. 13
3.2 Mapping of difference values to compatibility [10]. 14
3.3 Derivation of new version identifier. 15

5.1 Logical View Difference Meta-Data Repository 30

8.1 List of components used for performance tests. 59
8.2 Time to Compute Difference Meta-Data for a Single Pair of Components. . 60
8.3 Comparison of Computation Times - 1 vs 4 threads. 60
8.4 Scenario duration - comparison of single thread vs four threads. 62

C.1 Computation Times Based on Number of Previous Revisions - four threads. 78
C.2 Computation Times Based on Number of Previous Revisions - single thread. 78

Appendices

A Meta-Data XML Representation

```
<repository name="store" increment="0">
    <resource id="...">
        <capability namespace="osgi.wiring.bundle" id="...">

            <attribute name="manifest-version" value="2"/>
            <attribute name="symbolic-name" value="obcc-parking-example.gate"/>
            <attribute name="version" value="1.1.0" type="Version"/>
            <attribute name="presentation-name" value="obcc-parking-example.gate"/>
            <capability namespace="osgi.wiring.package" id="...">

                <attribute name="name" value="cz.zcu.kiv.osgi.demo.parking.gate.statistics"/>
                <attribute name="version" value="1.1.0" type="Version"/>
                <directive name="uses"
                 value="cz.zcu.kiv.osgi.demo.parking.carpark.status,
                 org.slf4j,cz.zcu.kiv.osgi.demo.parking.statsbase"/>
            </capability>
            <capability namespace="osgi.wiring.package" id="...">

                <attribute name="name" value="cz.zcu.kiv.osgi.demo.parking.gate.vehiclesink"/>
                <attribute name="version" value="0.0.0" type="Version"/>
                <directive name="uses"
                 value="cz.zcu.kiv.osgi.demo.parking.carpark.flow,
                 cz.zcu.kiv.osgi.demo.parking.gate.statistics,org.slf4j"/>
            </capability>
            <capability namespace="osgi.wiring.package" id="...">

                <attribute name="name" value="cz.zcu.kiv.osgi.demo.parking.lane.statistics"/>
                <attribute name="version" value="1.1.0" type="Version"/>
                <directive name="uses" value="cz.zcu.kiv.osgi.demo.parking.statsbase,org.slf4j"/>
            </capability>
        </capability>
        <capability namespace="osgi.identity" id="...">

            <attribute name="name" value="obcc-parking-example.gate-1.1.0"/>
            <attribute name="symbolic-name" value="obcc-parking-example.gate"/>
            <attribute name="version" value="1.1.0" type="Version"/>
            <attribute name="presentation-name" value="obcc-parking-example.gate"/>
        </capability>
        <capability namespace="crce.identity" id="d...">

            <attribute name="repository-id" value="1"/>
            <attribute name="file-name" value="obcc-parking-example.gate.jar"/>
            <attribute name="hash" value="..."/>
            <attribute name="name" value="obcc-parking-example.gate"/>
```

```
            <attribute name="categories" value="zip,osgi,versioned,metrics" type="List"/>
            <attribute name="mime" value="application/vnd.osgi.bundle"/>
            <attribute name="uri" value="file:/..." type="URI"/>
            <attribute name="size" value="19892" type="Long"/>
            <attribute name="url"
             value="http://.../rest/bundle/obcc-parking-example.gate-1.1.0"/>
        </capability>
        <requirement namespace="osgi.wiring.package" id="...">

            <attribute name="name" value="cz.zcu.kiv.osgi.demo.parking.carpark.flow"/>
            <directive name="text"
             value="Import package cz.zcu.kiv.osgi.demo.parking.carpark.flow"/>
        </requirement>
        <requirement namespace="osgi.wiring.package" id="...">

            <attribute name="name" value="cz.zcu.kiv.osgi.demo.parking.carpark.status"/>
            <directive name="text"
             value="Import package cz.zcu.kiv.osgi.demo.parking.carpark.status"/>
        </requirement>
        <requirement namespace="osgi.wiring.package" id="...">

            <attribute name="name" value="cz.zcu.kiv.osgi.demo.parking.gate.statistics"/>
            <directive name="text"
             value="Import package cz.zcu.kiv.osgi.demo.parking.gate.statistics"/>
        </requirement>
        <property namespace=’’crce.compatibility’’>

            <compatibility base-version="1.0.1" contract="syntax" value="INS">


              <diff level="PACKAGE" name="cz.zcu.kiv.osgi.demo.parking.gate.statistics"
                    namespace="osgi.wiring.package" role="CAPABILITY" syntax="java" value="INS">


                <diff level="TYPE" name="cz.zcu.kiv.osgi.demo.parking.gate.statistics.GateStatistics"
                       value="INS">
                    <diff level="OPERATION" name="vehiclesArrived" value="INS"/>
                    <diff level="OPERATION" name="vehiclesDeparted" value="INS"/>
                </diff>
              </diff>
              <diff level="PACKAGE" name="cz.zcu.kiv.osgi.demo.parking.gate.vehiclesink"
                    namespace="osgi.wiring.package" role="CAPABILITY" syntax="java" value="INS"/>
              <diff level="PACKAGE" name="cz.zcu.kiv.osgi.demo.parking.lane.statistics"
                    namespace="osgi.wiring.package" role="CAPABILITY" syntax="java" value="INS">


                <diff level="TYPE" name="cz.zcu.kiv.osgi.demo.parking.lane.statistics.LaneStatistics"
                      value="INS">
                    <diff level="OPERATION" name="getVehiclesPerInterval" value="INS"/>
                </diff>
              </diff>
            </compatibility>
        </property>
    </resource>
</repository>
```

B Compatibility Meta-Data JSON Representation

Example of a `Compatibility` instance. `Diff` instances can be found under the `details` key.


```
{
"_id" : ObjectId("53247659a444514db63cd3f9"),
"resourceName" : "obcc-parking-example.gate",
"resourceVersion" :
    {
    "major" : 1,
    "minor" : 1,
    "micro" : 0,
    "qualifier" : ""
    },
"baseName" : "obcc-parking-example.gate",
"baseVersion" :
    {
    "major" : 1,
    "minor" : 0,
    "micro" : 0,
    "qualifier" : ""
    },
"bundleDifference" : "INS",
"details" :
[
    {
    "name" : "cz.zcu.kiv.osgi.demo.parking.gate.statistics",
    "level" : "PACKAGE",
    "value" : "INS",
    "namespace" : "osgi.wiring.package",
    "role" : "CAPABILITY",
    "children" :
    [
        {
        "name" : "cz.zcu.kiv.osgi.demo.parking.gate.statistics.GateStatistics",
        "level" : "TYPE",
```

```
    "value" : "INS",
    "namespace" : null,
    "children" :
    [
        {
        "name" : "vehiclesArrived",
        "level" : "OPERATION",
        "value" : "INS",
        "namespace" : null,
        "children" : [ ]
        },
        {
        "name" : "vehiclesDeparted",
        "level" : "OPERATION",
        "value" : "INS",
        "namespace" : null,
        "children" : [ ]
        }
    ]
    }
]
},
{
"name" : "cz.zcu.kiv.osgi.demo.parking.gate.vehiclesink",
"level" : "PACKAGE",
"value" : "INS",
"namespace" : "osgi.wiring.package",
"role" : "CAPABILITY",
"children" : [ ]
},
{
"name" : "cz.zcu.kiv.osgi.demo.parking.lane.statistics",
"level" : "PACKAGE",
"value" : "INS",
"namespace" : "osgi.wiring.package",
"role" : "CAPABILITY",
"children" :
[
    {
    "name" : "cz.zcu.kiv.osgi.demo.parking.lane.statistics.LaneStatistics",
    "level" : "TYPE",
    "value" : "INS",
    "namespace" : null,
    "children" :
    [
        {
```

```
                "name" : "getVehiclesPerInterval",
                "level" : "OPERATION",
                "value" : "INS",
                "namespace" : null,
                "children" : [ ]
                },
                {
                "name" : "getVehiclesPerInterval",
                "level" : "OPERATION",
                "value" : "INS",
                "namespace" : null,
                "children" : [ ]
                }
            ]
            }
        ]
        }
    ]
}
```

C Performance Test Results

Values in the tables are in seconds.

Component	Number of Previous Revisions					
	1	10	20	40	80	140
config-types	0.0088	0.2264	0.3388	0.1562	0.2442	0.5270
osgi-adapter	0.0421	0.7564	2.17	1.0936	4.3712	3.6384
dataprovider	0.0762	0.5306	1.4684	2.7736	5.9630	-
asm-all-repackaged	0.1482	1.0482	3.6638	9.9158	12.2528	17.5252
bean-validator	0.3732	3.1326	6.3248	15.5300	21.3684	46.6588
webservices-osgi	33.1132	358.0122	723.0542	1324.5786	-	-

Table C.1: Computation Times Based on Number of Previous Revisions - four threads.

Component	Number of Previous Revisions					
	1	10	20	40	80	140
osgi-adapter	0.0028	0.2818	0.7556	1.6512	2.1546	3.5936
asm-all-repackaged	0.1031	0.9618	2.2762	4.2386	7.8308	14.2044

Table C.2: Computation Times Based on Number of Previous Revisions - single thread.