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
November 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

Todays 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 its well known that higher quality softwares are easier to maintain and evolve at long term.

The Open Services Gateway Initiative(OSGi) is the *de facto* standard for building Java modular applications but there is no automated way to measure the quality of OSGi systems. In the context of Java applications there are many well known quality metrics and tools to measure application's quality but when we move to Java modular applications where standard quality metrics does not fit or even exist, for example module dependency metrics, we run out of options.

In this work will be presented a tool called *Intrabundle* that analyses OSGi projects and measure their quality. Its 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 dividas em partes manores que colaboram entre si para resolver problemas maiores, o famoso *dividir para conquistar*. Outro aspecto importante na industria de software que ajuda à construir aplicações grandes é o conceito de qualidade de software já que é sabido que quanto maior a qualidade do software mais facil de mante-lo e evolui-lo a logo prazo será.

The Open Services Gateway Initiative(OSGi) é o *padrão de fato* para se criar aplicações modulares em java porém não existe forma automatizada de se medir a qualidade de sistemas OSGi. No ambito de aplicações java existem diversas metricas de qualidade e ferramentas para medir a qualidade de software mas quando entramos no contexto de aplicações modulares, onde as métricas conhecidas não se encaixam ou não existem, por exemplo dependência entre módulos, ficamos sem opções.

Neste trabalho será apresentada uma ferramenta chamada *Intrabundle* que analisa projetos OSGi a mede sua qualidade. Ainda seram 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

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
ABNT	Associação Brasileira de Normas Técnicas
CISQ	Consortium for IT Software Quality

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 functional or non-functional where the first focuses on how the software meets its specification and how it works accordingly to its requirements and the second is aimed on how well the software is structured, we can generalize the first as being *external quality* and second as *internal quality*. To measure external quality there is the need to execute the software, also known as *dynamic analysis*, either by an end user accessing the system or an automated process like for example functional testing or performance testing. There is no known way to assure functional quality without executing the software. Internal quality however can be verified by either *statical 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* which is the detailed investigation of internal logic and structure of the code (KHAN et al., 2012).

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 leads up to 100 times reduced costs (BEOHM et al., 2001) and also other system characteristics like reusability, reliability and portability are benefited by higher software quality.

A well known and successful way to structure software architecture is to modularize its components. In the Java ecosystem although there is a moving to modularize the JDK and Java applications with the project Jigsaw (KRILL, P.) and also the recent *microservices* movement (KNORR, E.) for now the only practical working and well known solution for modular Java applications is OSGi, a very popular component-based and service-oriented framework for building Java modular applications which is the *de facto* standard solution for this kind of software since early 2000's and have being used as basis of most JavaEE application servers, the open source IDE Eclipse, Atlassian Jira and Confluence to cite a few big players using OSGi.

In the context of Java modular applications using OSGi and software quality there is no way to measure software internal quality which is the main objective of this work.

1.2 Objectives

This work is focused on *internal* OSGi projects quality mainly due to the following facts:

1. There is no known standard way neither tools to measure OSGi projects internal quality (**?**).

2. We already have tools and approaches to measure standard projects internal and external quality.
3. For OSGi applications measuring *external quality* the classical approaches like automated testing are sufficient and widely used.

For measuring OSGi qualities first will be created the metrics based on good practices in the development of OSGi systems so in a second moment those metrics will be applied on top of real OSGi projects using a tool called *Intrabundle* which was created during this work and also will be presented here. In the end the resulting output of Intrabundle and introspected projects qualities will be analyzed to conclude if created metrics have value for measuring Java modular applications or not.

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 will introduce the main concepts and technologies used in this work and will be divided into two main sections where the first will be focused in the area of software quality like quality measurement, quality metrics, program analysis and quality analysis tools and the second section of chapter two will present Java and OSGi, how standard Java and OSGi are different in respect to quality metrics and why we need different metrics for OSGi. The fourth chapter presents Intrabundle, an OSGi code introspection tool to measure internal quality, we will see how Intrabundle works, what kind of information it extracts and what metrics it is applying. The fifth chapter will analyze the results Intrabundle produces and validate them to decide if this work has a valid contribution or not. The last chapter will present the conclusions and future work on this subject.

2 STATE OF ART

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 as well the motivation behind this solution and why standard quality metrics aren't sufficient for this kind of application.

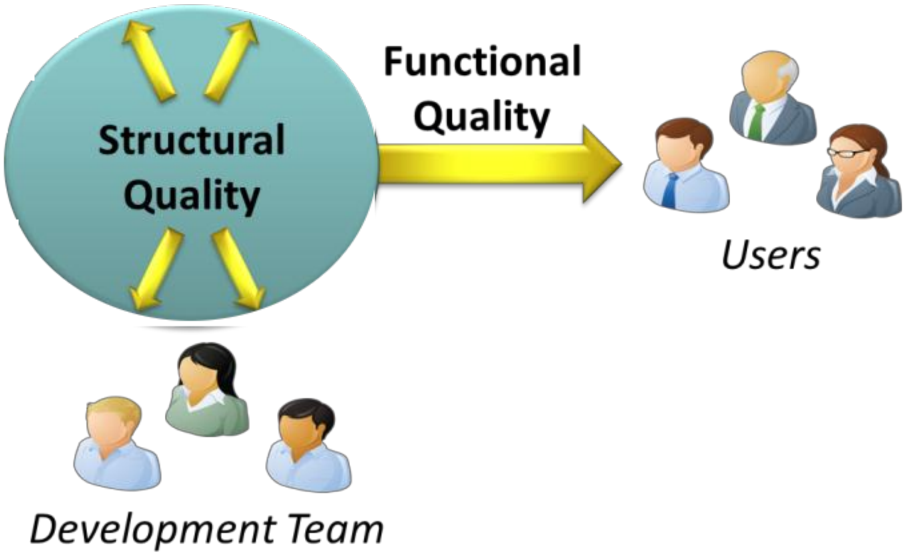
2.1 Software Quality

There has been many definitions of software quality(TODO REF - Metrics and Models in Software Quality Engineering) 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 requirements[1].

Another important aspect of software quality is that it can be divided in two groups, the **external** and **internal** 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. 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.

[1]Functional 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



2.1.1 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 coverage[2].

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

[2]A 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)

2.1.2 Software Metric

A software metric is the measurement of a software attribute which in turn is a quantitative calculation of a characteristic.

2.1.2.1 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 programing 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)

2.1.3 Program Analysis

Program analysis sis 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.

2.1.3.1 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.

2.1.3.2 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*[3], identify arithmetic errors and extract code coverage.

[3]Resources that are hold on system's memory and aren't released

2.1.4 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 PMD 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:

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.ufrgs.rmpestano.intrabundle.metric.DefaultMetricsCalculator:119 Rule:EmptyCatchBlock Priority:2 Must handle exceptions.
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.model.ManifestMetadata:143 Rule:StringInstantiation Priority:2 Avoid instantiating String objects; this is u
sually unnecessary..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.model.ManifestMetadata:160 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf
(String)..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.model.ManifestMetadata:162 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf
(String)..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.model.ManifestMetadata:201 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf
(String)..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.model.ManifestMetadata:255 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf
(String)..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.model.ManifestMetadata:309 Rule:UseIndexOfChar Priority:3 String.indexOf(char) is faster than String.indexOf
(String)..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.model.OSGiModuleImpl:186 Rule:EmptyCatchBlock Priority:2 Must handle exceptions.
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.model.OSGiProjectImpl:37 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'projectMetr
ic'..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.model.OSGiProjectImpl:38 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'metrics'..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.plugin.BundlePlugin:22 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'prompt'..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.plugin.LocalePlugin:34 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'event'..
[INFO] PMD Failure: br.ufrgs.rmpestano.intrabundle.plugin.OSGiScanPlugin:39 Rule:UnusedPrivateField Priority:3 Avoid unused private fields such as 'moduleLevel
s'..
[INFO] PMD Failure: br.ufrgs.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: Intrabunde PMD ruleset

```xml
<?xml version="1.0"?>
<ruleset name="Custom ruleset" xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0">
    <description>
        This ruleset checks my code for bad stuff
    </description>

    <exclude-pattern>.*/src/test/.*</exclude-pattern>

    <!-- Here's some rules we'll specify one at a time -->
    <rule ref="rulesets/java/unusedcode.xml/UnusedLocalVariable" />
    <rule ref="rulesets/java/unusedcode.xml/UnusedPrivateField" />
    <rule ref="rulesets/java/imports.xml/DuplicateImports" />
    <rule ref="rulesets/java/basic.xml/UnnecessaryConversionTemporary" />

    <rule ref="rulesets/java/strings.xml">
        <exclude name="AvoidDuplicateLiterals" />
        <exclude name="AppendCharacterWithChar" />
        <exclude name="ConsecutiveLiteralAppends" />
        <exclude name="InefficientStringBuffering" />
    </rule>

    <!-- We want to customize this rule a bit, change the message and raise
        the priority -->
    <rule ref="rulesets/java/basic.xml/EmptyCatchBlock" message="Must handle exceptions">
        <priority>2</priority>
    </rule>

    <!-- Now we'll customize a rule's property value -->
    <rule ref="rulesets/java/codesize.xml/CyclomaticComplexity">
```

Source: intrabundle ruleset (2014)

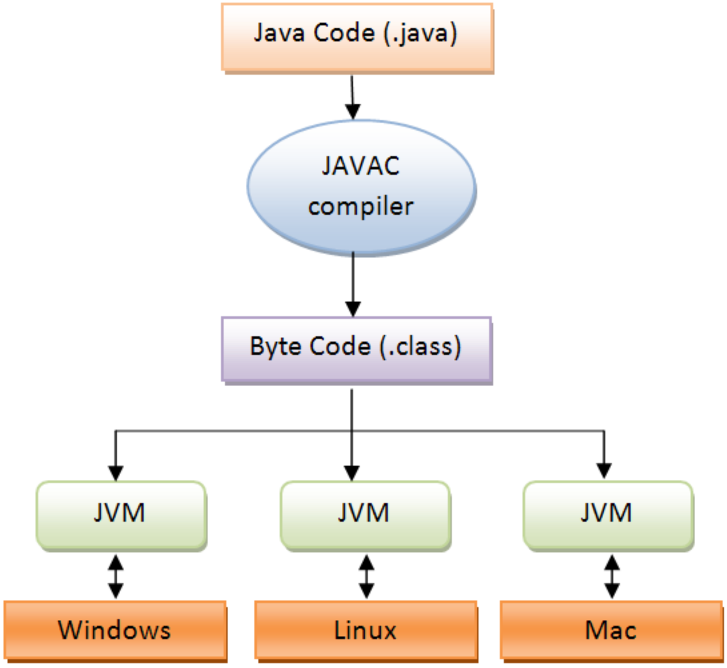
2.2 Java and OSGi

In the context of Java™ programming language (ARN05, 2005), which accordingly to IEEE spectrum of this year is the most popular programming language (IEEE Spectrum, 2014), and modular applications this section will introduce the Java language and OSGi framework.

2.2.1 The Java language

Java is a general purpose object oriented[4] 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 a abstract computing machine[5] and platform-independent execution environment that execute Java byte code[6] and converts it 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 safe[7]
- Dynamic: during the execution of a program, Java can dynamically load classes
- Strong memory management(no explicit pointer)
- Automatic garbage collection to release unused objects from memory
- Robust: extensive compile-time checking so bugs can be found early
- Multithreaded[8]
- Distributed: networking capability is inherently integrated into Java

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

[5]Also known as *Virtual Machine* which is an emulation of a particular computer system

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

[7]Type safety is the extent to which a programming language discourages or prevents type errors

[8]Multithreading is a program's capability to perform several tasks simultaneously

2.2.2 The OSGi framework

3 INTRABUNDLE - AN OSGI BUNDLE INTROSPECTION TOOL

3.1 Introduction

3.2 Design Decisions

To analyze large code bases of OSGi projects which can vary from KLOCs to thousands of KLOCs we needed a lightweight approach with the following functional requirements:

-

The following alternatives were evaluated:

-

3.3 JBoss Forge

3.4 Implementation Overview

3.5 Collecting Bundle Data

3.6 Metrics Calculation

3.7 Intrabundle Quality

In this section we will see how intrabundle quality is managed and how some concepts of *chapter 2 - State of art* were applied to the project.

3.7.1 Internal quality

Intrabundle internal is managed by PMD and JaCoCo. PMD is an static analysis tool and JaCoCo a dynamic analysis one. Both were presented at Chapter two in section *Quality Analysis Tools* with the objective to guarantee non functional requirements.

3.7.1.1 Example

A 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 3.1 shows JaCoCo code coverage report for Intrabundle:

Figure 3.1: Intrabundle code coverage

intrabundle

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
br.ufrgs.rmpestano.intrabundle.factory		100%		83%	1	6	0	6	0	3	0	1
br.ufrgs.rmpestano.intrabundle.event		100%		n/a	0	3	0	5	0	3	0	2
br.ufrgs.rmpestano.intrabundle.locator		97%		70%	3	11	2	34	0	6	0	2
br.ufrgs.rmpestano.intrabundle.jdt		95%		88%	2	11	1	18	1	7	0	2
br.ufrgs.rmpestano.intrabundle.plugin		89%		80%	27	114	20	251	4	54	0	4
br.ufrgs.rmpestano.intrabundle.facet		89%		73%	15	38	7	40	3	16	0	4
br.ufrgs.rmpestano.intrabundle.model		81%		61%	138	289	101	497	11	89	0	7
br.ufrgs.rmpestano.intrabundle.metric		78%		64%	15	40	21	100	0	13	0	2
br.ufrgs.rmpestano.intrabundle.util		76%		64%	67	140	52	197	4	34	0	7
br.ufrgs.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

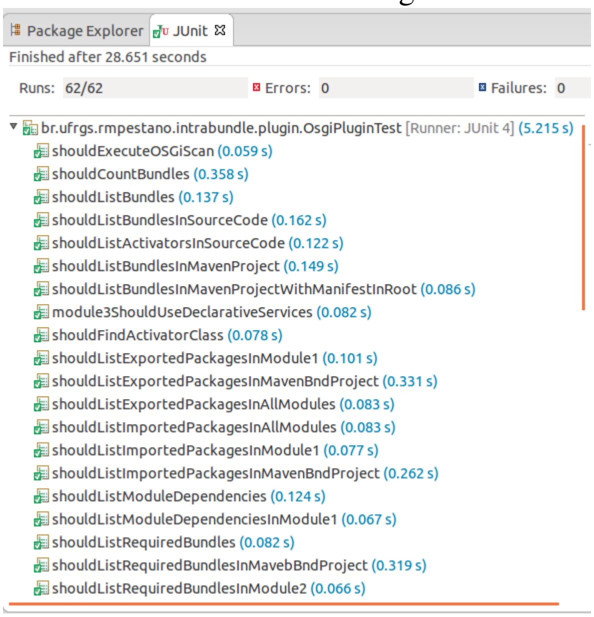
3.7.2 External quality

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.

3.7.2.1 Example

Intrabundle performs 62 **integration tests**, which can be defined as automated tests aimed to detect any inconsistencies between the software units that are integrated together, to guarantee its external quality. 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 (**?**), an integration test platform. Figure 3.2 shows the result of integration tests execution:

Figure 3.2: Intrabundle external tests



4 BUNDLE INTROSPECTION RESULTS

This chapter will make a deep analysis of results and prove that my contribution is valid(or not)

5 CONCLUSION

REFERENCES

BEOHM, B.; BASILI, V. R. Software Defect Reduction Top 10 List. **Computer**, Los Angeles, v. 34, no. 1, pp 135-137, January 2001.

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.

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 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 2014.

HAMZA, S.; SADOU, S.; FLEURQUIN, R. Measuring Qualities for OSGi Component-Based Applications. **International Conference on Quality Software**, Najing, pp 25-34, July 2013.

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 2014.

SQA. Software Quality Metrics. [S.l.:s.n]. Available at:<`http://www.sqa.net/softwarequalitymetrics.html`>. Accessed in: November 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 2014.

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.

Intrabundle PMD ruleset. Available at:<https://github.com/rmpestano/intrabundle/tree/master/src/test/resources/rulesets/pmd.xml>. Accessed in: November 2014.

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 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 2014.

K. Arnold, J. Gosling and D. Holmes.
THE Java™ Programming Language. 4th Edition. [S.l]:Addison Wesley Professional, 2005