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

CISQ Consortium for IT Software Quality

JVM Java Virtual Machine

IEC International Electrotechnical Commission

ISO International Organization for Standardization

API Application Programming Interface

IDE Integrated Development Environment

GUI Graphic User Interface

LOC Lines of Code

KLOC Kilo Lines of Code

1 INTRODUCTION

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

1.1 Context

One of the pillars of sustainable software development is its quality which can basically be defined as internal and external. Internal quality focuses on how software meets its specification and works accordingly to its requirements. External 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), and also other characteristics like reusability, reliability and portability are benefited by higher software quality.

A well known and successful way to structure software architecture is to modularize its components 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). For *external quality* the classical approaches like automated testing are sufficient because in this kind of quality aims in the *behavior* and not the *design* so

[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

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].

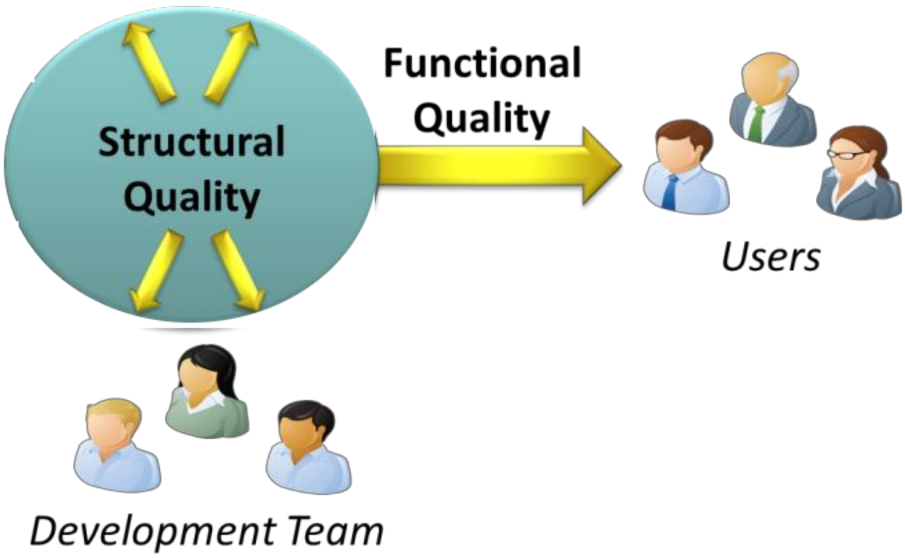
Another important aspect of software quality is that it can be divided in two groups, the **external** and **internal** quality. When we talk about *external quality* we are aiming to the user view which is the one that sees the software working and use it, this kind of quality is usually enforced through software testing. External quality can also be mapped to functional requirements so the greater external quality is the more usable and less defects it will have for example. The opposite is internal or structural quality that aims to how the software is architect-ed internally which is the perspective of the programmer and non functional requirements so the higher internal quality the better the code is structured, efficient, robust and maintainable it should be. Image 2.1 illustrates internal and external quality and its target audience.

[1] 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



2.1.1 Quality Measurement

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

2.1.2 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).

2.1.2.1 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.

2.1.2.2 *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)

2.1.3 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.

2.1.3.1 *Static Program Analysis*

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

[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

2.1.3.2 Dynamic Program Analysis

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

2.1.4 Quality Analysis Tools

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

Table 2.3: Quality analysis tools

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

Figure 2.2 shows the execution of static analysis on *Intrabundle* using *PMD*, note that 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)

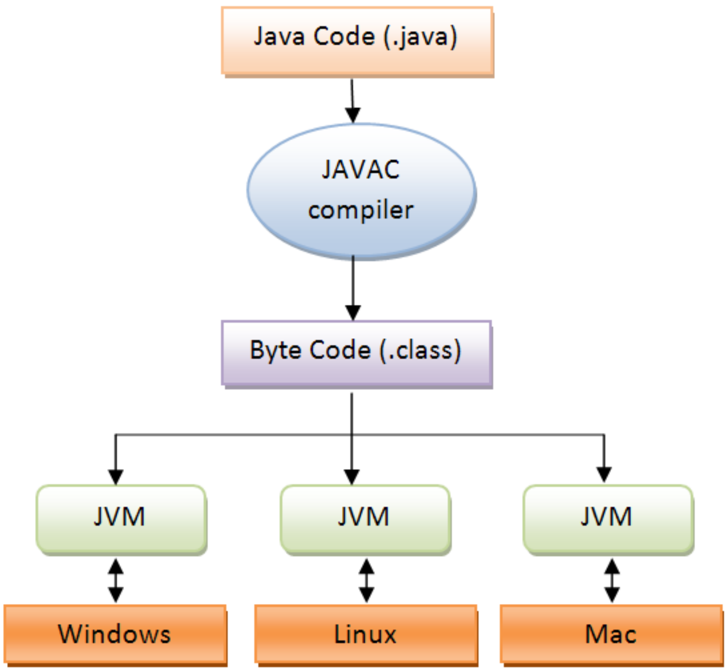
2.2 Java and OSGi

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

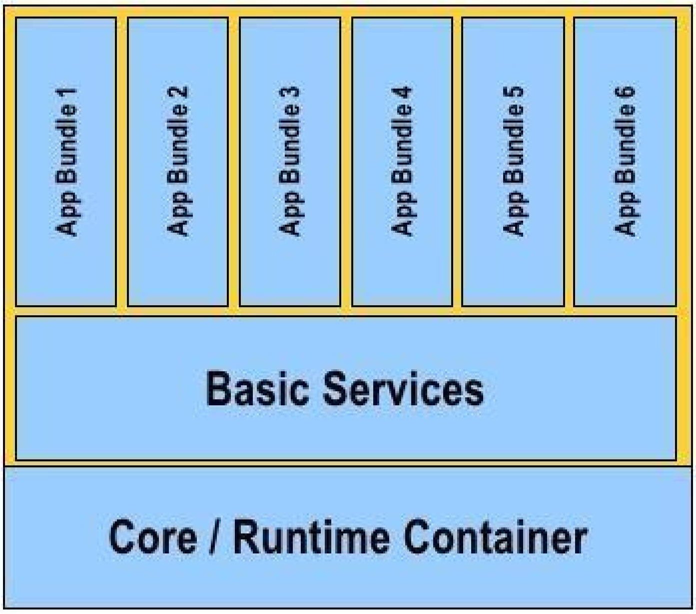
[15] Type safety is the extent to which a programming language discourages or prevents type errors

- 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**.

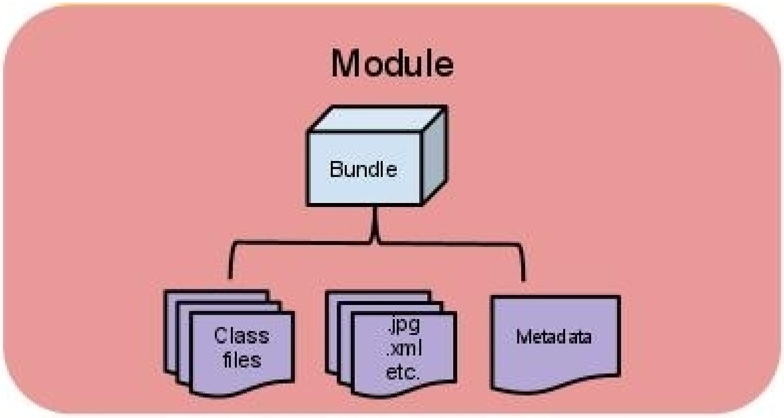
With OSGi, you modularize applications into bundles. Each bundle is a tightly coupled, dynamically loadable collection of classes packed in JARs[20], and configuration files that explicitly declare any external dependencies. All these characteristics are provided in OSGi by three conceptual layers that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

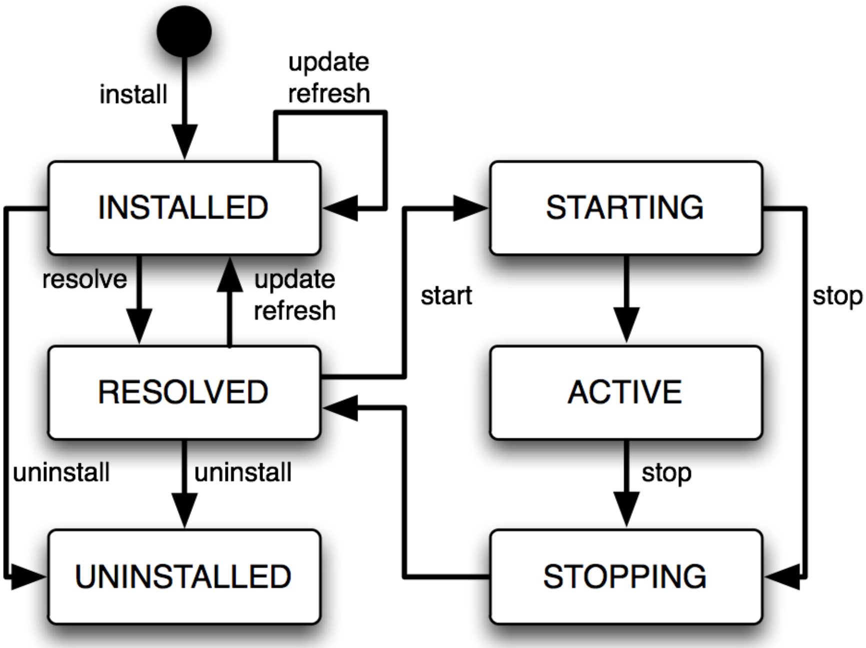


Source: OSGi conceptual layers (2011)

2.2.2.3 Lifecycle layer

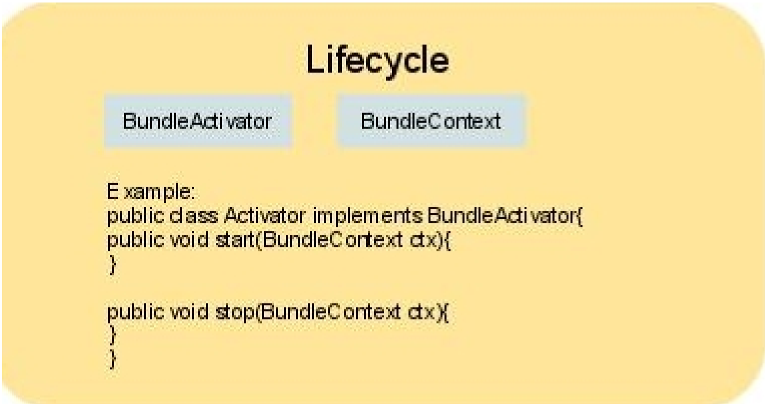
Provides access to the underlying OSGi framework through the *Bundle Context* object. This layer handles the lifecycle of individual bundles so you can manage your application dynamically, including starting and stopping bundles to manage and evolve them over time. Bundles can be dynamically installed,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



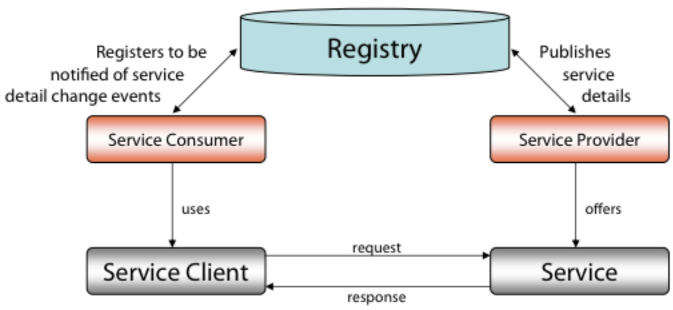
Source: OSGi conceptual layers (2011)

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



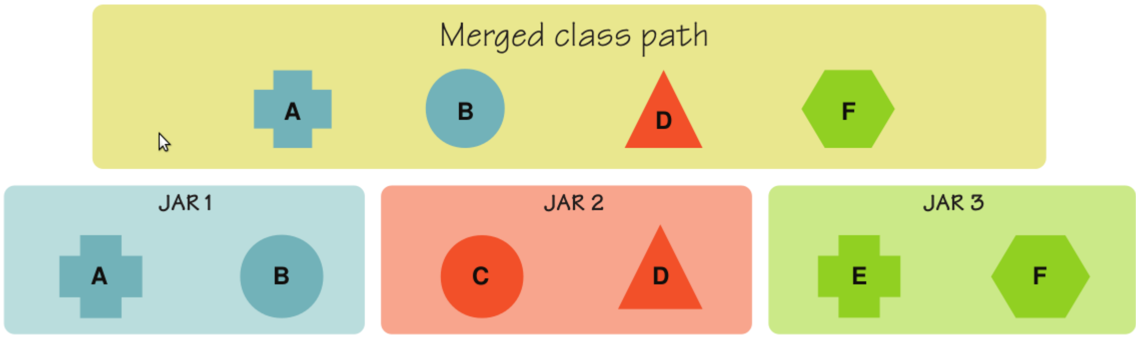
2.2.3 Vanilla Java vs OSGi

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 app lications. 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.

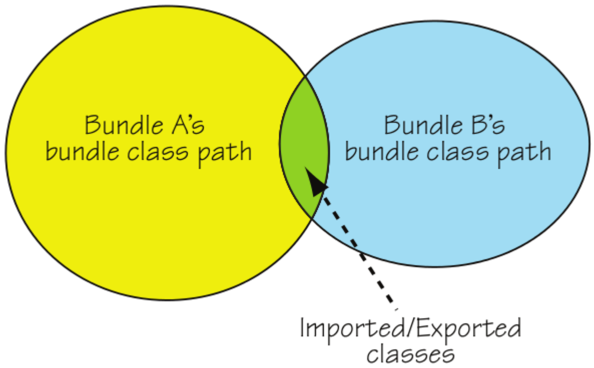
Figure 2.10: Java jar hell



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 development tool.

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[27] 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:

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.

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

[27]Term used to refer as publishing messages to the twitter social media.

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");
    }
}
```

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" );
    }
}
```

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

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 INTRABUNDLE - AN OSGI BUNDLE INTROSPECTION TOOL

3.1 Introduction

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

3.2 Design Decisions

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

- Analyze different formats of OSGi projects like 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 analyze lots of projects;
- Should output a detailed Human readable quality report so the extracted information can be analyzed.

and the following *non functional requirements*:

- Only open sourced 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;
- 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.

---

[1] Maven is a build tool for Java

[2] BND is a tool to easy OSGi projects development

[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

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

The following alternatives were evaluated:

1. Build a standalone Java client application using javaFX[5];
2. Create an eclipse plugin[6];
3. Create a Maven plugin[7];
4. Build the tool on top of JBoss Forge;
5. Extend an existing static/internal analysis tool like PMD.

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

- Works inside and outside eclipse;
- Works regardless of build tool;
- As its a command line tool its very lightweight and can analyze multiple OSGi projects at the same time;
- The programing model is based on top of the so called CDI[8] so managing Objects lifecycle and event is handled by CDI automatically;
- Forge has a very well established and documented file system manipulation api based on java.io;
- Forge is very flexible so generating quality reports is a matter of using a report API inside it;
- The author already had experience with JBoss Forge and CDI.

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

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

Maven plugins are limited to maven projects.

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

---

[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 cae CDI

[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

3.3 Implementation Overview

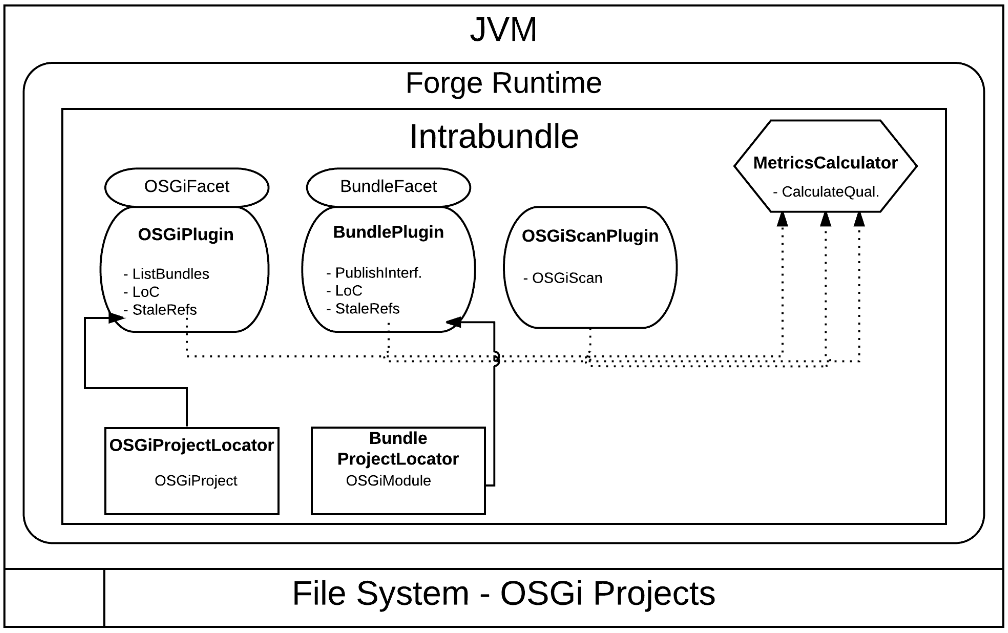
Intrabundle is composed by 3 Forge plugins, see section 2.3.2 for details about Forge plugins. The first is *BundlePlugin* which extracts OSGi bundle information, second is *OSGiPlugin* that has a vision of all bundles composed by the project. Third is OSGiScan a plugin responsible for scanning OSGi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

Intrabundle also provides 2 facets, see section 2.3.3 for details about Forge facets. *BundleFacet* and *OSGiFacet*, both restricts commands provided by BundlePlugin and OSGiPlugin in the context of OSGi bundle and project respectively. OSGiFacet is active when user enter on a directory containing an OSGiBundle and OSGiFacet is active when user in 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.

Image 3.1 illustrates Intrabundle architecture:

Figure 3.1: Intrabundle Architecture



3.4 Identifying OSGi Projects and Bundles

One important task that both *facets* and *locators*, provided by Intrabundle, perform is identifying OSGi bundles. To do that the tool searches for OSGi meta data in *MANIFEST* file[10]. So identifying bundles is as simple as locating the Manifest and verifies if it's content has OSGi information. The main problem is that the manifest location can vary depending on the project format. Table 3.1 lists the types of OSGi projects Intrabundle recognizes:

Table 3.1: Supported types of OSGi projects

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

When an OSGi project is found a message is printed in Forge console, as shown in Figure **??**. When users changes current directory to a bundle then the message shown in Figure **??** is printed.

3.5 Collecting Bundle Data

3.6 Metrics Calculation

The data collected earlier will be materialized into six metrics that will be used to calculate OSGi projects quality

3.7 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.

[10]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.

3.7.1 Internal quality

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

3.7.1.1 Example

PMD was already illustrated at Chapter 2 as an example of static analysis tool. JaCoCo is used to calculate code coverage to track files and methods that automated tests are covering. Figure 3.1 shows JaCoCo code coverage report for Intrabundle:

Figure 3.2: Intrabundle code coverage

intrabundle

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

Created with JaCoCo 0.7.1

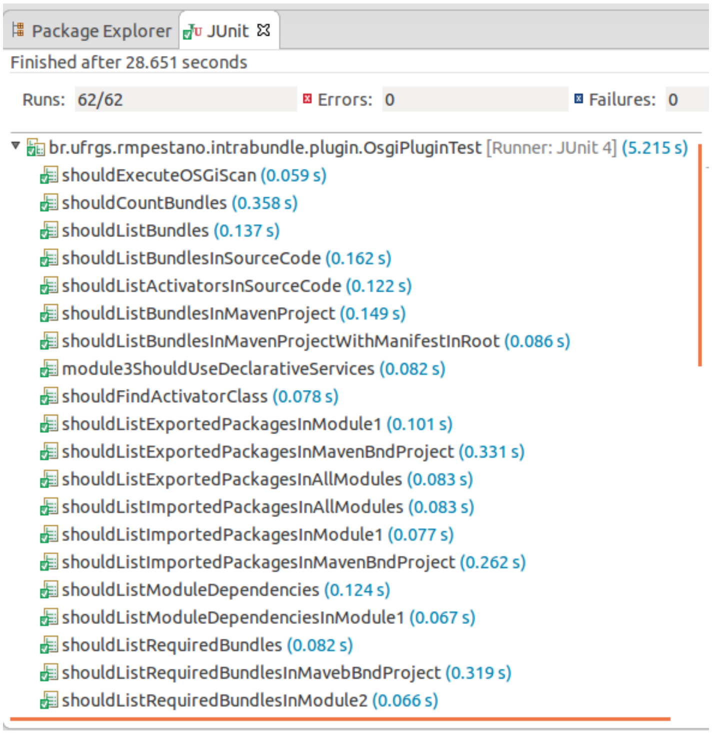
3.7.2 External quality

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

3.7.2.1 Example

As of November 2014 Intrabundle performs 62 **integration tests** which can be defined as automated tests aimed to detect any inconsistencies between the software units that are integrated together. In this kind of automated tests the system must be running and in case of Intrabundle we also need the Forge runtime up and running during tests and that is done by Arquillian (dan, 2011), an integration test platform. Figure 3.2 shows the result of integration tests execution:

Figure 3.3: Intrabundle integration tests



4 BUNDLE INTROSPECTION RESULTS

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

5 CONCLUSION

REFERENCES

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., 2011

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

WANG, T.; WEI, J.; ZHANG, W.; ZHONG, H. A Framework for Detecting Anomalous Services in OSGi-based Applications. **IEEE SCC 2012**, Honolulu, June 2012. p. 1-2

KHAN, M. E.; KHAN, F. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. **International Journal of Advanced Computer Science and Applications**, New York, v. 3, no. 6, pp 12-15, June 2012.

Wichmann, B. A.; Canning, A. A.; Clutterbuck, D. L.; Winsbarrow, L. A.; Ward, N. J.; Marsh, D. W. R. Industrial Perspective on Static Analysis. **Software Engineering Journal**, [S.l], v. 10, pp 69–75, Mar 1995.

KNORR, E. What microservices architecture really means. **InfoWorld**, Available at: http://www.infoworld.com/article/2682502/application-development/application-development-what-microservices-architecture-really-means.html. Accessed in: November 18 2014.

KRILL, P. Project Jigsaw delayed until Java 9. **InfoWorld**, Available at: http://www.infoworld.com/article/2617584/java/project-jigsaw-delayed-until-java-9.html. Accessed in: November 21 2014.

GAMA, K.; DONSEZ, D. **Service Coroner: A Diagnostic Tool for Locating OSGi Stale References**. Available at: http://www-adele.imag.fr/Les.Publications/intConferences/ECBSE2008Gam.pdf. Accessed in: November 01 2014.

HAMZA, S.; 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.

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.