

FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
INFORMATICS INSTITUTE
BACHELOR OF COMPUTER SCIENCE

RAFAEL MAURICIO PESTANO

Towards a Software Metric for OSGi

Graduation Thesis

Advisor: Prof. Dr. Cláudio Fernando Resin
Geyer

Coadvisor: Prof. Dr. Didier DONSEZ

Porto Alegre
December 2014

FEDERAL UNIVERSITY OF RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco

Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb

Coordenador do Curso de CIC: Prof. Raul Fernando Weber

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

*“If I have seen farther than others,
it is because I stood on the shoulders of giants.”*

— SIR ISAAC NEWTON

ACKNOWLEDGMENTS

Acknowledgments

CONTENTS

ABSTRACT	7
RESUMO	8
LIST OF FIGURES	9
LIST OF TABLES	10
LIST OF ABBREVIATIONS AND ACRONYMS	11
1 INTRODUCTION	12
1.1 Context	12
1.2 Objectives	13
1.3 Organization	13
2 BASIC CONCEPTS	14
2.1 Software Quality	14
2.1.1 External Quality	14
2.1.2 Internal Quality	14
2.1.3 Quality Measurement	15
2.1.4 Software Metrics	16
2.1.5 Program Analysis	17
2.1.6 Quality Analysis Tools	18
2.2 Java and OSGi	19
2.2.1 The Java language	20
2.2.2 The OSGi service platform	21
2.2.3 Vanilla Java vs OSGi	24
2.3 JBoss Forge	26
2.3.1 Introduction	26
2.3.2 Forge Plugin	26
2.3.3 Facets	27
2.3.4 Project Locator	28
2.3.5 Applications	28
3 INTRABUNDLE - AN OSGI BUNDLE INTROSPECTION TOOL	29
3.1 Introduction	29
3.2 Design Decisions	29

3.3	Implementation Overview	31
3.4	Identifying OSGi Projects and Bundles	32
3.5	Collecting Bundle Data	32
3.6	Quality Calculation	34
3.6.1	Quality labels	34
3.6.2	Metrics Created	35
3.6.3	Quality Formula	36
3.7	Intrabundle Reports	38
3.8	Intrabundle Quality	40
3.8.1	Internal quality	41
3.8.2	External quality	41
4	BUNDLE INTROSPECTION RESULTS	43
4.1	Analyzed Projects	43
4.2	Projects Quality Results	44
4.2.1	General quality comparison	44
4.2.2	Metric quality comparison	45
5	CONCLUSION	47
	REFERENCES	48
	APPENDICES	51
A	ANALYZING BUNDLE USAGE EXAMPLE	52
A.1	Setup environment	52
A.2	Begin Introspection	52

ABSTRACT

Today's software applications are becoming more complex, bigger, dynamic and harder to maintain. One way to overcome modern systems complexities is to build modular applications so we can divide it into small blocks which collaborate to solve bigger problems, the so called *divide to conquer*. Another important aspect in the software industry that helps building large applications is the concept of software quality because it's well known that higher quality softwares are easier to maintain and evolve at long term.

The Open Services Gateway Initiative(OSGi) is a very popular solution for building Java modular applications. It is very hard to measure the quality of OSGi systems due to its particular characteristics like service oriented, intrinsic modularity and component based approach.

In this work will be presented a tool called *Intrabundle* that analyses OSGi projects and measure their internal quality. The tool extracts useful information that is specific to this kind of project and organize the analyzed data into Human readable reports in various formats.

Yet it's also proposed 6 metrics based on good practices inside OSGi world which are applied to 10 real OSGi projects that vary in size, teams and domain.

Keywords: OSGi. java. quality. metrics. modularity. intrabundle.

RESUMO

As aplicações de software hoje em dia estão cada vez mais complexas, maiores, dinâmicas e mais difíceis de manter. Uma maneira de superar as complexidades dos sistemas modernos é através de aplicações modulares as quais são divididas em partes menores que colaboram entre si para resolver problemas maiores, o famoso *dividir para conquistar*. Outro aspecto importante na indústria de software que ajuda a construir aplicações grandes é o conceito de qualidade de software já que é sabido que, quanto maior a qualidade do software, mais fácil de mantê-lo e evolui-lo a longo prazo será.

The Open Services Gateway Initiative(OSGi) é uma solução bastante popular para se criar aplicações modulares em Java porém é muito difícil medir a qualidade interna de sistemas OSGi devido a suas características particulares como arquitetura orientada a serviços e componentes assim como modularidade intrínseca.

Neste trabalho será apresentada uma ferramenta chamada *Intrabundle* que analisa projetos OSGi e mede sua qualidade interna. A ferramenta extrai informações úteis que são específicas desse tipo de projeto e organiza os dados extraídos em relatórios em diversos formatos.

Ainda foram propostas métricas de qualidade baseadas em boas práticas conhecidas do mundo OSGi que serão aplicadas em 10 projetos reais que variam em tamanho, equipes e domínio.

Palavras-chave: OSGi. java. quality. metrics. modularity. intrabundle.

LIST OF FIGURES

2.1	Internal and external quality audience	15
2.2	Intrabundle PMD rule violation	19
2.3	Intrabunde PMD ruleset	19
2.4	JVM architecture	20
2.5	OSGi architecture	21
2.6	Module Layer	23
2.7	OSGi bundle Lifecycle	23
2.8	Lifecycle Layer	24
2.9	Service Layer	24
2.10	Java jar hell	25
2.11	Bundle classpath	25
2.12	Forge initial screen	26
3.1	Intrabundle Architecture	31
3.2	Intrabundle general report	39
3.3	Intrabundle general report - detailed section	39
3.4	Intrabundle metrics report	40
3.5	Intrabundle metrics report - detailed section	40
3.6	Intrabundle code coverage	41
3.7	Intrabundle integration tests	42

LIST OF TABLES

2.1	Quality characteristics to be considered	16
2.2	Common Software metrics	17
2.3	Quality analysis tools	18
3.1	Supported types of OSGi projects	32
3.2	Extracted data from OSGi projects	33
4.1	OSGi projects analyzed by Intrabundle	43
4.2	Projects general quality	45
4.3	Projects quality by metrics	45

LIST OF ABBREVIATIONS AND ACRONYMS

CISQ Consortium for IT Software Quality

JVM Java Virtual Machine

IEC International Electrotechnical Commission

ISO International Organization for Standardization

API Application Programming Interface

IDE Integrated Development Environment

GUI Graphic User Interface

LOC Lines of Code

KLOC Kilo Lines of Code

1 INTRODUCTION

This chapter will drive the reader through the context and motivation of this work followed by the objectives and later the organization of this text is presented.

1.1 Context

One of the pillars of sustainable software development is its quality which can basically be defined as internal and external. External quality focuses on how software meets its specification and works accordingly to its requirements. Internal quality is aimed on how well the software is structured and designed. To measure external quality there is the need to execute the software¹ either by an end user accessing the system or an automated process like for example functional testing or performance testing. Internal quality however can be verified by either *static analysis*, that is mainly the inspection of the source code itself, or by *dynamic analysis* which means executing the software like for example automated *whitebox testing*².

With good software quality in mind we take applications to another level where maintainability is increased, correctness is enhanced, defects are identified in early development stages, which can lead up to 100 times reduced costs (BEOHM et al., 2001).

A well known and successful way to structure software architecture is to modularize its components allowing easier evolution of the system because smaller decoupled modules are typically easier to maintain than classical applications. In the Java ecosystem there is a moving to modularize the JDK and Java applications with the project Jigsaw (KRILL, P.) and also a recent interest in *microservices* (KNORR, E.) arise. Although all this interest in modular application today, the only practical working and well known solution for modular Java applications is OSGi (HALL et al., 2011), a very popular component-based and service-oriented framework for building Java modular applications. OSGi is the *de facto* standard solution for this kind of software since early 2000's and have being used as basis of most JavaEE³ application servers⁴, the open source IDE Eclipse(ECLIPSE, 2006), Atlassian Jira and Confluence to cite a few big players using OSGi.

In the context of software quality and Java modular applications using OSGi there is no known standard way neither well known tools to measure OSGi projects *internal quality* (Hamza et al., 2013) (WANG et al., 2012). Some work have been done by (Gama and Donsez, 2012) and (WANG et al., 2012), both focus on OSGi services reliability and general project quality is not their objective. For *external quality* the classical approaches like automated testing are

¹Also known as dynamic analysis

²whitebox testing is the detailed investigation of internal logic and structure of the code (KHAN et al., 2012)

³A Java platform dedicated for enterprise applications which are usually secure and robust systems that display, manipulate and store large amounts of complex data maintained by an organization

⁴Java application servers are like an extended virtual machine for running applications, transparently handling connections to the database, connections to the Web client, managing components like Enterprise Java Beans(EJB) and so on

sufficient because this kind of quality aims in the *behavior* and not the *design* so technology and architecture is usually not taken into account.

1.2 Objectives

The main objective of this work is to create a tool to extract software metrics and measure internal quality of OSGi projects where these metrics must reflect good practices in the OSGi world. The main difference the proposed metrics have compared to classical software metrics is that the first will be based on modularity attributes that only exists in modular applications. The tool applies and validate the metrics on real OSGi projects and finally the resulting qualities are analyzed.

1.3 Organization

This text is organized in the following way. First chapter defines the context, motivation and objectives of this work. The second chapter introduces the main concepts and technologies used in this work and is divided into two main sections where the first is focused in the area of software quality like quality measurement, quality metrics, program analysis and quality analysis tools. The second section of chapter two presents Java and OSGi, how standard Java and OSGi are different in respect to quality metrics and why we need different metrics for OSGi. The third chapter presents **Intrabundle**, an OSGi code introspection tool to measure internal quality, it shows how Intrabundle works, what kind of information it extracts and what metrics it is applying. The fourth chapter analyzes the results Intrabundle produces and validates them to decide if this work has a valid contribution or not. The last chapter presents the conclusions and future work on this subject.

2 BASIC CONCEPTS

This chapter presents an overview of the concepts and technologies that were studied and used on the development of this work. In section 2.1 - *Software Quality*, will be presented general aspects of software quality such as *quality measurement*, *software metrics*, *program analysis* and some tools that are used in this area.

Section 2.2 - *Java and OSGi* will introduce OSGi a framework for build service oriented Java modular applications. Finally section 2.3 will introduce JBoss Forge, a Java framework used as runtime¹ for Intrabundle².

2.1 Software Quality

There has been many definitions of software quality (KAN, 2002, p. 23) and there is even an ISO norm for it, the ISO/IEC 25010 (ISO25010, 2011). All this definitions agree that the main motivation to perform continuous software quality management is to avoid **software failures** and increase **maintainability** in the sense that the more quality a program has the easier will be to maintain, the less bugs or abnormal behavior it will have and the more it will conform with its functional and non functional requirements³.

Software quality can be divided in two groups, the **external** and **internal** quality.

2.1.1 External Quality

When we talk about *external quality* we are aiming to the user view which is the one that sees the software working and use it. This kind of quality is usually enforced through software testing. External quality can also be mapped to functional requirements so the greater external quality is the more usable and less defects it will have for example.

2.1.2 Internal Quality

The opposite is internal or structural quality that aims to how the software is architect-ed internally which is the perspective of the programmer and non functional requirements. So the higher internal quality the better the code is structured, efficient, robust and maintainable it should be. Image 2.1 illustrates internal and external quality and its target audience.

¹Is software designed to support the execution of computer programs written in some computer language

²A Java based project that will be presented later on this work

³Functional and non functional requirements can be simply defined as *what* the software does and *how* the software will do respectively

Figure 2.1: Internal and external quality audience



2.1.3 Quality Measurement

Quality measurement focuses on quantifying software desirable characteristics and each characteristic can have a set of measurable attributes, for example *high cohesion* is a desirable characteristic and *LOC - lines of code* is a measurable attribute related to cohesion. Quality measurement is close related to internal quality and in most cases is performed via static code analysis where program code is inspected to search for quality attributes to be measured but in some cases a dynamic analysis, where the program analysis is done during software execution, can be performed to measure characteristics that can be perceived only when software is running, for example performance or code coverage⁴.

In the extent of this work the characteristics of software to be considered and measured later are listed and described in table 2.1:

⁴A technique that measures the code lines that are executed for a given set of software tests, its also considered a software metric.

Table 2.1: Quality characteristics to be considered

Characteristic	Description	OSGi example
Reliability	the degree to which a system or component performs its required functions under stated conditions for a specified period of time.	Bundles should not have stale service references.
Performance Efficiency	Performance relative to the amount of resources used under stated conditions for a specified period of time.	Bundle startup time, also bundle dependency can decrease performance.
Security	the degree of protection of information and data so that unauthorized persons or systems cannot read, access or modify them.	Bundles should declare permissions
Maintainability	The degree to which the product can be modified.	Modules should be loosely coupled, bundles should publish only interfaces etc.

Source: CISQ (2013)

2.1.4 Software Metrics

A software metric is the measurement of a software attribute which in turn is a quantitative calculation of a characteristic. Software metrics can be classified into three categories: product metrics⁵, process metrics⁶, and project metrics⁷. Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project (KAN, 2002).

2.1.4.1 Good Software Metrics

Good metrics may have the following aspects:

- *Linear*: metric values should follow an intuitive way to compare its values like for example higher values should correspond to better quality whereas lower values to worse quality and vice versa.
- *Independent*: two metric values should not interfere on each other.
- *Repeatable*: this is a very important aspect in continuous quality management where software is changing all the time and we want to measure quality on every change.
- *Accurate*: the metric should be meaningful and should help answer how good a software

⁵Product metrics describe the characteristics of the product such as size, complexity, design features, performance

⁶Process metrics can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development and response time of bug fixing

⁷Project metrics describe the project characteristics and execution. Examples include the number of software developers, cost, schedule, and productivity

attribute is, for example using latency⁸ to calculate response time⁹ in a web application isn't accurate.

2.1.4.2 Common Software Metrics

The table 2.2 below shows some well known software metrics and its description:

Table 2.2: Common Software metrics

Metric	Description
Cyclomatic complexity	It is a quantitative measure of the complexity of programming instructions.
Cohesion	measure the dependency between units of code like for example classes in object oriented programming or modules in modular programming like OSGi.
Coupling	measures how well two software components are data related or how dependent they are.
Lines of code (LOC)	used to measure the size of a computer program by counting the number of lines in the text of the program's source code.
Code coverage	measures the code lines that are executed for a given set of software tests
Function point analysis (FPA)	used to measure the size (functions) of software.

Source: SQA (2012)

2.1.5 Program Analysis

Program analysis is the process of automatically analyzing the behavior of computer programs. Two main approaches in program analysis are **static program analysis** and **dynamic program analysis**. Main applications of program analysis are program correctness, program optimization and quality measurement.

2.1.5.1 Static Program Analysis

Is the analysis of computer software that is performed without actually executing programs (Wichmann et al., 1995). In this kind of analysis source code is inspected and valuable information is collected based on its internal structure and components.

⁸The delay incurred in communicating a message, the time the message spends "on the wire"

⁹The total time it takes from when a user makes a request until they receive a response

2.1.5.2 Dynamic Program Analysis

Is a technique that analyze the system's behavior on the fly, while it is executing. The main objectives of this kind of analyze is to catch *memory leaks*¹⁰, identify arithmetic errors and extract code coverage and measure performance.

2.1.6 Quality Analysis Tools

The table 2.3 lists some code quality analysis tools in the Java ecosystem:

Table 2.3: Quality analysis tools

Name	Description	Type
SonarQube	An open source platform for continuous inspection of code quality.	static
FindBugs	An open-source static bytecode analyzer for Java.	static
Checkstyle	A static code analysis tool used in software development for checking if Java source code complies with coding rules.	static
PMD	A static ruleset based Java source code analyzer that identifies potential problems.	static
ThreadSafe	A static analysis tool for Java focused on finding concurrency bugs.	static
InFusion	Full control of architecture and design quality.	static
JProfiler	helps you resolve performance bottlenecks, pin down memory leaks and understand threading issues	dynamic
JaCoCo	A free code coverage library for Java.	dynamic
Javamelody	Java or Java EE application Monitoring in QA and production environments.	dynamic
Introscope	An application management solution that helps enterprises keep their mission-critical applications high-performing and available 24x7.	dynamic

Figure 2.2 shows the execution of static analysis on *Intrabundle* using *PMD*, note that it is based on rules and *Intrabundle* break some of them(intentionally) like *Unused variables*, *EmptyCatchBlock* so *PMD* consider them compile failure and the project cannot be compiled until the rules are fixed in code:

¹⁰Resources that are hold on system's memory and aren't released

Figure 2.2: Intrabundle PMD rule violation

```
[INFO] >>> maven-pmd-plugin:3.2:check (default) @ intrabundle >>>
[INFO]
[INFO] --- maven-pmd-plugin:3.2:pmd (pmd) @ intrabundle ---
[INFO]
[INFO] <<< maven-pmd-plugin:3.2:check (default) @ intrabundle <<<
[INFO]
[INFO] --- maven-pmd-plugin:3.2:check (default) @ intrabundle ---
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.metric.DefaultMetricsCalculator:119 Rule:EmptyCatchBlock Priority:2 Must handle exceptions.
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.model.ManifestMetadata:143 Rule:StringInstantiation Priority:2 Avoid instantiating String objects; this is usually unnecessary..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.model.ManifestMetadata:160 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf(String)..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.model.ManifestMetadata:162 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf(String)..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.model.ManifestMetadata:201 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf(String)..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.model.ManifestMetadata:255 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf(String)..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.model.ManifestMetadata:309 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf(String)..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.model.OSGiModuleImpl:186 Rule:EmptyCatchBlock Priority:2 Must handle exceptions.
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.model.OSGiProjectImpl:37 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'projectMetric'..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.model.OSGiProjectImpl:38 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'metrics'..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.plugin.BundlePlugin:22 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'prompt'..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.plugin.LocalePlugin:34 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'event'..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.plugin.OSGiScanPlugin:39 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'moduleLevel's'..
[INFO] PMD Failure: br.ufers.rmpestano.intrabundle.util.ProjectUtils:328 Rule:UnusedLocalVariable Priority:3 Avoid unused local variables such as 'line'..
[INFO] -----
[INFO] BUILD FAILURE
```

The rules are totally customizable via xml configuration, Intrabundle PMD rules are shown in Figure 2.3:

Figure 2.3: Intrabundle PMD ruleset

```
1 <?xml version="1.0"?>
2 <ruleset name="Custom ruleset" xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0">
5   <description>
6     This ruleset checks my code for bad stuff
7   </description>
8
9   <exclude-pattern>./src/test/.*</exclude-pattern>
10
11   <!-- Here's some rules we'll specify one at a time -->
12   <rule ref="rulesets/java/unusedcode.xml/UnusedLocalVariable" />
13   <rule ref="rulesets/java/unusedcode.xml/UnusedPrivateField" />
14   <rule ref="rulesets/java/imports.xml/DuplicateImports" />
15   <rule ref="rulesets/java/basic.xml/UnnecessaryConversionTemporary" />
16
17   <rule ref="rulesets/java/strings.xml">
18     <exclude name="AvoidDuplicateLiterals" />
19     <exclude name="AppendCharacterWithChar" />
20     <exclude name="ConsecutiveLiteralAppends" />
21     <exclude name="InefficientStringBuffering" />
22   </rule>
23
24   <!-- We want to customize this rule a bit, change the message and raise
25     the priority -->
26   <rule ref="rulesets/java/basic.xml/EmptyCatchBlock" message="Must handle exceptions">
27     <priority>2</priority>
28   </rule>
29
30   <!-- Now we'll customize a rule's property value -->
31   <rule ref="rulesets/java/codesize.xml/CyclomaticComplexity">
```

Source: intrabundle ruleset (2014)

2.2 Java and OSGi

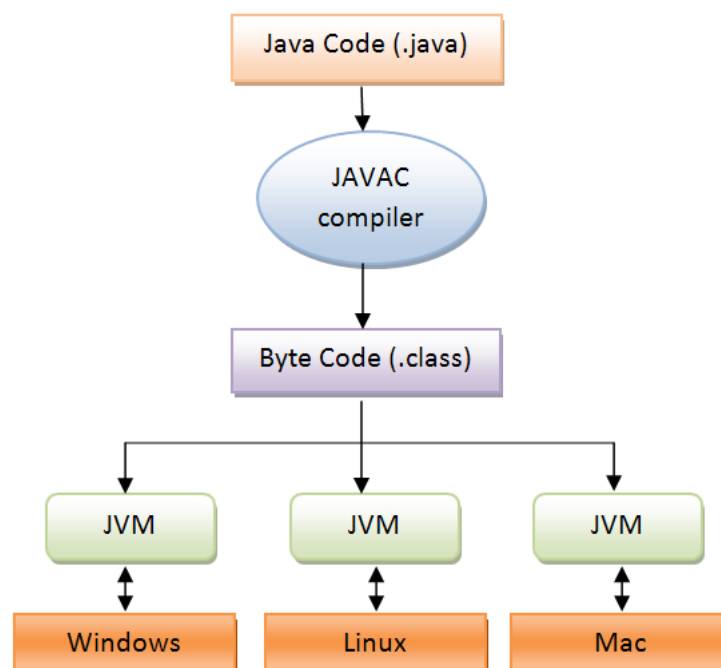
In the context of JavaTM programming language (Arnold et al., 2005), which accordingly to IEEE spectrum of this year is the most popular programming language (IEEE Spectrum, 2014),

and modular applications¹¹ this section will introduce the Java language and OSGi framework.

2.2.1 The Java language

Java is a general purpose object oriented¹² programming language created by Sun Microsystems in 1995 which aims on simplicity, readability and universality. Java runs on top of the so called JVM, the acronym for Java Virtual Machine, which is an abstract computing machine¹³ and platform-independent execution environment that execute Java byte code¹⁴. The JVM converts java byte code into host machine language(e.g. linux, windows etc...) allowing Java programs to "run everywhere" independently of operating system or platform. JVM implementations are different for each platform but the generated bytecode is the same, Figure 2.4 illustrates how JVM works:

Figure 2.4: JVM architecture



Other aspects of Java are listed below:

- Type safe¹⁵
- Dynamic: during the execution of a program, Java can dynamically load classes
- Strong memory management(no explicit pointer)

¹¹A software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules which represent a separation of concerns and improves maintainability

¹²Object-oriented programming(OOP) integrates code and data using the concept of an "object" which is a piece of software that holds state and behavior

¹³Also known as *Virtual Machine* which is an emulation of a particular computer system

¹⁴The intermediate output of the compilation of a program written in Java that can be read by the JVM

¹⁵Type safety is the extent to which a programming language discourages or prevents type errors

- Automatic garbage collection to release unused objects from memory
- Robust: extensive compile-time checking so bugs can be found early
- Multithreaded¹⁶
- Distributed: networking capability is inherently integrated into Java

2.2.2 The OSGi service platform

OSGi is a component based service oriented platform specification maintained by *OSGi Alliance*¹⁷ that runs on top of Java. As of November 2014 the specification is at version 6 and currently has four implementations¹⁸. It is composed by *OSGi framework* and *OSGi standard services*. The framework is the runtime that provides the basis of all OSGi module system functionalities like modules management for example. Standard services define some reusable apis and extension points to easy development of OSGi based applications. Figure 2.5 illustrates OSGi platform architecture:

Figure 2.5: OSGi architecture



2.2.2.1 Bundles

Bundles are the building blocks of OSGi applications. A bundle¹⁹ is a group of Java classes and resources packed as .jar extension with additional metadata in manifest MANIFEST.MF file

¹⁶Multithreading is a program's capability to perform several tasks simultaneously

¹⁷A non profit worldwide consortium of technology innovators

¹⁸[Apache Felix](#), [Eclipse Equinox](#), [Knopflerfish](#) and [ProSyst](#)

¹⁹Also known as module

describing its module boundaries like for example the packages it imports and exports. Below is an OSGi manifest file example:

```
Bundle-Name: Hello World
Bundle-SymbolicName: org.wikipedia.helloworld
Bundle-Description: A Hello World bundle
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Activator: org.wikipedia.Activator
Export-Package: org.wikipedia.helloworld;version="1.0.0"
Import-Package: org.acme.api;version="1.1.0"
```

Looking at manifest OSGi can ensure its most important aspect, *modularity*, so for example our **Hello World** bundle will only be started (later we will explore bundle lifecycle) if and only if there is a bundle (in resolved or installed state) that exports *org.acme.api* package, this is called **explicit boundaries**.

With OSGi, you modularize applications into bundles. Each bundle is a tightly coupled, dynamically loadable collection of classes packed in JARs²⁰, and configuration files that explicitly declare any external dependencies. All these characteristics are provided in OSGi by three conceptual layers that will be briefly presented here, *Module*, *Lifecycle* and *Service*.

2.2.2.2 *Module layer*

This layer is the basis for others as modularization is the key concept of OSGi. The module layer defines OSGi module concept - bundle, which is a JAR file with extra metadata. It also handles the packaging and sharing of Java packages between bundles and the hiding of packages from other bundles. The OSGi framework dynamically resolves dependencies among bundles and performs bundle resolution to match imported and exported packages. This layer ensures that class loading happens in a consistent and predictable way.

²⁰acronym for Java Archive, a file that used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute

Figure 2.6: Module Layer



Source: OSGi conceptual layers (2011)

2.2.2.3 Lifecycle layer

Provides access to the underlying OSGi framework through the *Bundle Context* object. This layer handles the lifecycle of individual bundles so you can manage your application dynamically, including starting and stopping bundles to manage and evolve them over time. Bundles can be dynamically installed, started, updated, stopped and uninstalled. Figure 2.7 shows bundle lifecycle and its possible states where transitions are performed by OSGi commands like *start* or *stop* for example and states are represented in squares:

Figure 2.7: OSGi bundle Lifecycle



If OSGi were a car, module layer would provide modules such as tire, seat, etc, and the

lifecycle layer would provide electrical wiring which makes the car run.

Figure 2.8: Lifecycle Layer



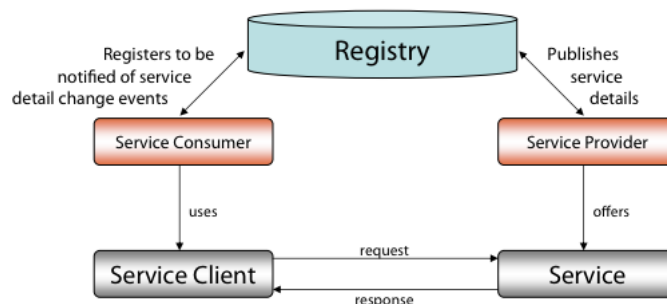
Source: OSGi conceptual layers (2011)

2.2.2.4 Service layer

This layer provides communication among modules and their contained components. Service providers publish services²¹ to *service registry*, while service clients search the registry to find available services to use. The registry is accessible to all bundles so they can *publish* its services as well *consume* services from other bundles.

This is like a service-oriented architecture (SOA) which has been largely used in web services. Here OSGi services are local to a single VM, so it is sometimes called SOA in a VM.

Figure 2.9: Service Layer



2.2.3 Vanilla Java vs OSGi

The main motivation behind OSGi and advantage over standard Java application, as illustrated before, is the modularity. The main issue with Java default runtime is the way Java classes

²¹ A Service is an operation offered as an interface that stands alone in the model, without encapsulating state (Evans and Fowler, 2003)

are loaded, it is the root cause that inhibits modularity in classical Java applications. In standard Java, user classes²² are loaded by a classloader²³ from the same classpath²⁴ which is commonly referred as a *flat classpath*. A flat classpath is the main cause of a well known problem in Java applications, the *Jar Hell*²⁵. Figure 2.10 is an example of Jar hell where multiple JARs containing overlapping classes(consider each shape as being a Java class) and/or packages are merged based on their order of appearance in the class path.

Figure 2.10: Java jar hell



Source: (HALL et al., 2011, p. 7)

In the OSGi environment instead of a *flat classpath* each bundle has its classloader and its classpath. See Figure 2.11 where Bundle A's classpath is defined as the union of its bundle classpath with its imported packages, which are provided by bundle B's exports.

Figure 2.11: Bundle classpath



Source: (HALL et al., 2011, p. 59)

In OSGi runtime we can say we have a graph of classpaths that allows powerful versioning mechanisms so for example we can have multiple versions of the same class or resource loaded

²²Classes that are defined by developers and third parties and that do not take advantage of the extension mechanism

²³A class loader is an object that is responsible for loading classes

²⁴classpath tells Java virtual machine where to look in the filesystem for files defining these classes

²⁵A term used to describe all the various ways in which the classloading process can end up not working

Listing 2.1: Forge plugin example

```

@Alias("hello-world")
public class HelloWorldPlugin implements Plugin {

    @Command(value = "sayHello")
    public void countBundles(PipeOut out) {
        out.println("Hello World" );
    }

}

```

Plugin is just a marker interface so Forge can identify plugins. To fire the sayHello command one have to start forge, install the HelloPlugin and then can use the command by typing *hello-world sayHello* in Forge console and so "Hello World" should be printed in console.

2.3.3 Facets

A Facet in the Forge environment is responsible for restricting the usage of a plugin. It is in fact an interface²⁹ with a method with return type boolean that must decide if the facet is installed.

2.3.3.1 Example

Below is an example of facet that restricts the usage of hello-world plugin, in the example the command should be only available when user is in a directory named *hello* otherwise Forge will claim that the command does not exist in current context.

Listing 2.2: Forge facet example

```

public class HelloFacet implements Facet {

    @Inject
    Project project;

    @Override
    public boolean isInstalled() {
        return project.getProjectRoot().getName().equals("hello");
    }

}

```

²⁹In object oriented programming is a contract that defines which methods the implementors of the interface must provide

So the idea of a facet is that it is active when `isInstalled` method return true. In case of `HelloFacet` only when user current directory is named "hello". To get user current directory we ask forge, through dependency injection, for the current project. Project is a Java object that holds information of the current user project like its directory.

To activate the facet we must annotate `HelloWorld` plugin with `RequiresFacet`:

Listing 2.3: Forge plugin with facet example

```
@Alias("hello-world")
@RequiresFacet(HelloFacet.class)
public class HelloWorldPlugin implements Plugin {

    @Command(value = "sayHello")
    public void countBundles(PipeOut out) {
        out.println("Hello World" );
    }
}
```

2.3.4 Project Locator

A project locator is a component responsible for creating Java objects that represent useful information in the forge runtime, they are called *project* in forge. Forge calls all locators available when user change directory in command line and the first locator that is matched will create a Java object representing the current Project. Its the same idea of facets but instead of restricting plugin commands it creates object and made them available for Forge runtime. That was how we could inject current user project in `HelloFacet` before.

2.3.5 Applications

Forge can be used as a command line tool or integrated in main IDEs like Eclipse, Netbeans or IntelliJ. To be used as command line tool one must download a zip distribution containing a forge executable that runs on main operating systems³⁰.

Forge has an important role on this work as it was the ground for creating *Intrabundle*, a tool based on forge runtime that will be introduced later.

³⁰As forge runs on top of Java, Forge inherits its *universality*

3 INTRABUNDLE - AN OSGI BUNDLE INTROSPECTION TOOL

3.1 Introduction

It was clear in previous chapters that modular and non modular applications have many differences and specific features hence the need for dedicated approach for quality analysis. This chapter presents a tool called *Intrabundle* (intrabundle github, 2014), an open source Java based application created in the context of this work. Intrabundle introspects OSGi projects collecting useful information and calculates OSGi bundle and project **internal quality**.

3.2 Design Decisions

To analyze and extract data from large code bases of OSGi projects, which can vary from KLOCs to thousands of KLOCs, there was the need of a lightweight approach. Some *functional requirements* were:

- Analyze different formats of OSGi projects like Maven¹, Eclipse projects and BND²;
- It should be able to dive deep into projects source code like counting methods calls, differentiate classes and interfaces and so on;
- Get general informations like project version, revision or latest commit in source repository;
- Should be easy analyze lots of projects;
- Should output a detailed Human readable quality report so the extracted information can be analyzed.

and the following *non functional requirements*:

- Only open sourced projects³ because we focus on internal quality where the code is important;
- The tool should be lightweight to analyze real, complex and huge OSGi projects;
- Find and Introspect manifest files where valuable OSGi information rely;
- Should be testable;
- Fast;
- Use Java to leverage the author's experience in the language;
- Use a good file system API⁴ because file manipulation is one of the most frequent tasks the tool should perform.

¹Maven is a build tool for Java

²BND is a tool to easy OSGi projects development

³Projects that have its source code made available with a license in which the copyright holder provides the rights to study, change and distribute the software to anyone and for any purpose

⁴An API expresses a software component in terms of its operations, inputs, outputs, and underlying types.

The following alternatives were evaluated:

1. Build a standalone Java client application using javaFX⁵;
2. Create an eclipse plugin⁶;
3. Create a Maven plugin⁷;
4. Build the tool on top of JBoss Forge;
5. Extend an existing static/internal analysis tool like PMD.

The chosen among the above options was JBoss Forge, due to the following facts:

- Works inside and outside eclipse;
- Works regardless of build tool;
- As its a command line tool its very lightweight and can analyze multiple OSGi projects at the same time;
- The programing model is based on top of the so called CDI⁸ so managing Objects lifecyle and event is handled by CDI automatically;
- Forge has a very well established and documented file system manipulation api based on java.io;
- Forge is very flexible so generating quality reports is a matter of using a report API inside it;
- The author already had experience with JBoss Forge and CDI.

Creating an eclipse plugin for analyzing OSGi projects could be not as lightweight as forge plugin. We would need eclipse started and OSGi projects imported inside IDE so the eclipse plugin could identify the project resources.

JavaFX would require use standard Java file system manipulation api(java.io) which has many caveats and pitfalls so for example its easy to create a memory leak or too many files opens error. Also with JavaFX there the need to implement the interface/GUI which is already well done in Eclipse or Forge.

Maven plugins are limited to maven projects.

PMD⁹ has a very limited API so it could be hard to generate reports or analyze multiple projects using it.

⁵JavaFX is a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications

⁶Eclipse plug-ins are software components with the objective to extend Eclipse IDE

⁷Maven is a build tool that consists of a core engine which provides basic project-processing capabilities and build-process management, and a host of *plugins* which are used to execute the actual build tasks.

⁸Context and Dependency Injection for the Java platform. CDI is a dependency injection framework where instead of dependencies construct themselves they are injected by some external means, in cae CDI

⁹A very nice tool for static code analysis. It is based on rules that can be created via xml or xpath expression. When a rule is violated it can output warns or errors to the console

3.3 Implementation Overview

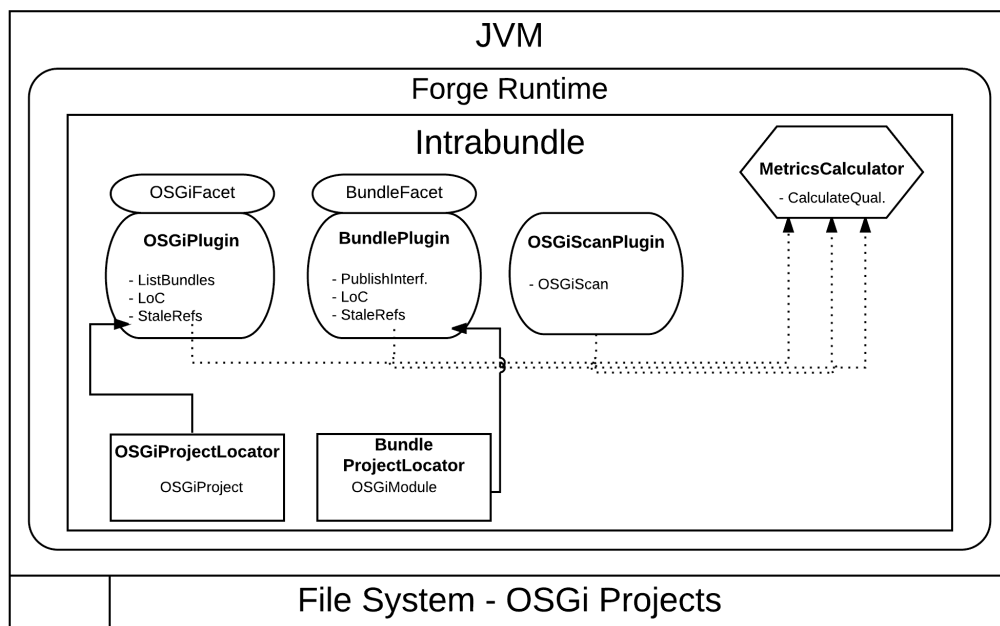
Intrabundle is composed by 3 Forge plugins, see section 2.3.2 for details about Forge plugins. The first is *BundlePlugin* which extracts OSGi bundle information, second is *OSGiPlugin* that has a vision of all bundles composed by the project. Third is *OSGiScan* a plugin responsible for scanning OSGi bundles recursively in file system. Another component in the architecture is *MetricsCalculator* that calculates bundle and OSGi project quality based on metrics produced by *OSGiPlugin* and *BundlePlugin*.

Intrabundle also provides 2 facets, see section 2.3.3 for details about Forge facets. *BundleFacet* and *OSGiFacet*, both restricts commands provided by *BundlePlugin* and *OSGiPlugin* in the context of OSGi bundle and project respectively. *BundleFacet* is active when user enter on a directory containing an OSGiBundle and *OSGiFacet* is active when user enters on a directory that contains at least one OSGiBundle. When *BundleFacet* is active then *OSGiFacet* is disabled meaning that only *BundlePlugin* commands will be active.

Another important component in Intrabundle architecture is the Project Locator, see section 2.3.4 for details about Forge locators. Intrabundle provides 2 locators. The first is *BundleLocator* that creates a Forge project object named *OSGiModule* representing and gathering data related to OSGi bundle. *BundleLocator* is activated when user is at an OSGi bundle directory. The second is *OSGiProjectLocator* which creates a Forge project object named *OSGiProject* representing an OSGi project which is a collection of bundles. *OSGiProject* locator is activated when user is in a directory that has at least one child directory that is an OSGiBundle.

Figure 3.1 illustrates Intrabundle architecture:

Figure 3.1: Intrabundle Architecture



3.4 Identifying OSGi Projects and Bundles

One important task that both *facets* and *locators*, provided by Intrabundle, perform is identifying OSGi bundles or OSGi projects. To do that the tool searches for OSGi meta data in *MANIFEST* file¹⁰. So identifying bundles is as simple as locating the Manifest and verifies if it's content has OSGi information. The main problem is that the manifest location can vary depending on the project format. Table 3.1 lists the types of OSGi projects Intrabundle recognizes:

Table 3.1: Supported types of OSGi projects

Type	Manifest location
Maven projects	/src/main/resource/META-INF.
Maven using BND tools	pom.xml ¹¹ with maven-bundle-plugin.
Standard Eclipse Java projects	/META-INF
Standard BND Tools	bnd.bnd file in any subfolder.
Package based bundles(Jitsi project)	each package has a manifest.

In the extent of this work, **OSGi projects** are collections of OSGi bundles in the same directory but its also important to say that OSGi bundles can be installed from anywhere from the file system or network.

3.5 Collecting Bundle Data

After identifying OSGi bundles and OSGi projects Intrabundle needs to extract useful information from them. Table 3.2 shows which information the tool is collecting:

¹⁰The manifest is a special file that can contain information about the files packaged in a JAR file. By tailoring this "meta" information that the manifest contains, you enable the JAR file to serve a variety of purposes.

Table 3.2: Extracted data from OSGi projects

Name	Description
Loc	Lines of code.
Declarative services	Verifies if bundles uses declarative service ¹² .
Ipojo	Verify if bundles uses Ipojo ¹³
Blueprint	Verify if bundles uses Blueprint ¹⁴
Stale References	Looks for possible Stale services references.
Publishes Interface	Verifies if bundle exposes only its interfaces(API).
Declares permission	Verifies if bundle declares permission.
Number of classes	Counts bundle's classes.
Number of abstract classes	Counts bundle's abstract classes.
Number of interfaces	Counts bundle's interfaces.
Bundle dependencies	Gather bundle dependencies.
Required bundles	Gather bundle required bundles.

Some observations about collected data. **Lines of code** are an indicative of high or low cohesion, if the component has too much lines of code its an evidence that it is probably doing more work then it should. LoC is a classical software metric that was adapted in this work to OSGi Bundles. **Ipojo**, **Blueprint** and **Declarative Services** are recommended for managing OSGi services because they hide the "dirty work" of publishing and consuming services which sometimes may lead to incorrect behavior. For example forgetting to release a service when bundle is stopped. **Stale Services References**¹⁵ are detected via approximation, in other words, Intrabundle counts the number of services *gets* and *ungets*¹⁶ for each class a bundle has. If the number of gets and ungets are equal then the class have no stale references, otherwise it is considered as having stale references. **Bundle dependencies** are calculated by looking at OSGi Manifest file in exported and imported packages. If bundle A *imports* package *x.y.z* and bundle B *exports* package *x.y.z* we say that bundle A depends on bundle B. **Required bundles** just counts the number of required bundles declared in manifest. **Publishes interfaces** looks at bundle exported packages, if all exported packages contains only interfaces we say that bundle only imports interfaces. **Declares permission** verifies if bundle implements security by contract searching for *permission.perm* file.

Each information retrieved by Intrabundle is usually mapped to a Forge command, see Listing 3.1 which is the command that prints bundle exported packages to the Forge console:

¹⁵Refers to code that may retain OSGi service references even when the providing bundles are gone (Gama and Donsez, 2012)

¹⁶Operations that consume and release a service reference respectively

Listing 3.1: Exported packages command

```

@Command(value = "exportedPackages", help = "list bundle exported
packages")
public void exportedPackages(PipeOut out) {
    if (bundle.getExportedPackages().isEmpty()) {
        out.println(messageProvider.getMessage("module.
noExportedPackages"));
    }
    else {
        for (String s : bundle.getExportedPackages()) {
            out.println(s);
        }
    }
}

```

All the logic is inside **bundle** variable, which is an immutable object¹⁷, in method *getExportedPackages*. The bundle variable is provided by a Forge locator when user navigates to a directory which is an OSGi bundle, as explained in section 2.3.4.

3.6 Quality Calculation

The data collected earlier will be materialized into six metrics that will be used to calculate OSGi projects quality. We saw on section 2.1.4 that a software metric is a quantitative calculation of a software attribute. This section shows which metrics were created based on extracted information.

3.6.1 Quality labels

Every created metric in this work can be classified into the following *quality labels*:

1. **STATE OF ART**: Metric fully satisfies good practices;
2. **VERY GOOD**: Satisfies most recommendations;
3. **GOOD**: Satisfies recommendations;
4. **REGULAR**: Satisfies some recommendations;
5. **ANTI PATTERN**: Does not satisfies any recommendation and follows some bad practices.

¹⁷Is an object whose state cannot be modified after it is created. A good practice and core principle in domain driven design (Evans and Fowler, 2003)

3.6.2 Metrics Created

The first metric created is **LoC**, its the simplest one. LoC is based on bundle lines of code(excluding comments) meaning that the less lines of code more *cohesion* the bundle has and easier to maintain it should be. This metric is an estimation, there is no exact LoC number because it depends on the context(e.g. complexity). To classify LoC metric we use the following rule:

$$\text{LoC} = \begin{cases} \text{STATE OF ART} & \text{if LoC} \leq 700, \\ \text{VERY GOOD} & \text{if LoC} \leq 1000, \\ \text{GOOD} & \text{if LoC} \leq 1500, \\ \text{REGULAR} & \text{if LoC} \leq 2000, \\ \text{ANTI PATTERN} & \text{if LoC} > 2000. \end{cases}$$

Second metric is **Publishes interfaces** meaning that bundles should hide their implementation and expose only it's API. It is a good practice expose only the API and hide the implementation details from consumers. This is considered an *Usability pattern* (Knoernschild et al., 2012). Here is how this metric is calculated:

$$\text{Publishes interfaces} = \begin{cases} \text{STATE OF ART} & \text{if publishes only interfaces,} \\ \text{REGULAR} & \text{if not publishes only interfaces,} \end{cases}$$

Next metric is **Bundle dependencies**, it evaluates the coupling between bundles. The less coupled a bundle is the more reusable and maintainable it will be. It is considered a base pattern called *Manage Relationships* in (Knoernschild et al., 2012). Here is how this metric is calculated by Intrabundle:

$$\text{Bundle dependencies} = \begin{cases} \text{STATE OF ART} & \text{if Bundle dependencies} = 0, \\ \text{VERY GOOD} & \text{if Bundle dependencies} \leq 3, \\ \text{GOOD} & \text{if Bundle dependencies} \leq 5, \\ \text{REGULAR} & \text{if Bundle dependencies} \leq 9, \\ \text{ANTI PATTERN} & \text{if Bundle dependencies} \geq 10. \end{cases}$$

Next one is **Uses framework**, in complex application it is important to use a framework to manage bundle services. This metrics takes into account 3 well known frameworks by OSGi application: *IPojo*, *Declarative services* and *Blueprint*:

$$\text{Uses framework} = \begin{cases} \text{STATE OF ART} & \text{if uses framework,} \\ \text{REGULAR} & \text{if not using framework,} \end{cases}$$

Next metric is **Stale references**, it focus on a very common problem in OSGi which can lead to resource and memory leaks (Gama and Donsez, 2011). Intrabundle calculate this metric by counting specific methods calls to OSGi services in a bundle. What Intrabundle does is an approximation and may lead to false positives. To get a real value for this software attribute one have to calculate it by dynamic analysis like done in (Gama and Donsez, 2012):

$$NC = \sum_{i=1}^n \text{ where } n = \text{number of classes a bundle have.}$$

$$NS = \sum_{i=1}^n \text{ where } n = \text{number of stale references found.}$$

$$\text{Stale references} = \begin{cases} \text{STATE OF ART} & \text{no stale references,} \\ \text{GOOD} & \frac{NS}{NC} < 0.1, \\ \text{GOOD} & \frac{NS}{NC} < 0.25, \\ \text{REGULAR} & \frac{NS}{NC} < 0.5, \\ \text{ANTI PATTERN} & \frac{NS}{NC} \geq 0.5. \end{cases}$$

In other words if no stale references is found then this metric receives a *state of art* quality label, if less then 10% of bundle classes have stale references(number of get and unget doesn't match) then it receives *very good* quality label, if > 10% and < 25% then it is *good*, if the number of stale references is between 25% and 50% its is *regular* but if it has 50% or more classes with stale references then its considered an *anti pattern*.

The last metric created in this work is **Declares permission**, it is concerned with security. In this metric Intrabundle searches for permissions.perm file in the bundle, if it finds it then the metric is considered state of art:

$$\text{Declares permission} = \begin{cases} \text{STATE OF ART} & \text{if declares permission,} \\ \text{REGULAR} & \text{if not declares permission,} \end{cases}$$

3.6.3 Quality Formula

OSGi *project* quality and *bundle* quality are calculated by Intrabundle using the quality labels. Each quality *label* adds points to bundle and project final quality which is based on percentage of quality points(QP) obtained. *State of art* add **5QP**, *Very good* **4QP**, *Good* adds **3QP**, *Regular* **2QP** and *Anti pattern* adds **1QP**

3.6.3.1 Bundle Quality

Bundle final quality is calculated as a function of *Total Quality Points* **TQP**, which is the total points obtained on each created metric, and *Maximum Quality Points* **MQP**, the maximum

points a bundle can have. MQP is equal to all metrics classified as State of art. Here is the formula:

$$MQP = \sum_{i=1}^n 5 \text{ where } n = \text{number of metrics.}$$

$$TQP = \sum_{i=1}^n q(i) \text{ where } n = \text{number of metrics and } q(i) \text{ is QP obtained in metric } i.$$

$$f(q) = \frac{TQP}{MQP};$$

if $1 \leq f(q) < 0.9$ then State of Art;

if $0.9 \leq f(q) < 0.75$ then Very Good;

if $0.75 \leq f(q) < 0.6$ then Good;

if $0.6 \leq f(q) < 0.4$ Regular;

if $0.4 \leq f(q)$ then Anti Pattern;

In terms of percentage of points obtained, more than 90% of TQP is considered State of Art, between 90% and 75% is Very good quality, from 60% to 75% is Good, 40% to 60% is Regular and less than 40% of TQP a bundle is considered Anti pattern in terms of software quality.

For example imagine we have three metrics and a bundle has 5QP(State of art) and 3QP(Good quality label) in the other two metrics> In this case the MQP is 15 (5*3) and TQP is 11(5 + 3 +3). In this example the bundle quality is 11/15(73%) which maps to *Good* quality label.

3.6.3.2 Project Quality

In Intrabundle, the quality of an OSGi project uses the same formula of bundle quality. The only difference is in MQP and TQP which in this case are based on bundles quality instead of metrics. In project quality the maximum point is calculated considering all bundle's quality as State of art, so for example if we have 3 bundles the MQP will be 15. TQP is just the sum of all bundles quality, here is the formula Intrabundle uses for *project quality*:

$$MQP = \sum_{i=1}^n 5 \text{ where } n = \text{number of bundles in the project.}$$

$$TQP = \sum_{i=1}^n q(i) \text{ where } n = \text{number of bundles and } q(i) \text{ is QP obtained by bundle } i.$$

$$f(q) = \frac{TQP}{MQP};$$

if $1 \leq f(q) < 0.9$ then State of Art;
 if $0.9 \leq f(q) < 0.75$ then Very Good;
 if $0.75 \leq f(q) < 0.6$ then Good;
 if $0.6 \leq f(q) < 0.4$ Regular;
 if $0.4 \leq f(q)$ then Anti Pattern;

In terms of percentage it's also the same rule used for bundle's quality. For example if a project has 3 bundles, one has 5QP(State of art) and other two has 3QP(good) then MQP for this case is 15(5*3) and TQP is 11(5 + 3 +3). In this example project final quality is 11/15(73%) which maps to *Good* quality label.

3.6.3.3 Project metric quality

The last way to measure quality using Intrabundle is to analyze the project quality on each metric. The project quality in a metric is the sum of all *bundles qualities* on that metric. The total points a bundle can have in a metric is considering all bundles State of art on that metric. The quality label for a project metric quality is also define as percentage of points obtained from maximum points:

$$MQP = \sum_{i=1}^n 5 \text{ where } n = \text{number of bundles in the project.}$$

$TQP = \sum_{i=1}^n q(i)$ where n = number of bundles and $q(i)$ is QP obtained by bundle i in the metric.

$$f(q) = \frac{TQP}{MQP};$$

if $1 \leq f(q) < 0.9$ then State of Art;
 if $0.9 \leq f(q) < 0.75$ then Very Good;
 if $0.75 \leq f(q) < 0.6$ then Good;
 if $0.6 \leq f(q) < 0.4$ Regular;
 if $0.4 \leq f(q)$ then Anti Pattern;

So for example if a project has 3 bundles, one has 5QP(State of art) in LoC and other two has 3QP(good) then MQP is 15(5*3) and TQP is 11(5 + 3 +3). In this example project final quality on LoC is 11/15(73%) which maps to *Good* quality label.

3.7 Intrabundle Reports

The tool generates two reports based on information it collects from bundles so the it can be analyzed carefully in one place. The reports can be generated in various formats (txt, pdf, html,

csv and excel). Figure 3.2 shows an example report:

Figure 3.2: Intrabundle general report

OSGi Quality Analysis	
General Report	
Project Location:	/home/rmpestano/projetos/OSGi/HelloOSGi
N° of bundles:	5
Lines of code:	2555
Lines of test code:	0
N° of stale references:	3
Revision:	commit a82b0b8ffa7e7f542d56b4cc1478ec4bb1647df9 1364139651 -----p
Project most frequent quality:	VERY_GOOD
Project quality points:	15 of 25
Project final quality:	GOOD
Bundles Found	
/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiFromBrazil	
/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiFromBrazilMaven	
/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiFromFrance	
/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiFromUSA	
/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiMain	

The first section of the report gives an overall idea of the project, second part lists information of each bundles, see Figure 3.3

Figure 3.3: Intrabundle general report - detailed section

Listing bundle Information	
Name:	HelloOSGiFromBrazil
Location:	/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiFromBrazil
Version:	1.0.0.qualifier
Lines of code:	57
Publishes interfaces:	Yes
Uses Declarative services:	No
Uses Blueprint:	No
Activator:	helloosgifrombrazil.Activator
Number of packages:	2
Number of classes:	2
Interfaces/abstract classes:	0
Uses Ipjojo:	No
Imported packages:	helloosgi.main.api org.osgi.framework.version="1.3.0"
Bundle dependencies:	/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiMain
Bundle metric points:	Points obtained: 23 of 30. Final score: VERY_GOOD
Name:	HelloOSGiFromBrazilMaven
Location:	/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiFromBrazilMaven

Another report Intrabundle generates is a metric report that details the punctuation of each metric, see Figure 3.4:

Figure 3.4: Intrabundle metrics report

OSGi Quality Analysis	
Metrics Report	
Project Location:	/home/rmpestano/projetos/OSGi/HelloOSGi
N° of bundles:	5
Lines of code:	Points obtained: 21 of 25 - 84%. Final score: VERY_GOOD
Bundle dependency:	Points obtained: 21 of 25 - 84%. Final score: VERY_GOOD
Publishes interfaces:	Points obtained: 25 of 25 - 100%. Final score: STATE_OF_ART
Uses framework:	Points obtained: 10 of 25 - 40%. Final score: REGULAR
Stale references:	Points obtained: 16 of 25 - 64%. Final score: GOOD
Declares permission:	Points obtained: 10 of 25 - 40%. Final score: REGULAR
Bundle most frequent quality:	GOOD
Project quality points:	17 of 25 - 68%
Project final quality:	GOOD

As in general report, in metrics report the first section of the report gives an overall idea of the project, second part lists information of each bundles, see Figure 3.5

Figure 3.5: Intrabundle metrics report - detailed section

Listing bundle metrics result	
Name:	HelloOSGiFromBrazil
Location:	/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiFromBrazil
Bundle metric points:	Points obtained: 23 of 30. Final score: VERY_GOOD
Lines of code:	57 - STATE_OF_ART(5)
Publishes interfaces:	Yes - STATE_OF_ART(5)
Bundle dependencies:	1 - VERY_GOOD(4)
Uses framework:	No - REGULAR(2)
Stale references:	0 of 2 classes - STATE_OF_ART(5)
Declares permission:	No - REGULAR(2)
Name:	HelloOSGiFromBrazilMaven
Location:	/home/rmpestano/projetos/OSGi/HelloOSGi/HelloOSGiFromBrazilMaven
Bundle metric points:	Points obtained: 18 of 30. Final score: GOOD
Lines of code:	1209 - GOOD(3)
Publishes interfaces:	Yes - STATE_OF_ART(5)
Bundle dependencies:	1 - VERY_GOOD(4)
Uses framework:	No - REGULAR(2)
Stale references:	1 of 3 classes - REGULAR(2)
Declares permission:	No - REGULAR(2)

All reports generated by Intrabundle can be found online intrabundle reports (2014).

3.8 Intrabundle Quality

In this section we will see how Intrabundle's quality is managed and how some concepts of *section 2.1* were applied to the project. As the project is not OSGi based we can't apply Intrabundle's metrics on itself so we used classical approaches to assure the quality of the project.

3.8.1 Internal quality

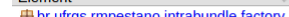
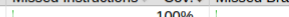
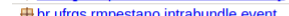

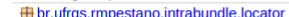

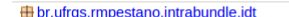

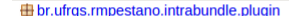
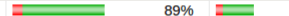










Intrabundle internal quality is managed by PMD and JaCoCo. PMD is an static analysis tool and JaCoCo a dynamic analysis one. Both were presented in section 2.1.6 with the objective to guarantee non functional requirements.

3.8.1.1 Example

PMD was already illustrated at Chapter 2 as an example of static analysis tool. JaCoCo is used to calculate code coverage to track files and methods that automated tests are covering. Figure 3.6 shows JaCoCo code coverage report for Intrabundle:

Figure 3.6: Intrabundle code coverage

intrabundle

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
br.ufgrs.rmpestano.intrabundle.factory		100%		83%	1	6	0	6	0	3	0	1
br.ufgrs.rmpestano.intrabundle.event		100%		n/a	0	3	0	5	0	3	0	2
br.ufgrs.rmpestano.intrabundle.locator		97%		70%	3	11	2	34	0	6	0	2
br.ufgrs.rmpestano.intrabundle.jdt		95%		88%	2	11	1	18	1	7	0	2
br.ufgrs.rmpestano.intrabundle.plugin		89%		80%	27	114	20	251	4	54	0	4
br.ufgrs.rmpestano.intrabundle.facet		89%		73%	15	38	7	40	3	16	0	4
br.ufgrs.rmpestano.intrabundle.model		81%		61%	138	289	101	497	11	89	0	7
br.ufgrs.rmpestano.intrabundle.metric		78%		64%	15	40	21	100	0	13	0	2
br.ufgrs.rmpestano.intrabundle.util		76%		64%	67	140	52	197	4	34	0	7
br.ufgrs.rmpestano.intrabundle.i18n		67%		100%	0	6	3	12	0	5	0	1
Total	1,040 of 6,080	83%	293 of 852	66%	268	658	207	1,160	23	230	0	32

Created with JaCoCo 0.7.1

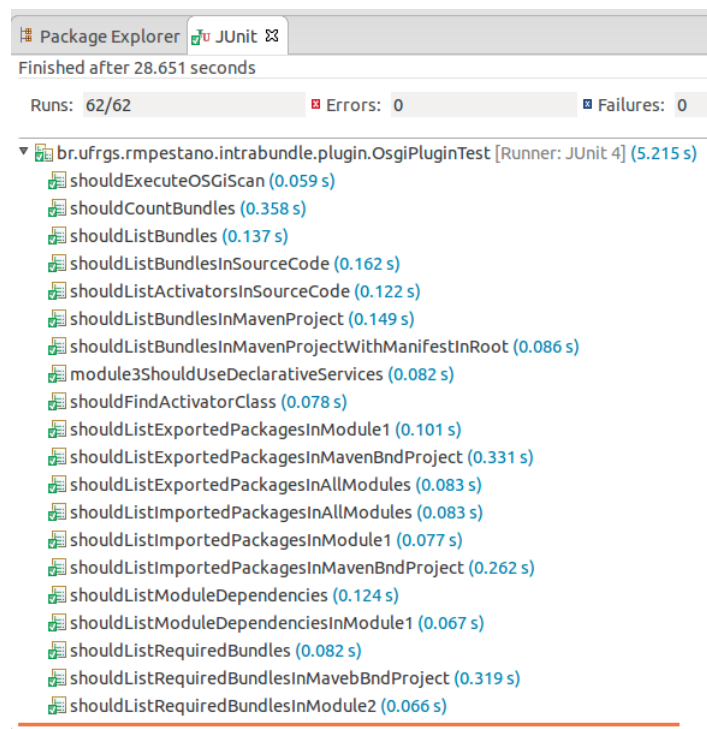
3.8.2 External quality

Intrabunde external quality is assured by automated whitebox tests so we can verify if Intrabundle is working as expected, if it meets its functional requirements.

3.8.2.1 Example

As of November 2014 Intrabundle performs 65 **integration tests** which can be defined as automated tests aimed to detect any inconsistencies between the software units that are integrated together. In this kind of automated tests the system must be running and in case of Intrabundle we also need the Forge runtime up and running during tests and that is done by Arquillian (dan, 2011), an integration test platform. Figure 3.7 shows the result of integration tests execution:

Figure 3.7: Intrabundle integration tests



4 BUNDLE INTROSPECTION RESULTS

Intrabundle was used to introspect and apply its metrics to 10 real OSGi projects, the projects are all open sourced and vary in size, teams and domain.

4.1 Analyzed Projects

In this section is presented an overview of projects that were analyzed during this work. Table 4.1 shows projects in terms of *size*:

Table 4.1: OSGi projects analyzed by Intrabundle

Name	Number of bundles	LoC
BIRT	129 (217)	2,226,436
Dali	35 (46)	1,058,16
Jitsi	155 (158)	607,144
JOnAS	117 (122)	366,940
Karaf	58 (60)	93,743
Openhab	181 (184)	347,492
OSEE	183 (190)	873,690
Pax CDI	21 (22)	19,480
Tuscany Sca	138 (140)	243,494
Virgo	36 (49)	77,859
Sum	1051	4,962,094

Note that number of bundle in parenthesis is considering bundles with zero lines of code which, in the extent of this work, are not considered for quality analysis. Also note that lines of code is considering only .java files removing comment lines.

Below is a brief description of each project:

1. **BIRT**: is an open source software project that provides the BIRT technology platform to create data visualizations and reports that can be embedded into rich client and web applications, especially those based on Java and Java EE;
2. **Dali**: The Dali Java Persistence Tools Project provides extensible frameworks and tools for the definition and editing of Object-Relational (O/R) mappings for Java Persistence API (JPA) entities;
3. **Jitsi**: is an audio/video Internet phone and instant messenger written in Java. It supports some of the most popular instant messaging and telephony protocols such as SIP, Jabber/XMPP (and hence Facebook and Google Talk), AIM, ICQ, MSN, Yahoo! Messenger;
4. **JOnAS**: is a leading edge open source Java EE 6 Web Profile certified OSGi Enterprise Server;

5. **Karaf:** Apache Karaf is a small OSGi based runtime which provides a lightweight container onto which various components and applications can be deployed
6. **Openhab:** a open source home automation software for integrating different home automation systems and technologies into one single solution that allows over-arching automation rules and that offers uniform user interfaces;
7. **OSEE:** The Open System Engineering Environment is an integrated, extensible tool environment for large engineering projects. It provides a tightly integrated environment supporting lean principles across a product's full life-cycle in the context of an overall systems engineering approach;
8. **Pax CDI:** brings the power of Context and Dependency Injection(CDI) to the OSGi platform;
9. **Tuscany SCA:** is a programming model for abstracting business functions as components and using them as building blocks to assemble business solutions;
10. **Virgo:** is a completely module-based Java application server that is designed to run enterprise Java applications and Spring-powered applications with a high degree of flexibility and reliability;

4.2 Projects Quality Results

In this section will be presented the resulting qualities of analyzed projects and some comparisons. First comparison groups all analyzed projects comparing their *bundle quality* and *metric quality*. Later the projects are separated by groups in terms of size of LoC and number of bundles.

All projects quality reports that provided data for all comparisons are available online, see intrabundle reports (2014) for detailed information.

4.2.1 General quality comparison

The first table shows general projects qualities, it is ordered by quality points percentage. Its important to note that each projects maximum quality points(MQP) is different because it depends on the number of bundles, see ?? for further information:

Table 4.2: Projects general quality

Name	TQP	MQP	Points percent	Quality label
Pax CDI	84	105	80%	Very Good
Openhab	666	905	73.6%	Good
Virgo	132	180	73.3%	Good
Karaf	211	290	72.8%	Good
OSEE	596	915	65.1%	Good
Tuscany Sca	433	690	62.8%	Good
JOnAS	356	585	60.9%	Good
Jitsi	414	775	53.4%	Regular
Dali	86	175	49.1%	Regular
BIRT	315	645	48.8%	Regular

The winner on general category, considering Intrabundle metrics, is **Pax CDI** project which obtained 80% of quality points and received a *Very Good quality label*. Pax CDI is a project from *OPS4J - Open Participation Software for Java* which is a community that is trying to build a new, more open model for Open Source development, where not only the usage is Open and Free, but the Participation is Open as well.

4.2.2 Metric quality comparison

The next category analyzes how good the projects are on each metric. It's important to note that each project *maximum quality points*(MQP) in a metric depends on the number of bundles, see 3.6.3.3 for more details. Value in table 4.3 are the *total quality points*(TQP) obtained and in parenthesis is the percentage of MQP that the value represents:

Table 4.3: Projects quality by metrics

Name	MQP	LoC	Publishes interfaces	Uses framework	Bundle dependency	Stale references
BIRT	645	294 (45.6%)	393 (60.9%)	258 (40%)	307 (47.6%)	644 (99.8%)
Dali	175	74 (42.3%)	175 (100%)	70 (40%)	65 (37.1%)	174 (99.4%)
Jitsi	775	492 (63.5%)	775 (100%)	310 (40%)	459 (59.2%)	473 (61%)
JOnAS	585	358 (61.2%)	480 (82.1%)	252 (43.1%)	481 (82.2%)	573 (97.9%)
Karaf	290	212 (73.1%)	257 (88.6%)	158 (54.5%)	290 (100%)	278 (95.9%)
Openhab	905	672 (74.3%)	791 (87.4%)	806 (89.1%)	664 (73.4%)	901 (99.6%)
OSEE	915	584 (63.8%)	909 (99.3%)	573 (62.6%)	529 (57.8%)	881 (96.3%)
Pax CDI	105	85 (81%)	105 (100%)	66 (62.9%)	103 (98.1%)	98 (93.3%)
Tuscany SCA	690	472 (68.4%)	684 (99.1%)	288 (41.7%)	451 (65.4%)	682 (98.8%)
Virgo	180	127 (70.6%)	180 (100%)	78 (43.3%)	176 (97.8%)	162 (90%)

Following is the champions on each metric:

- *LoC*: Pax CDI has Very Good quality label on LoC;
- *Publishes interfaces*: Dali, Jitsi, Pax CDI and Virgo are all tied on metric points(100%) and are labeled *State of Art* on this metric;
- *Uses framework*: Openhab is *Very Good* (almost State of art) on this metric;
- *Bundle dependency*: Karaf is leading with 100% and is *State of art* on this metric followed by Pax CDI and Virgo which are also State of art($\geq 90\%$) but not with 100% of quality points;

- *Stale references*: Birt is leading on this metric, it has only one (probably) stale reference class among its 2 million line of code. Openhab loses by 0.2% with 2 stale references on its 300 thousands of lines of code.
- *Declares permission*: Birt is the only analyzed project that has a bundle which declares permission;

Some interesting facts can be observed looking at table 4.3:

Birt was the only project to use OSGi permission mechanism among analyzed projects. In fact with 40.5%¹ means that only one Birt bundle declared permission which was *org.eclipse.birt.report.engine.emitter.postscript*.

Eclipse Dali project has the worst *dependency quality* metric which is a sign that its bundles are high coupled, as opposed to **Karaf** which may have low coupled bundles.

Projects that *use a framework* for managing services usually has less stale references because they are not likely to code for publish or consume service as a framework is doing that for them.

Jitsi has more *Stale references* which may affect its *reliability*. Although it has lots of stale references compared to other projects it received a *Good* quality label which means that this metric formula is not well dimensioned and may be revisited in future.

It looks like *publishing only interfaces* and hiding implementation is a well known and disseminated good practice as we have good punctuation on this metric in most analyzed projects.

We have evidences that **Pax CDI** has the more cohesive bundles as they have less lines of code than bundles of other projects.

¹When a bundle does not declares permission it receives 2 metric points(regular label). So if a project has all bundles with regular label it will have 40% of MQP

5 CONCLUSION

REFERENCES

- Kan S. H. **Metrics and Models in Software Quality Engineering**. 2nd Edition [S.l]: Addison Wesley, 2002. p. 113-114
- Evans, E.; Fowler, M. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. 1st Edition, [S.l]: Prentice Hall, August 2003. p. 75
- Arnold K.; Gosling J.; Holmes D. **THE Java™ Programming Language**. 4th Edition. [S.l]:Addison Wesley Professional, 2005.
- Hall, R. S.; Pauls K.; McCulloch S.; Savage D. **OSGi in Action:Creating Modular Applications in Java**. 1st Edition, Stamford: Manning Publications Co., May 2011.
- Knoernschild, K.; Martin, R. C.; Kriens, P. **Java Application Architecture: Modularity Patterns with Examples Using OSGi**. 1st Edition, [S.l]: Prentice Hall, March 2012.
- GAMA, K.; DONSEZ, D. **Towards Dependable Dynamic Component-based Applications**. [s.n.], Grenoble, October 2011. p. 94
- BEOHM, B.; BASILI, V. R. Software Defect Reduction Top 10 List. **Computer**, Los Angeles, v. 34, no. 1, pp 135-137, January 2001.
- WANG, T.; WEI, J.; ZHANG, W.; ZHONG, H. A Framework for Detecting Anomalous Services in OSGi-based Applications. **IEEE SCC 2012**, Honolulu, June 2012. p. 1-2
- KHAN, M. E.; KHAN, F. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. **International Journal of Advanced Computer Science and Applications**, New York, v. 3, no. 6, pp 12-15, June 2012.
- Wichmann, B. A.; Canning, A. A.; Clutterbuck, D. L.; Winsbarrow, L. A.; Ward, N. J.; Marsh, D. W. R. Industrial Perspective on Static Analysis. **Software Engineering Journal**, [S.l], v. 10, pp 69–75, Mar 1995.
- KNORR, E. What microservices architecture really means. **InfoWorld**, Available at: <http://www.infoworld.com/article/2682502/application-development/application-development-what-microservices-architecture-really-means.html>. Accessed in: November 18 2014.
- KRILL, P. Project Jigsaw delayed until Java 9. **InfoWorld**, Available at: <http://www.infoworld.com/article/2617584/java/project-jigsaw-delayed-until-java-9.html>. Accessed in: November 21 2014.

- GAMA, K.; DONSEZ, D. **Service Coroner: A Diagnostic Tool for Locating OSGi Stale References**. Available at: <http://www-adele.imag.fr/Les.Publications/intConferences/ECBSE2008Gam.pdf>. Accessed in: November 01 2014.
- HAMZA, S.; SADOUE, S.; FLEURQUIN, R. Measuring Qualities for OSGi Component-Based Applications. **International Conference on Quality Software**, Najing, pp 25-34, July 2013.
- Dan Allen. Arquillian - A Component Model for Integration Testing. **Jaxenter**, Mar 2011. Available at: <http://jaxenter.com/arquillian-a-component-model-for-integration-testing-103003.html>. Accessed in: November 19 2014.
- IEEE Spectrum. Top 10 Programming Languages. **Spectrum's 2014 Ranking**, Jul 2014. Available at: <http://spectrum.ieee.org/computing/software/top-10-programming-languages>. Accessed in: November 19 2014.
- CISQ. Specification for Automated Quality Characteristic Measures. **CISQ quality standard version 2.1**. Available at: <http://it-cisq.org/wp-content/uploads/2012/09/CISQ-Specification-for-Automated-Quality-Characteristic-Measures.pdf>. Accessed in: November 14 2014.
- ECLIPSE. **Eclipse Platform Technical Overview**. Available at: <http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf>. Accessed in: November 30 2014.
- SQA. Software Quality Metrics. [S.l.:s.n]. Available at: <http://www.sqa.net/softwarequalitymetrics.html>. Accessed in: November 14 2014
- ISO25010:2011 System and software quality models. **Systems and software Quality Requirements and Evaluation (SQuaRE)**. Available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35733. Accessed in: November 16 2014.
- Intrabundle PMD ruleset. Available at: <https://github.com/rmpestano/intrabundle/tree/master/src/test/resources/rulesets/pmd.xml>. Accessed in: November 14 2014.
- Intrabundle github repository. Available at: <https://github.com/rmpestano/intrabundle>. Accessed in: November 19 2014.
- Intrabundle quality reports. Available at: <http://rmpestano.github.io/intrabundle/#reports>. Accessed in: December 01 2014.
- Semantic Versioning. Available at: <http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf>. Accessed in: November 20 2014.

OSGi Framework Architecture - Three Conceptual Layers. Available at: <http://www.programcreek.com/2011/07/osgi-framework-architecture-three-conceptual-layers/>. Accessed in: November 20 2014.

Appendices

AppendixA INTRABUNDLE USAGE EXAMPLE

A.1 Setup environment

- instaling forge;
- instaling intrabundle;
- starting forge;
- download OSGi example project

A.2 Begin Introspection

- fire some commands at project and bundle level
- OSGi Scan
- generate some reports