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

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 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 industria 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 facil de mante-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 dificil medir a qualidade interna de sistemas OSGi devido a suas caracteristicas 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

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CISQ	Consortium for IT Software Quality
GUI	Graphic User Interface
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
IDE	Integrated Development Environment
JVM	Java Virtual Machine
LOC	Lines of Code
KLOC	Kilo Lines of Code
MQP	Maximum Quality Points
QP	Quality Points
TQP	Total Quality Points

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 software[1] 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 *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*[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 JavaEE[3] application servers[4], 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

[1] Also known as dynamic analysis

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

[3] A 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

[4] Java 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 runtime[1] for Intrabundle[2].

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 requirements[3].

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

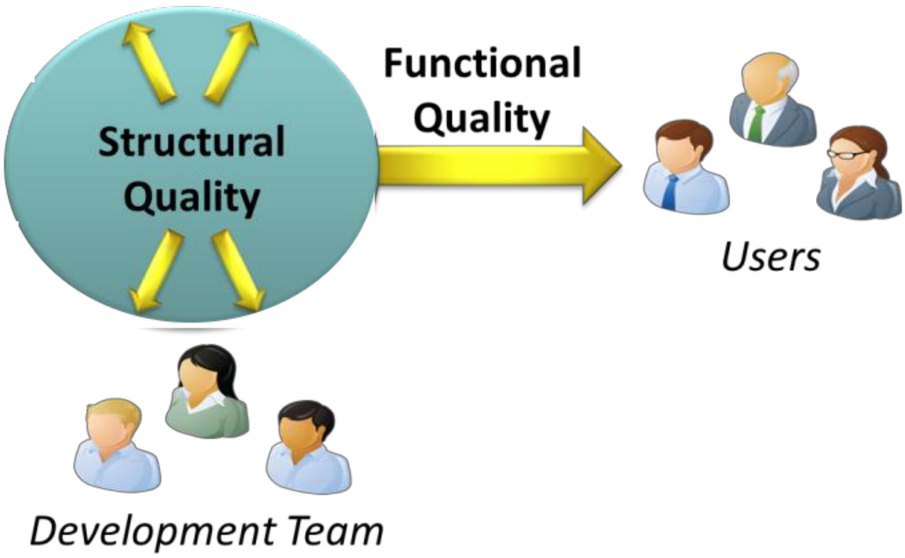
The opposite is internal or structural quality that aims to how the software is architected internally which is the perspective of the programmer and non functional requirements. So the higher internal quality the better the code is structured, efficient, robust and maintainable it should be. Image 2.1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

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

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

[3] 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



2.1.3 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 coverage[4].

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

[4] 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)

2.1.4 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 metrics[5], process metrics[6], and project metrics[7]. 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

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

[6]Process 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

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

attribute is, for example using latency[8] to calculate response time[9] 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 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)

2.1.5 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.

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

[9]The 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:

[10]Resources 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.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)

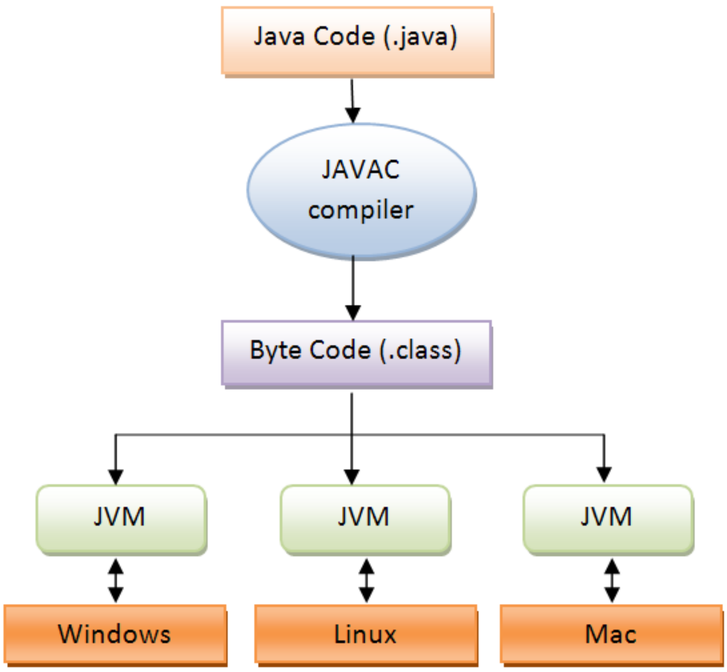
2.2 Java and OSGi

In the context of Java™ programming language (Arnold et al., 2005), which accordingly to IEEE spectrum of this year is the most popular programming language (IEEE Spectrum, 2014), and modular applications[11] this section will introduce the Java language and OSGi framework.

2.2.1 The Java language

Java is a general purpose object oriented[12] 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 machine[13] and platform-independent execution environment that execute Java byte code[14]. 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 safe[15]
- Dynamic: during the execution of a program, Java can dynamically load classes
- Strong memory management(no explicit pointer)

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

[12] 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

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

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

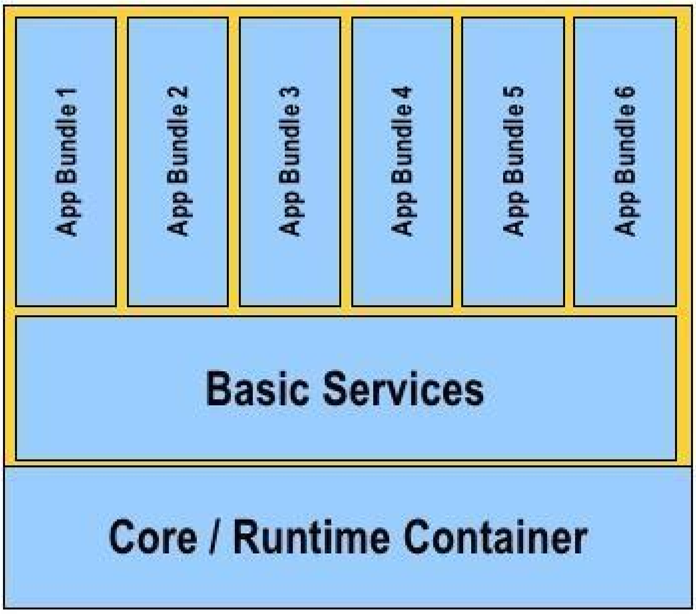
[15] Type safety is the extent to which a programming language discourages or prevents type errors

- Automatic garbage collection to release unused objects from memory
- Robust: extensive compile-time checking so bugs can be found early
- Multithreaded[16]
- 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 implementations[18]. 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 bundle[19] is a group of Java classes and resources packed as .jar extension with additional metadata in manifest MANIFEST.MF file

[16] Multithreading is a program's capability to perform several tasks simultaneously
[17] A non profit worldwide consortium of technology innovators
[18] Apache Felix, Eclipse Equinox, Knopflerfish and ProSyst
[19] Also 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**.

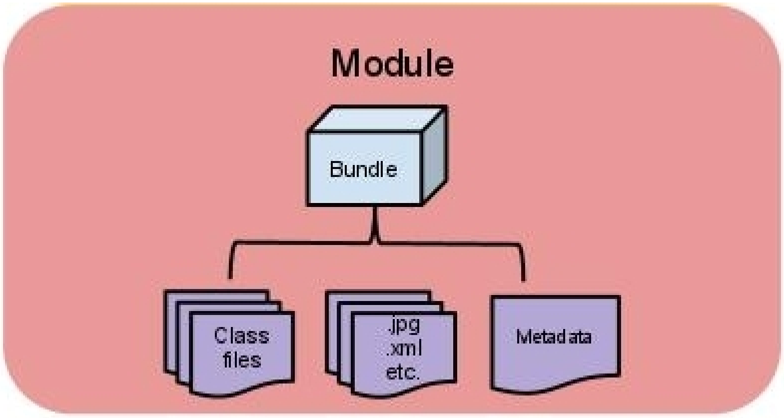
With OSGi, you modularize applications into bundles. Each bundle is a tightly coupled, dynamically loadable collection of classes packed in JARs[20], and configuration files that explicitly declare any external dependencies. All these characteristics are provided in OSGi by three conceptual layers that 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.

[20] acronym 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

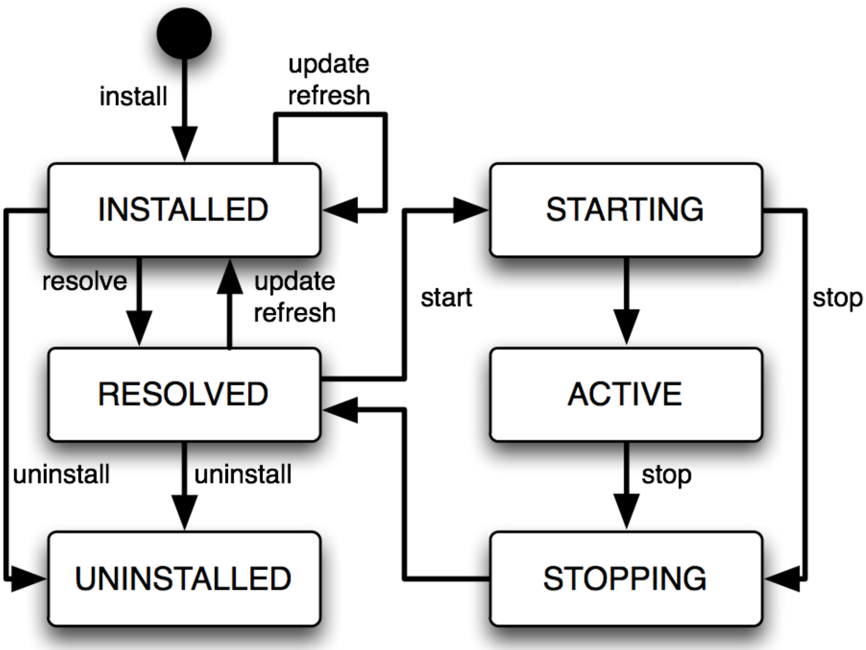


Source: OSGi conceptual layers (2011)

2.2.2.3 Lifecycle layer

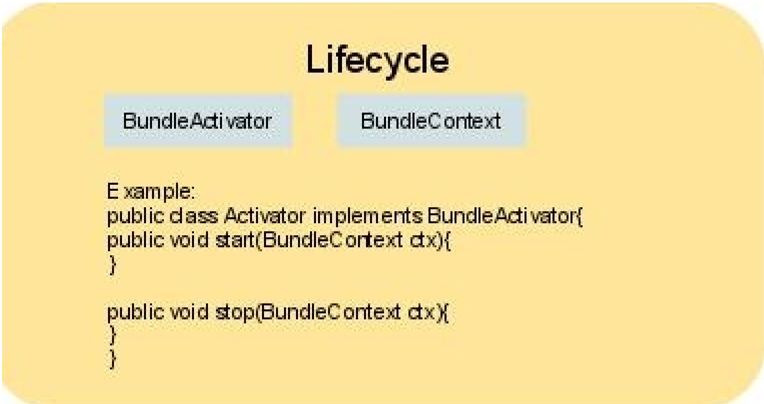
Provides access to the underlying OSGi framework through the *Bundle Context* object. This layer handles the lifecycle of individual bundles so you can manage your application dynamically, including starting and stopping bundles to manage and evolve them over time. Bundles can be dynamically installed, 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



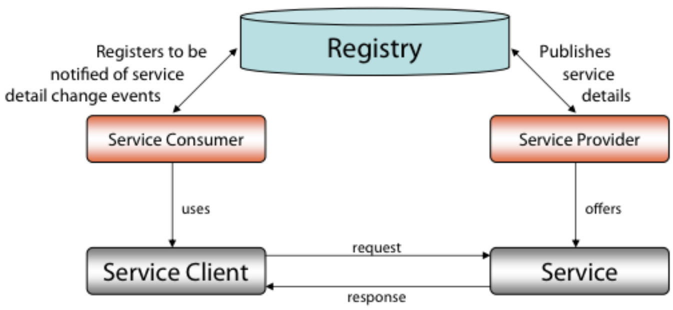
Source: OSGi conceptual layers (2011)

2.2.2.4 Service layer

This layer provides communication among modules and their contained components. Service providers publish services[21] 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



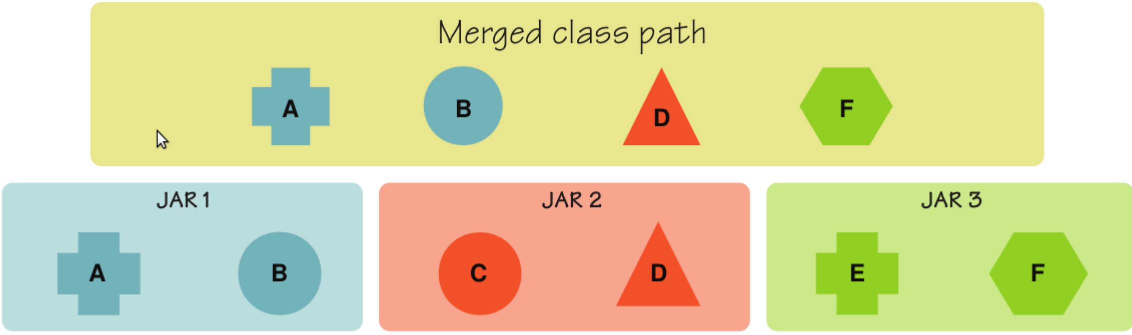
2.2.3 Vanilla Java vs OSGi

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 classes[22] are loaded by a classloader[23] from the same classpath[24] 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.

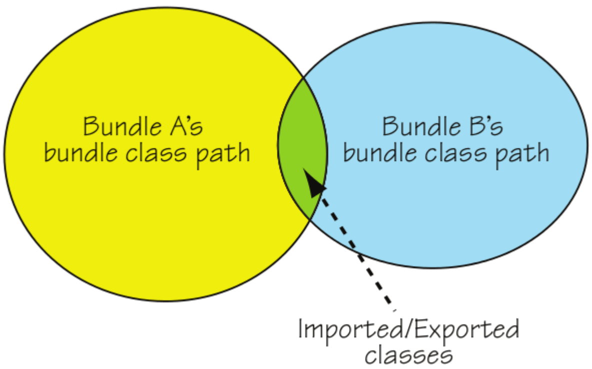
Figure 2.10: Java jar hell



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

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

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

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

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

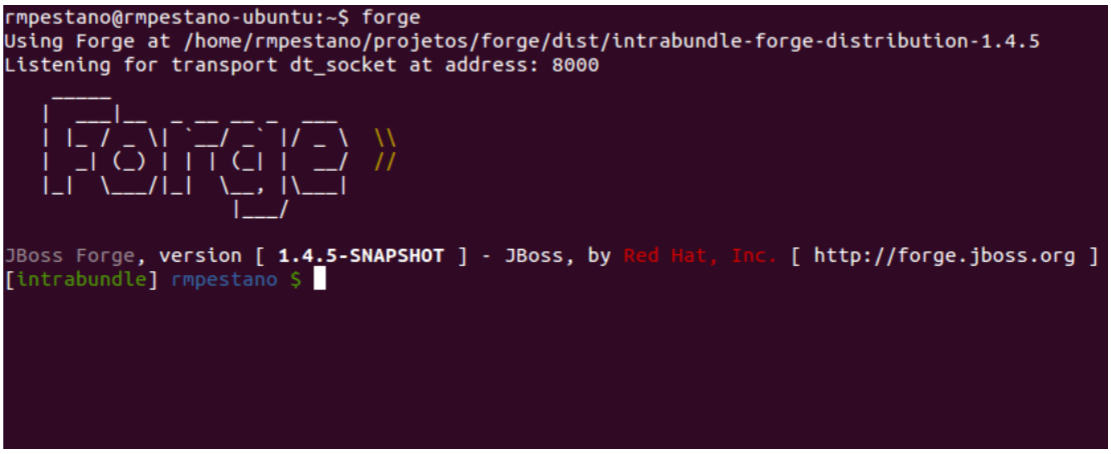
at the same time(used by different bundles). This enables independent evolution of dependent artifacts which, in the Java world, is unique to OSGi environments (semantic versioning, 2010).

2.3 JBoss Forge

2.3.1 Introduction

JBoss Forge is a modular[26] plugin based general purpose command line tool(CLI). Forge can be started through command line or be integrated in an IDE. Figure 2.12 shows Forge initial screen:

Figure 2.12: Forge initial screen



Forge runs on any operating system that can run Java and have built in startup scripts[27] for Windows, Linux and MacOS.

2.3.2 Forge Plugin

A Forge plugin can be as simple as a tool to print files to the console, or as complex as deploying an application to a server, tweeting the status of your latest source-code commit, or even sending commands to a home-automation system. The sky is the limit!.

Every plugin offers a set o commands that may be restricted by a facet.

2.3.2.1 Example

Below is a simple Forge plugin named *hello-world* with a command named *sayHello* that prints "Hello World" when executed:

[26]Forge runtime is based on JBoss modules which is a technology, like OSGi but not so popular, for building modular applications

[27]Executable files that initiate a process or system

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 interface[28] 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");
    }
}
```

[28]In 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 systems[29].

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.

[29] As forge runs on top of Java, Forge inherits its *universality*

3 DESIGNING AN OSGI BUNDLE INTROSPECTION TOOL

This chapter discusses the design of the tool created in this work. It is splited in the following sections: The first section introduces Intrabundle, the second talks about design decisions, next section will specify the data the tool is collecting, later the metrics created will be explained and finally the quality calculation will be specified.

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 the design of a tool called *Intrabundle* (intrabundle github, 2014), an open source Java based application created in the context of this work. Intrabundle introspects OSGi projects thought static code analysis. It collects useful information from OSGi projects and later calculates its **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 Maven[1], Eclipse projects and BND [2];
- 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 to analyze lots of projects through its interface;
- Should output a detailed quality report so the extracted information can be analyzed.

and the following *non functional requirements*:

- Only open sourced projects[3] 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;

[1] Maven is a build tool for Java

[2] BND is a tool to easy OSGi projects development and bundle management and configuration

[3] Projects 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

- Fast;
- Use Java to leverage the author's experience in the language;
- Use a good file system API[4] because file manipulation is one of the most frequent tasks the tool should perform.

The following tools were evaluated to implement the tool:

1. Build a standalone Java client application using javaFX[5];
2. Create an Eclipse plugin[6];
3. Create a Maven plugin[7];
4. Build the tool on top of JBoss Forge;
5. Build a Java project on top of OSGi platform;
6. 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 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 CDI[8] so managing Objects lifecycle and events 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 framework inside it;
- Is modular, each plugin has its own classpath;
- 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

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

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

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

[7]Maven 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.

[8]Context 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 case CDI

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.

Although an OSGi based tool would be benefited by modularity and service oriented architecture it would have the same limitations of a standalone JavaFX application and also the author's experience with OSGi projects is not as advanced as in Forge environment.

PMD[9] has a very limited API so it could be hard to generate reports or analyze multiple projects using it.

3.3 Identifying OSGi Projects and Bundles

To collect data and calculate quality of project we first need to identify those projects. In the case of OSGi projects the tool must be capable of find OSGi *projects* and its *bundles*, the module itself. 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.

There are many formats of OSGi projects and each one may require a different algorithm to be identified. In this work we will be concerned with the following types of OSGi projects:

- Standard Maven projects[10];
- Maven projects using BND tools[11];
- Standard BND Tools project[12];
- Standard Eclipse Java projects[13];
- Package based bundles[14].

3.4 Collecting Bundle Information

After identifying OSGi bundles and OSGi projects Intrabundle needs to extract useful information from them. Table 3.1 shows which attributes the tool must collect from OSGi projects:

[9]A 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

[10]Each project is a bundle and its meta data is in maven resources folder

[11]Each project is a bundle and bundle meta data is in pom.xml configuration file

[12]Each project is a bundle and meta data is in bnd file

[13]Each project is a bundle and its meta data is in META-INF folder

[14]In this kind of OSGi projects each package is a bundle and meta data is in the same package

Table 3.1: Extracted data from OSGi projects

Name	Description
Loc	Lines of code.
Declarative services	Verifies if bundles uses declarative services[15].
Ipojo	Verify if bundles uses Ipojo[16]
Blueprint	Verify if bundles uses Blueprint[17]
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.

Here is the justification of each attribute:

- **LoC** is being extracted because it is 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.
- **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.
- **Stale Services References** refers to code that may retain OSGi service references from being collected by Java garbage collection[18] even when the providing bundles are gone (Gama and Donsez, 2012).
- **Bundle dependencies** and **Required bundles** are closed related to coupling[19] between bundles. The less bundles a bundle depends the better it will be to maintain it as changes to other components will not affect it.
- **Publishes Interface** verifies if a bundle is exposing only its API and hide the implementation details from consumers. It is a good practice having functionalities that are independent of their respective implementations, this allows definitions and implementations to vary without compromising each other.
- **Declares permission** is a security software attribute in OSGi projects and may restrict access to bundles.

[18]Is the process of looking at memory, identifying which objects are in use and which are not, and deleting the unused objects

[19]which is a measure of how closely connected two software components are

- Number of classes, interfaces and abstract classes are being collected to support the calculation of other attributes.

3.5 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 and how bundle and project quality is calculated based on the metrics.

3.5.1 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.

3.5.2 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[20]. To classify LoC metric we use the following rule:

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

Second metric is **Publishes interfaces** meaning that bundles should hide its 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).

[20]If your algorithm is trying solve a very complex problem then it probably will have lots of lines of code and not necessarily have a low cohesion

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 is. It is considered a base pattern called *Manage Relationships* in (Knoernschild et al., 2012). Here is how this metric is calculated by Intrabundle:

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

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 applications: *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 calculates this metric by counting specific method 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):

$NC = \sum_{i=1}^{n}$ where **n** = number of classes a bundle have.
$NS = \sum_{i=1}^{n}$ where **n** = number of stale references found.

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

In other words if no stale references are 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 then the metric is considered state of art:

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

3.5.3 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* adds **5QP**, *Very good* **4QP**, *Good* **3QP**, *Regular* **2QP** and *Anti pattern* **1QP**.

3.5.4 Bundle Quality

Bundle final quality is calculated as a function of *Total Quality Points* **TQP**, which is the total points obtained in all created metric, and *Maximum Quality Points* **MQP** that is the maximum points a bundle can have. MQP is equal to all metrics classified as State of art. Here is the formula:

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

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

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

if 1 $<= f(q) > 0.9$ then State of Art;
if $0.9 <= f(q) > 0.75$ then Very Good;
if $0.75 <= f(q) > 0.6$ then Good;
if $0.6 <= f(q) > 0.4$ Regular;
if $0.4 <= f(q)$ 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)in one metric 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.

3.5.5 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 bundle's 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*:

$MQP = \sum_{i=1}^{n} 5$ where **n** = 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**.

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

if 1 $<= f(q) >$ 0.9 then State of Art;
if 0.9 $<= f(q) >$ 0.75 then Very Good;
if 0.75 $<= f(q) >$ 0.6 then Good;
if 0.6 $<= f(q) >$ 0.4 Regular;
if 0.4 $<= f(q)$ 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.

3.5.6 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 defined as percentage of points obtained from maximum points:

$MQP = \sum_{i=1}^{n} 5$ where **n** = 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 $<= f(q) >$ 0.9 then State of Art;
if 0.9 $<= f(q) >$ 0.75 then Very Good;
if 0.75 $<= f(q) >$ 0.6 then Good;
if 0.6 $<= f(q) >$ 0.4 Regular;
if 0.4 $<= f(q)$ 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.

4 IMPLEMENTING AN OSGI BUNDLE INTROSPECTION TOOL

This chapter describes how Intrabundle was implemented and architecture overview is presented. Objects and classes that composes it will be detailed. First section gives a general overview of Intrabundle's components, second section explains how OSGi bundles and projects are identified by the tool, next section shows how useful information is being collected and how this information is gathered by reports. Last section gives an overview of how Intrabundle's quality is being maintained.

4.1 Implementation Overview

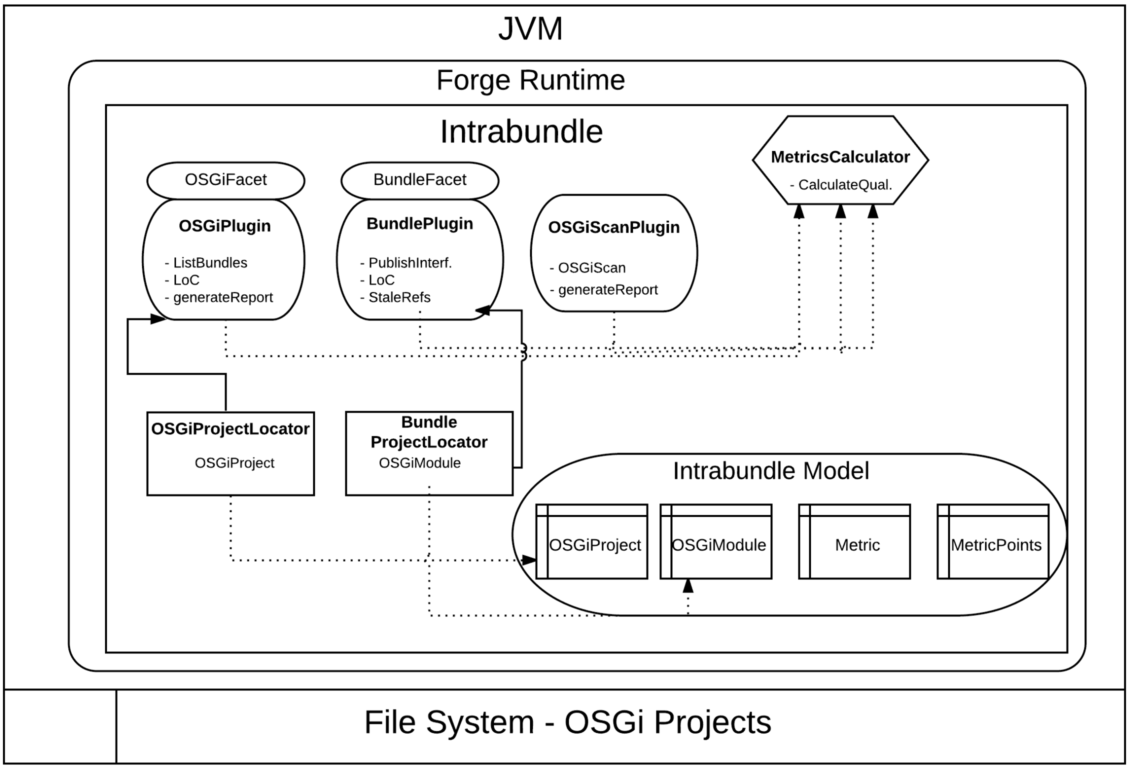
Intrabundle is composed by 3 Forge plugins, see section 2.3.2 for details about Forge plugins. The first is *BundlePlugin* which extracts OSGi bundle information, second is *OSGiPlugin* that has a vision of all bundles composed by the project. Third is *OSGiScan* a plugin responsible for scanning OSGi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

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 that is 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.

Another component in the architecture is *MetricsCalculator* that calculates bundle and OSGi project quality based on data contained on OSGiProject and OSGiModule objects. To calculate projects qualities Intrabundle creates the *Metric* and *MetricPoints* objects. MetricPoints has a list of Metrics and the quality is calculated in MetricPoints object based on all metrics it has. The final quality is represented by a Java object called *MetricScore* which holds the *quality label* presented in 3.5.1. Figure 4.1 gives an overview of Intrabundle architecture:

Figure 4.1: Intrabundle architecture



4.2 Bundle and Project Identification

Intrabundle implements its *facets* and *locators* to identify OSGi bundles and OSGi projects. To do that the tool searches for OSGi meta data in *MANIFEST* file[1]. 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 4.1 lists the types of OSGi projects and the location of Manifest file:

Table 4.1: Supported types of OSGi projects

Type	Manifest location
Maven projects	/src/main/resource/META-INF.
Maven using BND tools	pom.xml[2] with maven-bundle-plugin.
Standard Eclipse Java projects	/META-INF
Standard BND Tools	bnd.bnd file in any subfolder.
Package based bundles	each package has a manifest.

[1]The 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.

4.3 Retrieving bundle information

Section 3.4 described which information the tool extracts. Now its presented how that is done.

LoC is a classical software metric that was adapted in this work to OSGi Bundles and its calculation is straight forward. The tool just sum the bundle .java files lines of code. It is important to note that comments are excluded from this calculation. **IPojo**, **Blueprint** and **Declarative Services** are extracted by looking for specific file configurations(xml files) or annotations that each technology uses. **Stale Services references** are detected via approximation, Intrabundle counts the number of services *gets* and *ungets*[3] 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 publishes interfaces. **Declares permission** verifies if bundle implements security by contract searching for *permission.perm* file inide OSGI-INF bundle directory.

Each information retrieved by Intrabundle is usually mapped to a Forge command, see Listing 4.1 which is the command that prints bundle exported packages, an information used to calculate bundle dependency and publish interfaces metric, to the Forge console:

[3] Operations that consume and release a service reference respectively

Listing 4.1: Exported packages command

```
@Command(value = "exportedPackages",help = "list bundle exportedpackages
    ")
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 of type *OSGiModule*[4], that is an immutable object[5], in method *getExportedPackages*. All information described in table 3.1, except bundle dependency, is calculated inside OSGiModule object. Bundle dependency is Calculated by OSGiProject because it has all modules and can calculate its dependencies. Above is OSGiModule Java interface describing all operations it provides:

[4]The bundle variable is created by Bundle Locator, a Forge locator, when user navigates to a directory which is an OSGi bundle, as explained in section 2.3.4.

[5]Is 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)

Listing 4.2: Intrabundle OSGiModule interface

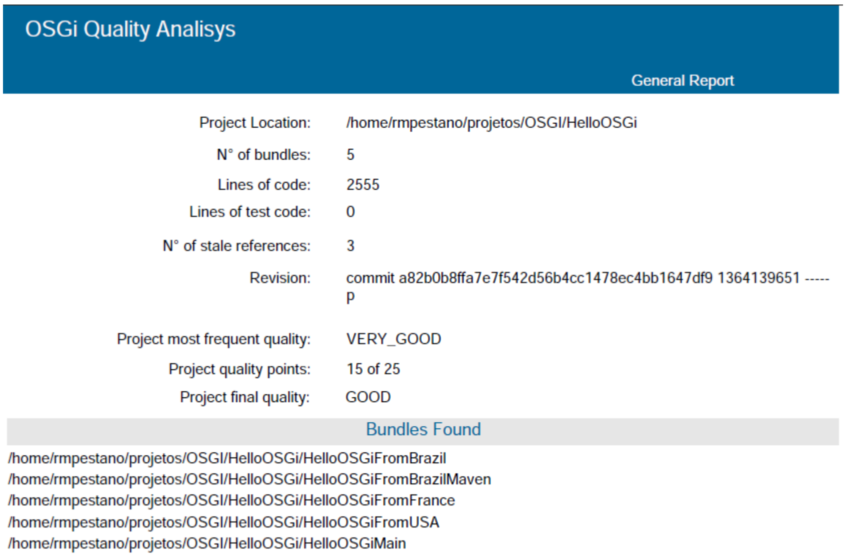
```
public interface OSGiModule extends Serializable, Comparable<OSGiModule>{

    /**
     *
     * @return total .java files(under src or src/main/java) lines of
        code
     */
    Long getLinesOfCode();


    /**
     * @return <code>true</code> if bundle uses declarative services
        specification
     * <code>false</code> if it doesnt
     */
    Boolean getUsesDeclarativeServices();

    /**
     * @return <code>true</code> if bundle uses Blueprint specification
     * <code>false</code> if it doesnt
     */
    Boolean getUsesBlueprint();

    /**
     *
     * @return object representing bundle MANIFEST.MF  or .bnd or pom.xml
         with maven-bundle-plugin
     */
    ManifestMetadata getManifestMetadata();

    /**
     *
     * @return bundle activator java file
     */
    FileResource<?> getActivator();

    /**
     *
     * @return bundle imported packages
     */
    List<String> getImportedPackages();

    /**
     *
     * @return bundle exported packages
```

```
     */
    List<String> getExportedPackages();

    /**
     *
     * @return bundle required bundles
     */
    List<String> getRequiredBundles();

    /**
     * @return <code>true</code> if bundle exported packages contains
     *     only interfaces
     * <code>false</code> if it has one or more classes
     */
    Boolean getPublishesInterfaces();

    /**
     *
     * @return <code>true</code> if bundle declares permissions
     * <code>false</code> otherwise
     */
    Boolean getDeclaresPermissions();

    /**
     *
     * @return .java files possibly containing OSGi service stale
     *     references
     */
    List<Resource<?>> getStaleReferences();
}
```

4.4 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 4.2 shows an example report:

Figure 4.2: Intrabundle general report



OSGi Quality Analisys

General Report

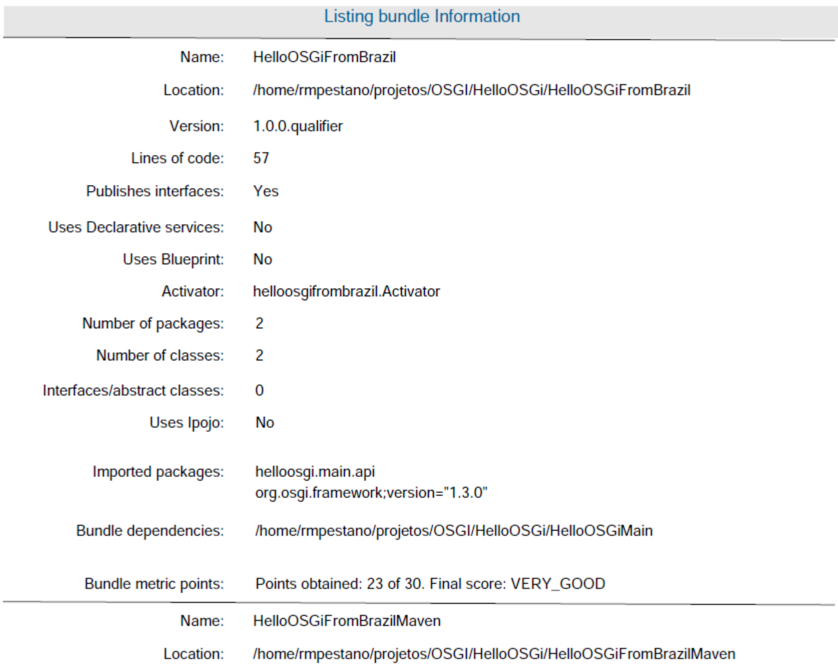
Project Location:	/home/rmpestano/projetos/OSGI/HelloOSGi
N° of bundles:	5
Lines of code:	2555
Lines of test code:	0
N° of stale references:	3
Revision:	commit a82b0b8ffa7e7f542d56b4cc1478ec4bb1647df9 1364139651 ----- p
Project most frequent quality:	VERY_GOOD
Project quality points:	15 of 25
Project 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 bundle, see Figure 4.3

Figure 4.3: 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 Ipojo:	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 4.4:

Figure 4.4: 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 4.5

Figure 4.5: 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).

4.5 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.

4.5.1 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.

4.5.1.1 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 4.6 shows JaCoCo code coverage report for Intrabundle:

Figure 4.6: 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

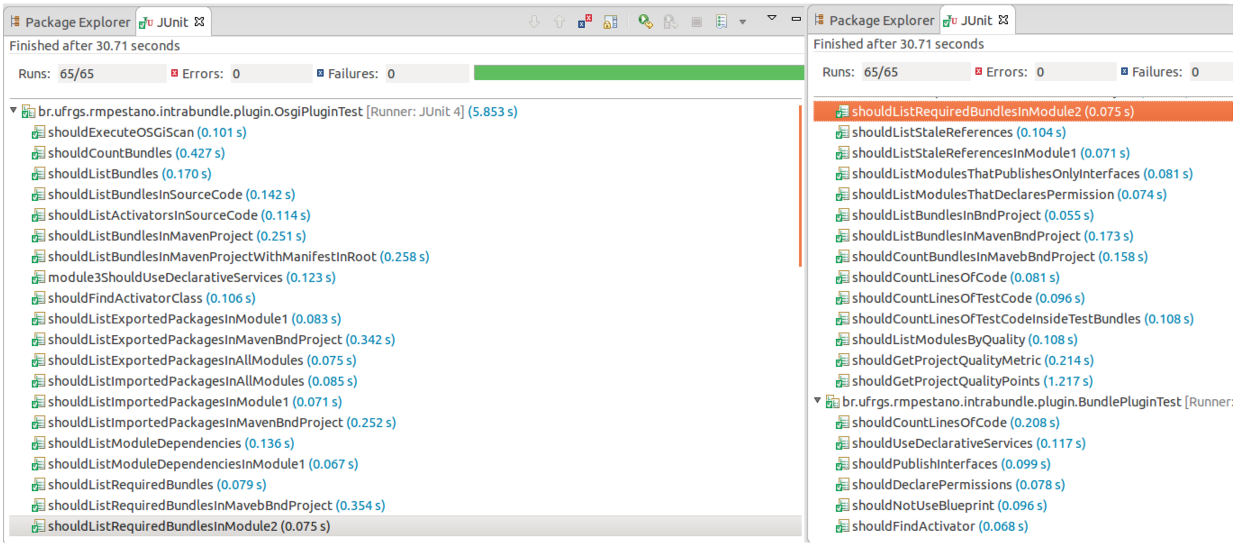
4.5.2 External quality

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.

4.5.2.1 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 during tests. That is done by Arquillian (dan, 2011), an integration testing platform. The tests are also executed online on each commit[6] by Travisci[7], a technique called *continuous integration*. Figure 4.7 shows the result of integration tests execution:

Figure 4.7: Intrabundle integration tests



4.6 Validation

In order to validate our implementation and if proposed metrics make sense we will generate Intrabundle reports on top 10 real OSGi projects. These reports will be analyzed and we will try to infer useful information and tendencies from them. The reports must gather information that make it possible to compare and confront data in the most variable scenarios.

[6]A command that pushes software changes to version control
[7]An online continuous integration server

5 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.

5.1 Analyzed Projects

In this section is presented an overview of projects that were analyzed during this work. Table 5.1 shows projects in terms of *size*. We've chosen projects that vary in size, are from different organizations (Apache, Eclipse, etc), they solve different problems and are all open source.

Table 5.1: OSGi projects analyzed by Intrabundle

Name	N° of bundles	LoC	LoC/bundle	Analisys time(sec)
BIRT	129 (217)	2,226,436	17,259	73.9
Dali	35 (46)	1,058,160	30,233	56.9
Jitsi	155 (158)	607,144	3,917	29.6
JOnAS	117 (122)	366,940	3,136	27.5
Karaf	58 (60)	93,743	1,616	8.6
Openhab	181 (184)	347,492	1,919	23.2
OSEE	183 (190)	873,690	4774	10.3
Pax CDI	21 (22)	19,480	927	2.5
Tuscany Sca	138 (140)	243,494	1,764	23.2
Virgo	36 (49)	77,859	2,162	5.6
Sum	1051	4,962,094	4721	261.3(~4min)

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 excluding comment lines. Analysis time column is the time in seconds to extract data and generate reports using Intrabundle osgi-scan 12 command on the root directory of each projects. The environment the reports were generated was: Sony Vaio laptop eg series, Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz, 4GB RAM, OS Ubuntu 12.04, Java version "1.7.0_67".

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**: An 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 system 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.

5.2 Projects Quality Results

In this section will be presented the resulting qualities of analyzed projects and some comparisons. First comparison groups 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.

5.2.1 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 3.5.5 for further information:

Table 5.2: 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 ppen and free, but the participation is open as well.

5.2.2 Metrics Qualities 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 section 3.5.6 for more details. Values in table 5.3 are the *total quality points* (TQP) obtained. Values in parenthesis are the percentage of MQP of table values and after the parenthesis is the quality label that the percentage represents (as described in 3.5.6):

Table 5.3: Projects quality by metrics

Name	MQP	LoC	Publishes interfaces	Uses framework	Bundle dependency	Stale references	Declares permission
BIRT	645	294 (45.6%) regular	393 (60.9%) good	258 (40%) regular	307 (47.6%) regular	644 (99.8%) state of art	261 (40.5%) regular
Dali	175	74 (42.3%) regular	175 (100%) state of art	70 (40%) regular	65 (37.1%) anti pattern.	174 (99.4%) state of art	70 (40%) regular
Jitsi	775	492 (63.5%) good	775 (100%) state of art	310 (40%) regular	459 (59.2%) regular	473 (61%) good	310 (40%) regular
JOnAS	585	358 (61.2%) good	480 (82.1%) very good	252 (43.1%) regular	481 (82.2%) very good	573 (97.9%) state of art	234 (40%) regular
Karaf	290	212 (73.1%) good	257 (88.6%) very good	158 (54.5%) regular	290 (100%) state of art	278 (95.9%) state of art	116 (40%) regular
Openhab	905	672 (74.3%) good	791 (87.4%) very good	806 (89.1%) very good	664 (73.4%) good	901 (99.6%) state of art	362 (40%) regular
OSEE	915	584 (63.8%) good	909 (99.3%) state of art	573 (62.6%) good	529 (57.8%) regular	881 (96.3%) state of art	366 (40%) regular
Pax CDI	105	85 (81%) very good	105 (100%) state of art	66 (62.9%) good	103 (98.1%) state of art	98 (93.3%) state of art	42 (40%) regular
Tuscany SCA	690	472 (68.4%) good	684 (99.1%) state of art	288 (41.7%) regular	451 (65.4%) good	682 (98.8%) state of art	276 (40%) regular
Virgo	180	127 (76.1%) very good	180 (100%) state of art	78 (43.3%) regular	176 (97.8%) state of art	162 (90%) state of art	72 (40%) regular
Average		64.9% - good	91.7% - state of art	51.7% - regular	71.8% - good	93.2% - state of art	40.05% - regular

Following are 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 (>=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 5.3:

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 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* for 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 then bundles of other projects. We may infer that most bundles are high cohesive as they receive good quality label on LoC metric.

Bundle coupling seen to be good among analyzed projects as the average quality on bundle dependency was good.

5.2.3 Projects Qualities by Size

In the tables below the projects are separated by size as we believe that comparing projects with large code base with minor sized projects is unfair. It is easier to keep good practices in new projects as opposed to bigger projects where teams are usually larger, a person hardly will know every detail, multiple versions are being worked in parallel and so on. In this section we separate analyzed projects by LoC and number of bundles. Table 5.4 separates projects by bundles where small projects range from 0 to 100 bundles and large projects has more then 100 bundles.

[1] When 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

Table 5.4: Projects qualities by number of bundles

(a) Less then 100 bundles

Name	TQP	MQP	Points percent	Quality label	Nº of Bundles
Pax CDI	84	105	80%	Very Good	21
Virgo	132	180	73.3%	Good	36
Karaf	211	290	72.8%	Good	58
Dali	86	175	49.1%	Regular	35

(b) 100 or more bundles

Name	TQP	MQP	Points percent	Quality label	
Openhab	666	905	73.6%	Good	181
OSEE	596	915	65.1%	Good	183
Tuscany Sca	433	690	62.8%	Good	138
JOnAS	356	585	60.9%	Good	117
Jitsi	414	775	53.4%	Regular	155
BIRT	315	645	48.8%	Regular	129

For small projects there is no news, *Pax CDI* still winning. For larger projects **Openhab** is the new champion with a good quality in large code base. It is also interesting to note that we have three small projects among the five first positions in the quality rank.

The next table compares projects by LoC dividing them into small, medium and large sized. In this comparison small projects range from 0 to 100,000 lines of code, medium sized range from 100,001 to 500,000 LoC and large projects are the ones with more than half million lines of code:

Table 5.5: Projects qualities by number of LoC

(a) Up to 100,000 LoC

Name	TQP	MQP	Points percent	Quality label	LoC
Pax CDI	84	105	80%	Very Good	19,480
Virgo	132	180	73.3%	Good	77,859
Karaf	211	290	72.8%	Good	93,743

(b) Between 100,001 and 500,00 LoC

Name	TQP	MQP	Points percent	Quality label	LoC
Openhab	666	905	73.6%	Good	347,492
Tuscany Sca	433	690	62.8%	Good	243,494
JOnAS	356	585	60.9%	Good	366,940

(c) More then 500,00 LoC

Name	TQP	MQP	Points percent	Quality label	LoC
OSEE	596	915	65.1%	Good	873,690
Jitsi	414	775	53.4%	Regular	607,144
Dali	86	175	49.1%	Regular	1,058,160
BIRT	315	645	48.8%	Regular	2,226,436

In small category there is no surprise, **Pax CDI** is still leading. **Openhab** is the king of medium projects and **OSSE** is the winner among the larger group.

Note that the worse qualities are in larger projects as we expected. The two projects with more the one million LoC have the worst qualities measured by Intrabundle. The opposite is also valid and can be observed in small group table.

5.3 Results

In this section we summarize the results obtained by Intrabundle in previous sections. The first thing to note is that Intrabundle could analyze big projects in a small amount of time. It analyzed more than 1000 bundles and almost 5 million lines of code and generated detailed reports in a few minutes.

We noted that smaller projects are easier to keep good practices and that reflected on the resulting qualities where the smaller projects were on the top. We also noted that good quality is possible in bigger projects, as proved by *Openhab*.

Bundle cohesion seem to be good on most analyzed projects as they obtained good quality points on LoC metric.

Publishes Interfaces metric received the higher quality points and proved to be a very disseminated good practice among OSGi projects.

As opposed to Publishing only interfaces, the permission by contract is not being adopted in analyzed projects and received the worst punctuation.

Stale references metric was not well dimensioned as projects received good punctuation(State of art) although Stale references were found on most projects. This metric is close related to memory leaks and system reliability but was too relaxed in comparison to its importance.

About coupling, the bundle dependency metric showed that in most analyzed projects the bundles are low coupled. Most projects received *Good* quality label on this metric.

Although using a framework for service management is a good practice, it is not so popular in projects we used as example. Maybe this metric need to be revisited because not all bundles publish or consume services.

6 CONCLUSION

This work presented the design and implementation of a tool called Intrabundle. The tool extracts useful information from OSGi projects to calculate its internal quality based on static code analysis. The focus of the analysis was internal design and architecture of components where OSGi application really differs from classical Java systems.

All basic concepts were presented and it became clear that new approaches were needed to extract quality from OSGi applications. Metrics were created based on good practice in the context of Java modular applications. A quality calculation system was created to measure projects quality attributes. In the end Real OSGi projects varying from KLOCs to thousands of KLOCs, from application servers to IDEs were analyzed using the metrics proposed.

Intrabundle's quality was also a concern of this work so classical good practices like integration tests, static and dynamic analysis were applied to the tool as well as good programming techniques like immutable objects and dependency injection. The tool proved to be very useful and performed really well, taking just seconds to analyze and generate reports from huge OSGi projects. Some tendencies were verified like that is hard to keep good practices on bigger projects, as well as some OSGi specific quality aspects could also be observed.

We notice during experiments that *Stale reference* and *Uses framework* metrics were not well dimensioned so they should be revisited and calibrated in future.

The objective of this work was met. Basic aspects were studied, designed and implemented. The implementation was discussed and detailed. A fully working tool was created and presented. It provided detailed reports and reliable results that made it possible to make important assumptions about analyzed projects. We think the quality metrics created for OSGi projects were valid and useful to verify good practices in modular applications.

6.1 Future Work

Some metrics were proposed and we think more metrics can be created from the information already been collected. Also more data can be collected to enrich the analysis. As modularity is gaining focus and becoming popular we feel that the tool can be extended to other modular environments. The only difference may be how modules will be identified on those non OSGi modular applications, like JBoss Forge for example. Most metrics proposed measure attributes that are present in every modular system and so may be also used in this possible new version.

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.

TAVARES, A. L. C.; VALENTE, M. T. A Gentle Introduction to OSGi. **ACM SIGSOFT Software Engineering Notes**, Belo Horizonte, V. 33, no. 5, September 2008.

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.; SADOU, 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.

AppendixA INTRABUNDLE USAGE

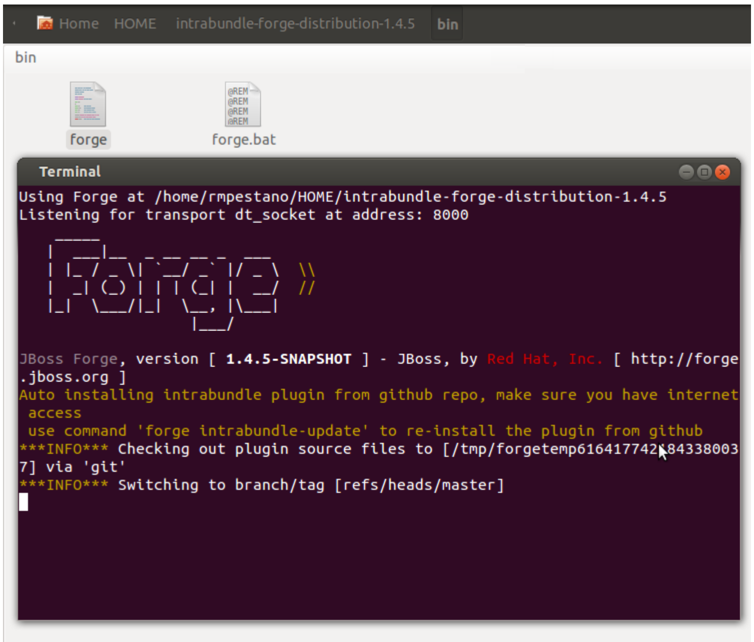
A.1 Setup environment

In this work we provide a customized Forge distribution. This distribution downloads Intrabundle from its source code repository and automatically installs it in Forge environment when Forge is started.

The only prerequisite is to have JAVA_HOME environment system variable pointing to a Java 6 or higher installation. Below are the steps to install Intrabundle and Forge:

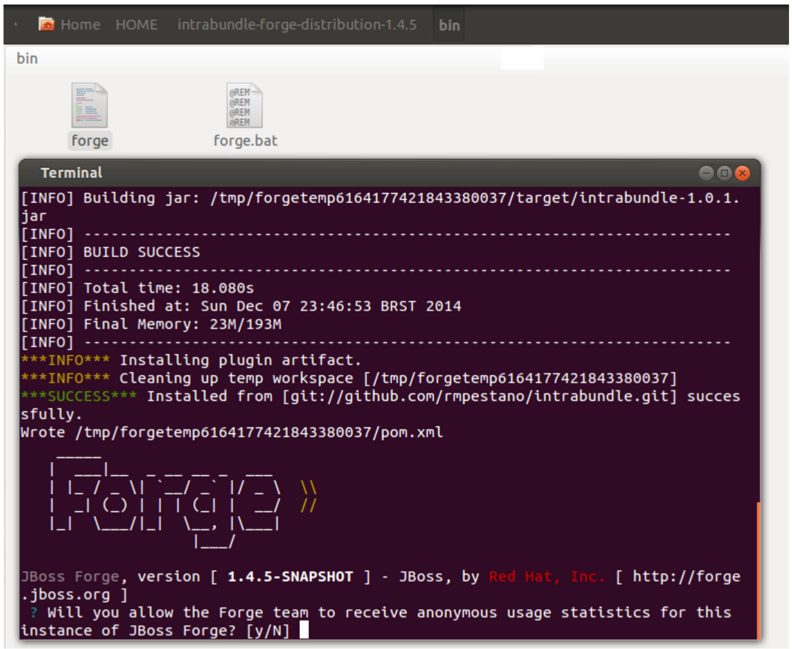
1. Download Intrabundle Forge distribution from sourceforge: http://sourceforge.net/projects/intrabundle/files/latest/download;
2. unzip it to a folder, i will call it HOME in this tutorial;
3. execute *HOME/bin/forge* file if you are on Linux or MacOS, on Windows execute *HOME\bin\forge.bat* file;
4. you should see image A.1 and image A.2 as below:

Figure A.1: Forge start



Intrabundle should be installed from its online source code repository, make sure you have internet access during this process:

Figure A.2: Intrabundle installation



There is also an online video you can watch to get you started with Intrabundle, see (intrabundle github, 2014).

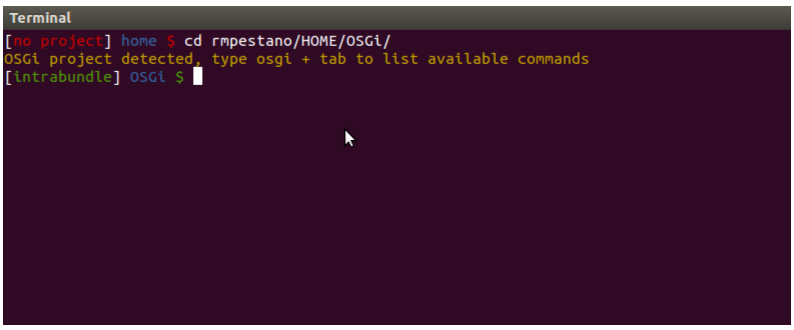
From now on you are ready to fire Forge and Intrabundle commands.

A.2 Begin Introspection

With Forge up and running now you can start OSGi project introspection with Intrabundle. An example OSGi project can be found at http://www.dcc.ufmg.br/~mtov/osgi_example.zip, it is from the article (TAVARES et al., 2008). Unzip the downloaded project to HOME and go back to Forge console.

Navigate to folder OSGI using *cd* command: *cd /HOME/OSGI*(you can use *tab* for auto completion), like in Image A.3:

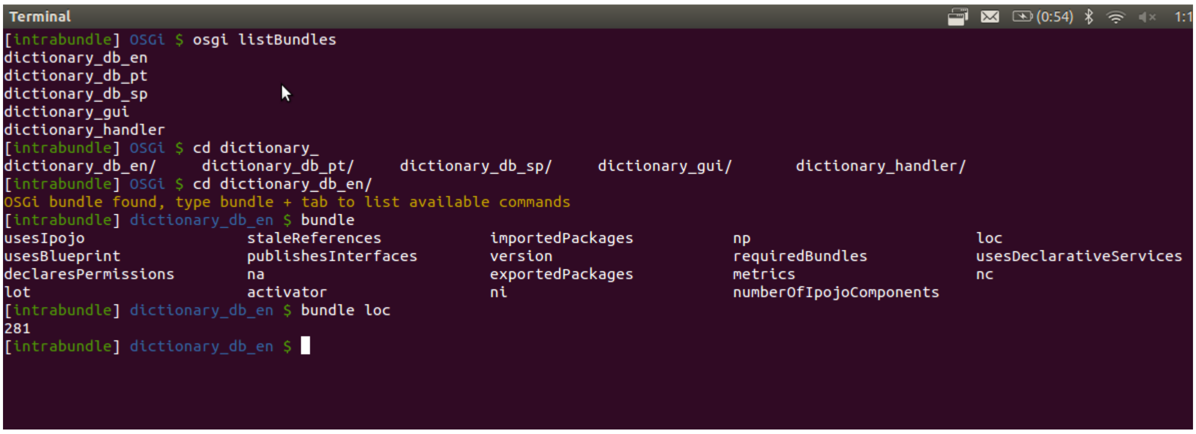
Figure A.3: Navigating to project



You can see that intrabundle recognized the OSGi project, so you can fire commands at OSGi project level like generate report or list bundles as well inspect its bundles, as in Image

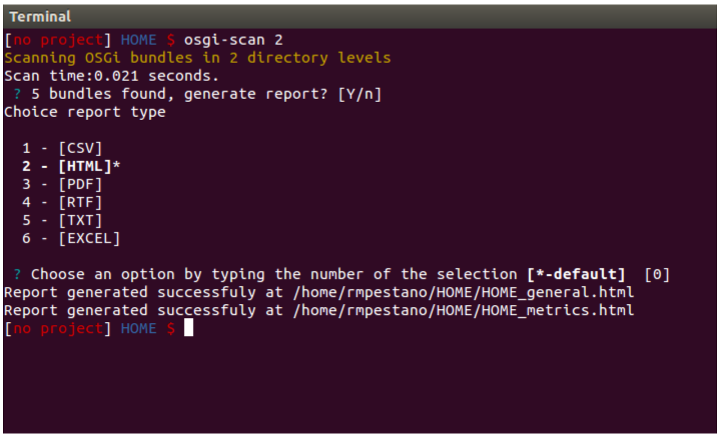
A.4:

Figure A.4: Fire commands



Another useful command Intrabundle provides is *osgi-scan*, it search for OSGi bundles in file system and generate reports on top of them. To use it go back to HOME folder and fire **osgi-scan 2** command[1], it must find bundles within downloaded project as is Image A.5:

Figure A.5: osgi-scan



[1] number argument is the depth of folders to scan