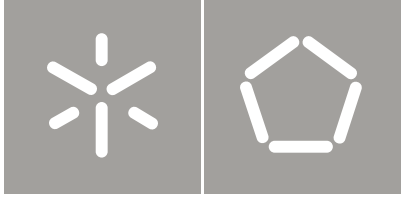
Universidade do Minho
Escola de Engenharia

João Miguel Rodrigues Quintas Veiga

Quality Assessment of Java Source Code

Quality Assessment of Java Source Code
João Miguel Rodrigues Quintas Veiga
UMinho | 2010

Outubro de 2010



Universidade do Minho
Escola de Engenharia

João Miguel Rodrigues Quintas Veiga

Quality Assessment of Java Source Code

Tese de Mestrado
Informática

Trabalho efectuado sob a orientação da
Professora Doutora Maria João Gomes Frade

Outubro de 2010

Acknowledgements

First of all I want to thank my supervisor Professor Maria João Frade for the research guidance and also for all the suggestions that improved this thesis.

I am grateful to Professor Joost Visser for showing me the Sonar platform.

Many thanks to Multicert company for showing interest in this project and for providing case studies. A special thanks to Renato Portela who was always available to help.

Finally, I want to thank my family, for all their support along the years.

Quality Assessment of Java Source Code

Abstract

The development of software products is a people-intensive process and several software development methodologies are used in order to reduce costs and enhance the quality of the final product. Source code quality assessment is a crucial process in the software development, focusing in the certification of the quality of the code in its various aspects (functionality, reliability, maintainability, portability, etc). It contributes to the undeniable reduction in product costs and helps to increase the quality of final software.

This master thesis focuses on assessing the quality of Java source code. A research is made on quality models and existing standards, quality factors and the associated metrics, and the tools available to analyse Java source code and to calculate these metrics. We also identify the existing methodologies for determining metrics thresholds in order to better understand the obtained metrics results.

However, software metrics are not enough to evaluate the quality of a software product. We describe other ways of analysing source code like unit testing and static analysis. These complementing analysis and the results of the metrics allow to create a more complete view of the quality of the source code in its different aspects.

After making a brief survey of existing software metric tools for Java source code we concentrate on Sonar: an open source tool used to analyse and manage source code quality in Java projects. Sonar evaluates code quality through seven different approaches: architecture & design, complexity, duplications, coding rules, potential bugs and tests.

We developed a new plugin for Sonar for design quality assessment of Java programs, which we call TreeCycle. The TreeCycle plugin represents the dependencies between packages in a tree graph highlighting it's dependency cycles. For each package it represents in a graphical way the results of a suite of metrics for object-oriented design. This plugin helps to provide an overall picture of the design quality of Java projects.

Finally, using Sonar and the TreeCycle plugin, we analyse two industrial projects of medium size in order to show the usefulness of this tool in the process for assessing the quality of software products.

Avaliação da Qualidade de Código Fonte Java

Resumo

O desenvolvimento de produtos de software é um processo intensivo em que várias metodologias de desenvolvimento de software são utilizadas para reduzir custos e melhorar a qualidade do produto final. A avaliação da qualidade de código fonte é essencial para o desenvolvimento de software, focando na certificação da qualidade do código nos seus vários aspectos (funcionalidade, confiabilidade, manutenção, portabilidade, etc.), contribuindo assim para a inegável redução dos custos e o aumento da qualidade final do software.

Esta tese de mestrado foca na determinação da qualidade de código fonte Java. A investigação é feita sobre modelos de qualidade e padrões existentes, factores de qualidade e as métricas associadas, e as ferramentas disponíveis para analisar código fonte Java e calcular essas métricas. Identificamos também as metodologias existentes para determinar valores de referências de modo a compreender melhor os resultados obtidos pelas métricas.

Contudo, as métricas para software não são suficientes para avaliar a qualidade de um producto de software. Descrevemos outras formas de analisar código fonte, tais como testes unitários e análise estática. Estas análises complementares e os resultados das métricas criam uma imagem mais completa da qualidade do código fonte.

Após um breve levantamento das ferramentas existentes para calcular métricas sobre código fonte Java, concentramo-nos no Sonar: uma ferramenta *open source* utilizada para analisar e gerir a qualidade de código fonte em projectos Java. O Sonar avalia a qualidade do código a partir de sete abordagens: *design*, arquitectura, complexidade, duplicação, regras de codificação, potenciais erros e testes.

Desenvolvemos um novo *plugin* para o Sonar que permite analisar a qualidade de *design* de programas Java, o qual designamos TreeCycle. O *plugin* TreeCycle representa as dependências entre pacotes numa árvore onde também são assinalados os ciclos de dependências. Para cada pacote, este representa de uma forma gráfica os resultados do conjunto de métricas para *design* orientado aos objectos. Este *plugin* permite criar uma imagem global da qualidade de *design* de projectos Java.

Por fim, utilizando o Sonar e o *plugin* TreeCycle, analisamos dois projectos industriais de médio tamanho para demonstrar a utilidade desta ferramenta no processo de avaliação da qualidade de um producto de softwre.

"...when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind..."

William Thomson Kelvin (1824 - 1907)

Contents

1 Introduction **17**

2 Quality Models **23**

2.1 McCall's Quality Model 23
2.2 Boehm's Quality Model 24
2.3 ISO/IEC 9216 25

3 Software Metrics **27**

3.1 Some Traditional Metrics 27
3.2 Software Quality Metric Methodologies 28
3.2.1 IEEE Standard 1061 28
3.2.2 Goal Question Metric Approach 29
3.3 Object-Oriented Design Metrics 30
3.3.1 C&K Metrics Suite 30
3.3.2 R.C. Martin Metrics Suite 32
3.3.3 Metrics for Object-Oriented Design Suite 33
3.3.4 Lorenz & Kidd Metric Suite 34
3.4 Software Metrics Thresholds 34
3.5 Software Quality Evaluation Process 35
3.5.1 Process for Developers 36
3.5.2 Process for Acquires 37
3.5.3 Process for Evaluators 38
3.6 Software Metrics Tools 39
3.6.1 CyVis 39
3.6.2 JavaNCSS 40
3.6.3 JDepend 42
3.6.4 CKjm 44
3.6.5 Eclipse Plugin 44

3.6.6 Survey Results 46

4 Complementing Software Metrics Information 49

4.1 Unit Testing 49
4.1.1 Different Types of Tests 50
4.1.2 Java Unit Testing Framework (JUnit) 50
4.1.3 Stubs and Mock Objects 55
4.1.4 Unit Tests and Code Coverage 55
4.2 Static Analysis 58
4.2.1 FindBugs 59
4.2.2 PMD 60
4.2.3 CheckStyle 61

5 Sonar: A Platform For Source Code Quality Management 63

5.1 Sonar Functionalities 64
5.1.1 Violations Drilldown 65
5.1.2 Dependency Structure Matrix 65
5.1.3 Coverage Clouds 66
5.1.4 Hotspots 66
5.1.5 Components 67
5.1.6 Time Machine 67
5.1.7 Quality Profiles 67
5.2 Sonar Plugins 67
5.3 The TreeCycle Plugin 68
5.3.1 How It Works 69
5.3.2 Assessing TreeCycle with Sonar 72
5.4 Sonar in the Evaluation Process 77

6 Case Studies 79

6.1 Maestro Web Service Test Project 80
6.1.1 Statical Analysis (Rules Compliance) 80
6.1.2 OOD Metrics (Design & Architecture) 86
6.2 SMail J2EE Project 90
6.2.1 Statical Analysis (Rules Compliance) 91
6.2.2 OOD Metrics (Design & Architecture) 94

7 Conclusions and Future Work **99**
7.1 Conclusion 99
7.2 Future Work 100

List of Figures

2.1 McCall software Quality Model 24
2.2 Boehm's Quality Model 25
2.3 ISO/IEC 9216 internal and external quality model 26

3.1 FreeCS example in CyVis 40
3.2 FreeCS example in JavaNCSS 41
3.3 FreeCS example in JDepend 43
3.4 FreeCS example in CKjm 45
3.5 FreeCS example in Eclipse Plugin 47

4.1 Running *testToStringPlayer* with JUnit in Eclipse 53
4.2 Running *testToStringPlayer* with EMMA in Eclipse 58

5.1 Example of a dashboard of a project in Sonar 64
5.2 TreeCycle: package dependencies tree graph 70
5.3 TreeCycle: list of dependency cycles 71
5.4 TreeCycle: C&K metrics 72
5.5 TreeCycle: general information 73
5.6 TreeCycle: quality evolution 73
5.7 TreeCycle: rules compliance info 74
5.8 TreeCycle: rules compliance drill-down 74
5.9 TreeCycle: design & architecture 75
5.10 TreeCycle: dependencies tree 75
5.11 TreeCycle: C&K metrics results 76
5.12 TreeCycle: unit tests 77
5.13 TreeCycle: coverage cloud 77

6.1 Maestro: general information 81
6.2 Maestro: rules compliance info 81

6.3 Maestro rules compliance drill-down 82
6.4 Maestro: design & architecture 86
6.5 Maestro: dependencies tree 87
6.6 Maestro: dependency structure matrix 87
6.7 Maestro: C&K metrics results 89
6.8 Maestro: lack of cohesion methods 90
6.9 SMail: general information 90
6.10 SMail: rules compliance info 91
6.11 SMail: rules compliance drill-down 91
6.12 SMail: design & architecture 94
6.13 SMail: dependency tree 95
6.14 SMail: dependency structure matrix 95
6.15 SMail: C&K metrics results 96
6.16 SMail: lack of cohesion methods 97

List of Tables

3.1 Tools results 47

6.1 C&K metrics thresholds 80

1 Introduction

Software quality assessment is on the agenda due to several factors among which include the development of increasingly complex software, the use of libraries developed by third parties, the use of open source, as well as the integration of pieces of code from various sources. Software engineering remains a people-intensive process and several software development methodologies are used in order to reduce costs and enhance the quality of the final product. Software quality assessment is a crucial process in the software development, focusing in the certification of the quality of the code in its various aspects (functionality, reliability, maintainability, portability, etc). It contributes to the undeniable reduction in product costs and helps to increase the quality of final software.

But what is meant by *software quality*? The concept of software quality is ambiguous. Some software engineers relate software quality to the lack of bugs and testing, others relate it to the customer satisfaction, or the level of conformity with the requirements established [13, 36]. Therefore it all depends very much on the point of view of each person.

Quality is a complex and multifaceted concept. In [18] David Garvin presented a study on different perspectives of quality in various areas (philosophy, economics, marketing, and operations management) and identified five major perspectives to the definition of quality. In the *transcendent* perspective quality is something that can not be defined and can only be identified through gained experience. In the *product-based* perspective quality is something that can be evaluated or measured by the characteristics and attributes inherent to a product. In the *user-based* perspective the quality of a product is evaluated or measured through consumer satisfaction and consumer demand. The *manufacturing-based* perspective relates quality with the level of conformance of the product with its requirements. And in the *value-based* perspective the quality of a product is evaluated through its manufacturing cost and final price: no matter how good a product is, its quality does not matter if it is too expensive and no one buys it.

Our focus will be the product-based perspective of software quality. In this view, software quality can be described by a hierarchy of quality factors inherent to the software

product and all its components (source code, documentation, specifications, etc).

This master thesis focuses on assessing the quality of Java source code. We aim to identify and understand the existing standards, the quality factors and the associated metrics, and the tools available to analyse Java source code and to calculate these metrics. It was also our goal to implement a tool to produce reports on source code analysis according to the established methodology, and finally to apply this knowledge and this tool to medium case studies.

Quality Models

Over the years many software quality models have been proposed. These models define, in general, a set of characteristics (quality factors) that influence the software product quality. Those characteristics are then divided into attributes (quality sub-factors) that can be measured using software metrics. These models are important because they allow for a hierarchical view of the relationship between the characteristics that determine the quality of a product and the means for measuring them, thus providing an operational definition of quality.

The quality models proposed by Jim McCall [7, 17] in 1977 and by Barry W. Boehm [7, 5] in 1978 are the predecessors of modern quality models. Both these models use an hierarchical approach and were the basis for the ISO/IEC 9126 [25, 32], a standard that aims to define a quality model for software and a set of guidelines for measuring the quality factors associated with it, and that is probably one of the most widespread quality standards.

Java

As already stated we will focus on Java code. Java [2] programming language was developed in the early 90's by a team of engineers at Sun Microsystems, led by James Gosling. Its is an object-oriented language since it implements many of the features related to object-oriented paradigm like the concept of object, class, encapsulation, inheritance, data abstraction and polymorphism. It is an interpreted language since Java source code is compiled to byte-code that can later be interpreted by a Java virtual machine independently of computer architecture. It is also a multi-threaded, distributed, dynamic language, making it suitable for developing web applications. However it is also mature, simple, robust and secure, making it the most popular programming language, with a large community support. It is for these reasons that more software companies

are investing in Java technology for developing there software products.

Metrics

Metrics are defined as being "the process by which numbers or symbols are assigned to attributes of entities in the real world in such way as to describe them according to clearly defined rules" [16]. We use tens of metrics in our daily lives. In software engineering metrics can be used to determine the cost and effort of a software project, staff productivity, and also the quality of a software product [16, 50]. Software metrics are the most direct way to evaluate each factor that forms the quality model of a software product.

Because more traditional metrics were proved to be incapable of dealing with concepts specific to the object-oriented paradigm, object-oriented design (OOD) metrics where developed and proposed. Examples of object-oriented design metrics suites are the ones of Chidamber & Kemerer [11, 53], Robert C. Martin [45, 44], Fernando Brito e Abreu [14, 21], and Lorenz & Kidd [21].

When working with software metrics one has to know how to interpret the obtained results, in order to make decisions based on them. Reference values are needed to determine whether the metrics results are too high, too low, or normal, these reference values are known as software *metrics thresholds*. Not too much work is been done about this topic, however there are some methodologies based on empirical studies for determining software metrics thresholds.

However, software metrics are not enough to evaluate the quality of a software product. It is important to use other techniques, like unit testing and static analysis, to complement the information obtained through software metrics, thus creating a more complete report.

Sonar

Sonar[1] is an open source tool used to analyse and manage source code quality. Sonar follows the ISO/IEC 9126 to assess the quality of the projects under evaluation and it provides as core functionality code analysers, defects hunting tools, reporting tools and a time machine. It enables to manage multiple quality profiles and also has a plugin mechanism giving the opportunity to extend the functionality to the community.

[1] http://www.sonarsource.org/

Sonar has more than forty plugins available, however only four plugins are devoted to visualisation and report of results.

We have developed a Sonar plugin (the TreeCycle[2]) for design quality assessment of Java programs. The TreeCycle plugin helps in the analysis of design quality by representing the dependencies between packages in a tree graph highlighting it's dependency cycles. Moreover, for each package it represents in a graphical way the results of a suite of metrics for object-oriented design. The use of this plugin provides an overall picture of the design quality of a Java project and will enhance reports produced about the code.

Finally, we analyse two industrial projects of medium size that were developed by Multicert[3], a Portuguese company that develops complete security solutions focused on digital certification for all types of electronic transactions that require security.

These analysis were made with Sonar (among with TreeCycle and other Sonar plugins) to show the usefulness of this tool in the process for assessing the quality of software products. In the analysis of each project, there were presented several examples of cases that are related to different aspects of the software product's source code (design, architecture, coding rules compliance, unit test coverage), that contribute somehow to the diminution of its quality. The report produced can be used as evidence to propose improvements to the source code.

Organization of the Thesis

The rest of this thesis is organized as follows:

Chapter 2 presents the quality models commonly used to define the set of characteristics and attributes on which the software product will be evaluated: McCall's quality model, Boehm's quality model and ISO/IEC 9216.

Chapter 3 is devoted to software metrics with special emphasis on object-oriented design metrics. We also describe the IEEE Standard 1061 and the Goal Question Metric Approach, two methodologies for identifying, analysing, and validating software quality metrics. Moreover, we briefly present three techniques for deriving software metrics thresholds and analyse ISO/IEC 14598, a standard that defines the process for measuring and assessing the quality of software products. This chapter finishes with a small survey of software metric tools capable of measuring all metrics presented in this chapter.

[2]http://wiki.di.uminho.pt/twiki/bin/view/Research/CROSS/Tools
[3]https://www.multicert.com/home

Chapter 4 identifies and describes various techniques and tools capable of complementing the information obtained through software metrics, these being the following: static analysis and unit testing.

Chapter 5 presents Sonar: a tool for source code quality management and try to understand how this tool can be used in an evaluation process. We also present the TreeCycle plugin, describing how it works and giving examples of its use and analyse it.

Chapter 6 is devoted to the presentation of two case studies of medium size and the interpretation of the results obtained through Sonar.

Chapter 7 is reserved for conclusions and directions that can be taken in future work.

2 Quality Models

Having defined the concept of software quality is now necessary to identify the set of quality factors and sub-factors, also known as *quality models*, on which the software product will be measured and evaluated.

Next an overview of the ISO/IEC 9126 quality model will be done because it is considered "one of the most widespread quality standards" [8], and also McCall's and Boehm's quality models from who the ISO/IEC 9126 was based. But there are others like the FURPS quality model proposed by Robert Grady and Hewlett-Packard Co. that has the peculiarity of dividing the quality factors into two categories (functional and non-functional), or the quality model proposed by R. Geoff Dromey that tries to adapt to each different software product, but they seem not to be so well-known and will not be featured in this chapter.

2.1 McCall's Quality Model

This model, known as the "first of the modern software product quality models" [5], was proposed by Jim McCall [7, 17] in 1977 and is used in the United States in various military, space and public projects.

The McCall model, as shown in Figure 2.1, is formed by eleven quality factors that represent the *external view* or the way users perceive the quality of a software product. These quality factors form a hierarchy relationship with the sub-factors of the software product also known as *quality criteria* that represent the *internal view* or the way the developer perceives the quality of a software product. The software metrics provide a way of measuring the quality criteria and therefore evaluating the quality factors. There are three major perspectives of software product quality in the McCall model.

The product operation where the quality of the software product is measured by its operational characteristics includes the following quality factors: *efficiency* (efficient use of computer resources), *usability* (cost and effort to learn how to handle the software product), *integrity* (program's level of protection against unauthorized access), *correct-*

ness (specification conformity) and *reliability* (the probability of failure). The product revision where the quality of a software product is based on it's capability to be updated includes the following quality factors: *maintainability* (effort necessary to locate and fix an error), *testability* (ease of testing a software for specification violations) and *flexibility* (cost of modifying the software product). The product transition where the quality is measured by the capability of the software product to adapt to new environments includes the following quality factors: *re-usability* (the cost reusing it in another software product), *portability* (effort of transferring the software from a environment to another) and *interoperability* (effort of coupling a software product to another).

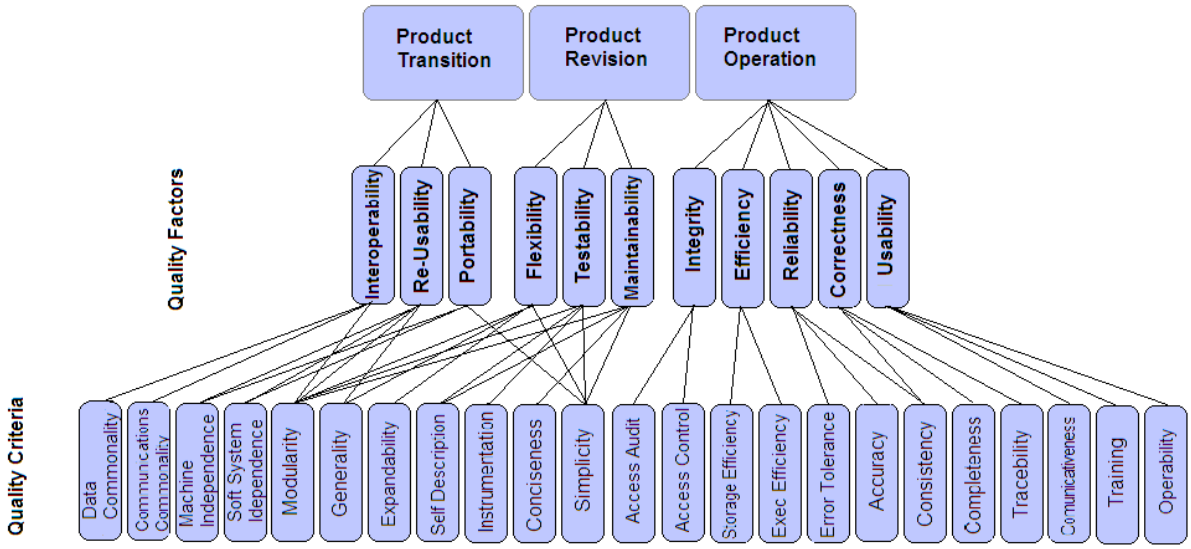


Figure 2.1: McCall software Quality Model

2.2 Boehm's Quality Model

Another of the first quality models was proposed by Barry W. Boehm [7, 5] in 1978 and uses the same hierarchical approach as the McCall software quality model. The notion of quality in Boehm's model, as shown in Figure 2.2, is represented by three high level characteristics, *maintainability* that represents the effort to understand, modify and test the software product, *portability* that represents the effort to adapt to a new environment, and *as-is utility* that requires the software product to be easy and reliable to use. These three high level characteristics represent the user's point of view and are linked to seven intermediate characteristics similar to the quality factors in the McCall model (portability, reliability, efficiency, flexibility, testability, understandability

and modifiability), which in turn are divided into low-level attributes upon which the software metrics will be applied.

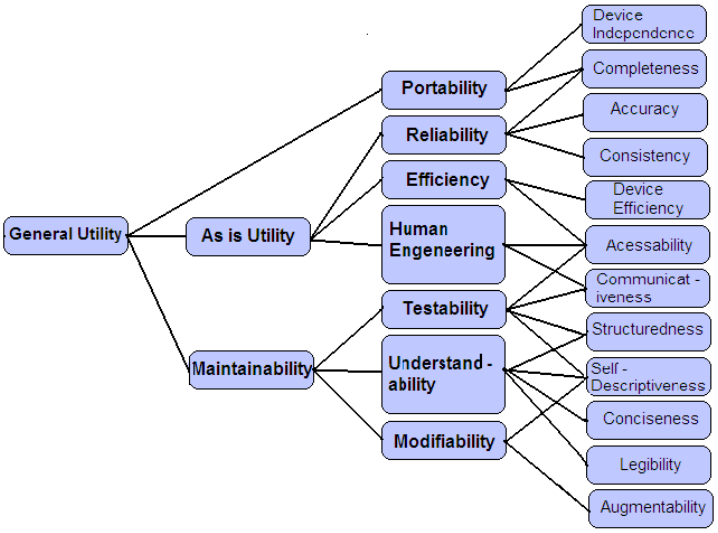


Figure 2.2: Boehm's Quality Model

2.3 ISO/IEC 9216

The International Organization for Standardization (ISO) presented in 1991 the first international standard on software product evaluation: ISO/IEC 9126: *Software Product Evaluation - Quality Characteristics and Guidelines for Their Use* [25]. This standard intended to define a quality model for software and the guidelines for measuring the characteristics associated with it. The standard was further developed during 2001 to 2004 period and is now published by the ISO in four parts: the quality model [32], external metrics [33], internal metrics [34] and quality in use metrics [35].

ISO/IEC 9126 is considered one of the most widespread quality standards. The new release of this standard recognises three views of software quality:

- *External quality*: covers characteristics of the software that can be observed during its execution.
- *Internal quality*: covers the characteristics of the software that can be evaluated without executing it.
- *Quality in use*: covers the characteristics of the software from the user's view, when it is used in different contexts.

The quality model in ISO/IEC 9126 comprises two sub-models: the internal and external quality model, and the quality in use model.

The internal and external quality model was inspired from McCall's and Boehm's models. Figure 2.3 illustrates this model. The model is divided in six quality factors: functionality, reliability, usability, efficiency, maintainability, and portability; which are further subdivided into 27 sub-characteristics (also called attributes or quality subfactors). The standard also provides more than an hundred metrics that can be used to measure these characteristics. However those metrics are not exhaustive, and other metrics can also be used.

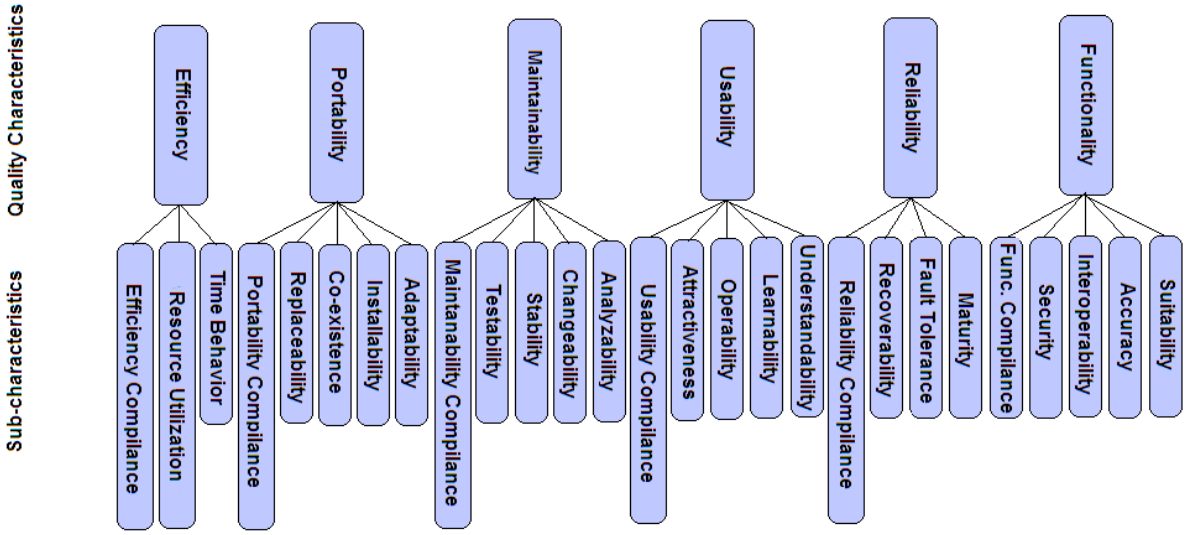


Figure 2.3: ISO/IEC 9216 internal and external quality model

The quality in use is modelled in a different way. It breaks-down in four quality factors: security, satisfaction, productivity and efficiency. These quality factors are not subdivided further.

The re-usability factor is defined by McCall et al. [47] as the cost of transferring a module or program to another application and although ISO/IEC 9126 does not contemplate it, re-usability can be seen as a special case of usability [41].

3 Software Metrics

After seeing different software quality models and identifying the quality factors and sub-factors of software products that can influence its quality it is now important to understand how the attributes will by measured and evaluated.

Software quality metrics fall in the category of *product metrics* that are applied to the software product including source code and documentation produced during development stage. However there are also *process metrics* used to measure the development process, such as time, cost, and effort. Software metrics can also be categorized as being *direct* if they don't rely on other metrics, or *indirect* otherwise [37].

3.1 Some Traditional Metrics

Next we briefly present two of the first, best-known and most used software product metrics.

Lines of code (LOC). This metric[48, 50] is one of most well-known metrics, and it is used to determine the size of a software product. However even this apparently simple metric can be difficult to define because the meaning of "line of code" can include comments, non-executable statements and even blank lines. This metric is considered one of the best all-around error predictors [48].

Cyclomatic complexity (CC). This metric was developed by Thomas McCabe [46, 48] in 1976 and measures the number of linearly independent paths through a program using its control flow graph. This metric measures the level of complexity. The higher the cyclomatic complexity value, the harder it is to understand the source code and test the program. Therefore high cyclomatic complexity leads to loss of software quality.

Halstead's Metrics Set. In 1977 Maurice Halstead [50, 48] proposed a set of metrics based on the number of operators and operands in a module (function or method).

Halstead defines, among others, the length (N) that given the number of operators in a program (η_1) and number of operands (η_2) can be calculated using the formula $N = \eta_1 + \eta_2$.

Given the number of unique operators (μ_1) and the number of number of unique operands (μ_2), the *vocabulary* of a program (U) can be calculated through the formula $U = \mu_1 + \mu_2$.

The *volume* (number of bits) of a program (V) can be seen as an indirect metric that is calculated through the formula $V = N * \log_2 U$.

3.2 Software Quality Metric Methodologies

Next we describe two methodologies for identifying, analysing, and validating software quality metrics.

3.2.1 IEEE Standard 1061

The Institute of Electrical and Electronics Engineers (IEEE) published in March 1993 the standard 1061 [24] for a software quality metrics methodology. It provides a five step methodology for identifying, implementing, analysing, and validating process and product software quality metrics for the established quality requirements throughout a software life cycle. This standard also provides a software quality metrics framework for defining simple quality models formed by quality factors, sub-factors and the metrics that measure them.

The first step in the standard 1061 methodology consists in the creation of a list of software quality requirements. All parties involved in the project have to participate in the creation of this list and it is advised the use of organizational experience, required standards, regulations, or laws. And for each quality factor one or more direct metrics should be assigned.

The second step begins by applying the software quality metrics framework by assigning quality factors to each quality requirement. Each quality factor is decomposed into quality sub-factors which in turn are related to metrics. It is also important to define a target value, a critical value and the range for each metric. After all this it is necessary to document each metric by giving emphasis to the costs and benefits of applying them. This step helps defining the set of metrics that will be used throughout the project.

The third step has describes how the software quality metrics will be implemented.

First is necessary to define for each metric the data and tools that will be used. It is also important to describe data storage procedures, who will be making the data collection, and establishing a traceability matrix between metrics and the data collected. Then the tools and data collection are tested to help improve metric and data descriptions, as well as to examine the cost of the measurement process. After successful testing, all collected data has to be stored in the project metrics database and calculate the metric values through the use of the collected data.

The fourth step is related to the analysis the software metrics results. After analysing and recording the metric results it is important to identify values that are outside the tolerance intervals. These values can represent undesirable attributes like excessive complexity, or inadequate documentation, that leads to the non conformance with the quality requirements. Depending on the results collected, it can be necessary to re-design the software component or even to create a new one from scratch.

Finally, the fifth step describes how to validate software quality metrics by applying a validity criteria defined in the standard 1061.

3.2.2 Goal Question Metric Approach

The Goal Question Metric (GQM) [6, 8] was originally created for evaluating defects for projects in the NASA Goddard Space Flight Center and used by several organizations like NASA, Hewlett Packard Motorola, and Coopers & Lybrand. This approach is used by companies and organizations that want to improve their use of software metrics. The GQM approach works by first specifying a set of goals for the company and its projects, then tracing those goals to the data that defines them operationally, and finally providing a framework for interpreting and quantifying the data turning it into information that can be analysed as to whether the goals have been achieved or not.

From the GQM approach the result is a three level measurement model through which one can measure the collected data and interpret the measurement results. The first level is named *conceptual level* and is where the goals are chosen for an object taking into account various aspects like the object's environment, quality models, and different points of view. These objects can be different components from the software product like, for example, specifications, designs, programs or test suits. They can be activities related to the Software development process like the testing, designing or specifying process. And they can also be resources used by the development process like hardware, software or personnel.

In the second level named *operational level* is where a set of questions are created

for each goal, based on a characterizing model of the object like a quality model or a productivity model. These set of questions are created to better understand how the goals will be achieved.

Finally in the third level named *quantitative level* each question will be associated to one or more metrics that need to be calculated in order to answer the question in a quantitative way.

3.3 Object-Oriented Design Metrics

The object-oriented paradigm brought a new way of viewing and developing software systems. This can be seen as a group of objects that interact with each other through message passing trying to solve a problem. An object-oriented programming language has to provide support for object-oriented concepts like objects, classes, encapsulation, inheritance, data abstraction and message passing.

There are metrics especially designed to measure distinct aspects of the object-oriented approach. Some sets of object-oriented design metrics have been proposed. And there are authors who have tried to relate these metrics to the quality factors that form the ISO/IEC 9126 quality model [41]. Next we present four sets of object-oriented design metrics and their relation with the quality factors described in Chapter 2.

3.3.1 C&K Metrics Suite

In 1994 Shyam R. Chidamber and Chris F. Kemerer proposed a metrics suite for object-oriented design [11, 53]. This suite consists of six metrics.

Weighted methods per class (WMC) metric is equal to the sum of all methods complexities in a class. A method complexity can be measured by the cyclomatic complexity, however a definition of complexity was not proposed by Chidamber and Kemerer in order to allow for general applications of this metric. If methods complexities are considered to be unity, the WMC metric turns in to the number of methods in a class. The WMC gives an idea of the effort required to develop and maintain the class. Since the children of a class inherit all its methods, the number of methods in a class have potential impact on its children. Classes with many methods are probably more application specific. High WMC values negatively influences maintainability and portability, because complex classes are harder to analyse, test, replace or modify. It also negatively influences re-usability since it is harder to understand and learn how to integrate complex classes.

Depth of inheritance tree (DIT) metric determines the number of ancestors of a class in the hierarchy of classes. Deep inheritance trees make the design complex. This metric negatively influences maintainability and portability because classes with high DIT potentially inherit more methods, and so it is more complex to predict their behaviour. However re-usability benefits from classes with high DIT because those classes potentially have more inherit methods for reuse.

Number of children (NOC) metric is equal to the number of immediate subclasses subordinated to a class. NOC gives an idea of the potential influence a class has on the design. If a class has a large NOC, it may justify more tests. If NOC is too high, it can indicate that the subclass structuring is not well designed. Re-usability benefits from classes with high NOC since inheritance is a form of reuse. NOC affects portability and maintainability, because classes with subclasses that depend on it are harder to replace or change.

Coupling between object classes (CBO) metric represents the total number of other classes a class is coupled to. A class is coupled to another class if methods of one uses methods or instance variables from the other. Excessive coupling is bad for modular design. It makes classes complex and difficult to reuse. It also makes testing a more difficult task and makes software very sensitive to changes. CBO is so highly connected to portability, maintainability and re-usability.

Lack of cohesion in methods (LCOM) metric determines the difference between the number of pairs of methods of a class that do not share instance variables and the number of pairs of methods that share instance variables. This metric helps to identify flaws in the design of classes. For instance, high lack of cohesion in methods may indicate that the class would be better divided into two or more subclasses. Low cohesion increases complexity. So, classes with high LCOM values are harder to understand and test. Therefore, LCOM influences maintainability and re-usability.

Response for a class (RFC) metric represents the number methods, including methods from other classes, that can be executed in response to messages received by objects from the class. RFC is an indicator of class complexity and of the test effort required. Classes with high RFC are harder to test and debug, since they are harder to understand. These reason also make classes with high RFC more difficult to reuse and less adaptable to changes. Hence RFC negatively influence maintainability, re-usability and portability.

3.3.2 R.C. Martin Metrics Suite

Robert C. Martin proposed in 1994 a set of metrics for measuring the quality of an object-oriented design in terms of the interdependence between packages [45, 44]. This suite consists of the following metrics.

Afferent couplings (CA) metric measures the total number of classes outside a package that depend upon classes within that package. This metric is highly related with portability, because packages with higher CA are harder to be replaced since they have a lot of other packages that depend upon them.

Efferent couplings (CE) metric measures the total number of classes inside a package that depend upon classes outside this package. High CE value will negatively influence package re-usability, since it is harder to understand and isolate all the components necessary to reuse the package. CE negatively influences package maintainability since packages with high CE are prone to changes from the packages it depends on. It also negatively influences portability since packages with high CE are hard to be adapted because they are hard to understand.

Instability (I) metric measures the ratio between CE metric and the total number between the CE and CA metric. Basically, packages with many efferent couplings are more unstable, because they are prone to changes from other packages. So, instability negatively influences re-usability, maintainability and portability. On the other hand, packages with many afferent couplings are responsible for many other packages, making them harder to change and therefore more stable.

Abstractness (A) metric measures the ratio between the number of abstract classes or interfaces and the total number of classes inside a package. Stable packages have to be abstract so that they can be extended without being changed. On the other hand, highly unstable packages must be concrete, because its classes have to implement the interfaces inherited from stable packages.

Distance from the main sequence (D) metric measures the perpendicular distance of a package from the *main sequence*. Because not all packages can be totally abstract and stable or totally concrete and unstable, these packages have to balance the number of concrete and abstract classes in proportion to there efferent and afferent couplings. The

main sequence is a line segment that joins points (0,1) (representing total abstractness) and (1,0) (representing total instability). This line represents all the packages whose abstractness and stability are balanced. So it is desirable that packages are the closest to the main sequence as possible.

3.3.3 Metrics for Object-Oriented Design Suite

The MOOD metrics set was proposed by Fernando Brito e Abreu [14, 21] and focuses most on measuring key characteristics of the object-oriented paradigm like inheritance, encapsulation and coupling.

Polymorphism Factor (PF) metric measures the ratio of the total number of overriding methods to the total number of possible overridden methods in the software system. Given M(C_i) the set of methods in a class C_i and DC(C_i) the set of subclasses of a class C_i, the total number of possible overridden methods can be calculated using the following formula:

$$V = \sum_{i=1}^{n}[|M(C_i)| * |DC(C_i)|]$$

Coupling Factor (CF) metric measures the ratio between the the actual number of non-inheritance couplings and the total number of possible non-inheritance couplings in the software system. The CF metric negatively influences the quality factors maintainability, portability and re-usability because of the same reasons listed in the CBO metric.

Method Hiding Factor (MHF) metric gives the ratio between the total number of hidden methods and the total number of methods in a software system. This metric was proposed as a form of measuring the encapsulation level.

Attribute Hiding Factor (AHF) metric gives the ratio between the total number of hidden attributes (instance variables) and the total number of attributes in a software system. This metric was also proposed as a form of measuring the encapsulation level.

Method Inheritance Factor (MIF) metric measures the ratio between the total number of inherited methods and the total number of methods in a software system. This metric was proposed as a mean of measuring inheritance that, like in the case of the

DIT and NOC metrics, benefits re-usability however affects analyzability and testability (sub-factors of maintainability).

Attribute Inheritance Factor (AIF) metric gives the ratio between the total number of inherited attributes (instance variables) and the total number of attributes in a software system. This metric was proposed for the same reasons listed in the MIF metric.

3.3.4 Lorenz & Kidd Metric Suite

The L&K Metrics developed by Mark Lorenz and Jeff Kidd [21] and it is formed by basic and direct metrics like the number of public methods in a class (NPC), the number of public, private and protected methods in a class (NM), the number of public instance variables of a class (NPV), the number of public, private and protected instance variables of a class (NV), the number of methods inherited by a subclass (NMI). However they also proposed more complex metrics like:

Number of Methods Overridden by a subclass (NMO) metric gives the total number of overridden methods of a class. Classes with a high number of overridden methods probably are wrongly connected to its superclasses, leading to a design problem.

Number of Methods Added by a subclass (NMA) metric gives the total number of new methods of a subclass. This metric strengthens the idea expressed in the NMO metric.

Average Method Size (AMS) metric gives the total number of source lines in a class divided by the number of its methods. This metric gives an idea of a class size.

Number of Friends of a class (NFC) metric is similar to the CBO metric, it gives the number of other classes coupled to a class.

3.4 Software Metrics Thresholds

Over time many authors proposed software metric thresholds based on their experience. For example, NASA Independent Verification & Validation (IV&V) Facility metrics data program[1] collects, organizes and stores software metrics data. Its website gives

[1] http://mdp.ivv.nasa.gov/index.html

general users access to a repository with information about various metrics used in NASA projects like threshold values, scales of measurement, range and usage.

NASA IV&V puts the LOC threshold (including blank lines, comment lines, and source code lines), used at method level for NASA software projects, around 200. High cyclomatic complexity leads to the loss of software quality. In the NASA IV&V, a method with a cyclomatic complexity value of over 10 is considered difficult to test and maintain.

However, since these thresholds rely on experience, it is difficult to reproduce or generalize these results[1]. There are some authors who propose methodologies based on empirical studies for determining software metrics thresholds. Below we describe some of these methods.

Erni et al. [15] propose a simple methodology based on the use of well-known statistical methods to determine software metrics thresholds. The lower (T_{min}) and the higher (T_{max}) thresholds are calculated using the following formulas $T_{min} = \mu - s$ and $T_{max} = \mu + s$, being μ the average of a software metric values in a project and s the standard deviation. The lower and the higher thresholds work as lower and upper limit for the metric values.

Shatnawi et al. [55] propose a methodology based on the use of Receiver-Operating Characteristic (ROC) curves to determine software metrics thresholds capable of predicting the existence different categories of errors. This method was experimented in three different releases of Eclipse and using the C&K metrics.

Alves et al. [1] propose a novel methodology for deriving software metric threshold values from measurement data collected from a benchmark of software systems. It is repeatable, transparent and straightforward method that extracts and aggregates metric values for each entity (packages, classes or methods) from all software systems in the benchmark. Metric thresholds are then derived by choosing the percentage of the overall code one wants to represent.

3.5 Software Quality Evaluation Process

Although the standard ISO/IEC 9126 defines a quality model, quality factors, subfactors and measures to determine the quality of a software product, it does not define the process for evaluating its quality.

This is why the International Organization for Standardization released in 1998 the standard ISO/IEC 14598 [57]. This standard defines a process for measuring and as-

sessing the quality of software products and it is based on three different perspectives: development, acquisition and independent evaluation. The most recent version of this standard is divided in six parts: *general overview* [27], *planning and management* [29], *process for developers* [30], *process for acquires* [28], *process for evaluators* [26], and *documentation and evaluation modules* [31].

3.5.1 Process for Developers

The process defined in ISO/IEC 14598 for developers is divided in five stages:

Organization. In this first stage of the process aspects related with development and support have to be defined. Aspects like definition of organizational and improvement objectives, identification of technologies, assignment of responsibilities, identification and implementation of evaluation techniques for developed and third-party software products, technology transfer and training, data collection and tools to be used. This will contribute to the quality system and to establish a measurement plan.

Project planning and quality requirements. In this stage, the development life cycle of the software product is established and documented. It is necessary to check the quality model defined in ISO/IEC 9126 for any conflicts that may exist and whether the quality factors and metrics are complementary and verifiable. It is also important to verify if they are feasible, reasonable and achievable by taking into account, for example, the given budget and time schedules.

Specifications. In this stage, the internal and external quality factors are mapped to the requirements specification which contains the complete description of the behaviour of the software product that will be developed. It is also in this stage that the metric scales and thresholds are defined.

Design and planning. When doing the design planning it is important to define schedules, delineate responsibilities, and determine tools, databases and any specialist training. In this stage, it is important to specify measurement precision and statistical modelling techniques. It is also important to try to understand how the metrics results will influence development. Therefore, the need for contingency actions, additional review and testing, and improvement opportunities should all be considered in the design planning.

Build and test. In this last stage, it is where the metrics results are collected, and decisions are made based on the the analysis of the results. In the coding phase, internal metrics are applied, in the testing phase, external metrics are used. Therefore the conclusions drawn from analysis of the metrics results must also appear in the design reviews and testing plans. This allows to have an overall image of the quality of the software product at all stages of development.

At the end of the project, a review of the whole process of measurement collection should be made in order to understand what went well and what can be improved in future projects.

3.5.2 Process for Acquires

Acquires can be seen as companies who purchase complete software packages, companies who have part of their development activity done by a third party, or companies who want to use specific tools. The process defined in ISO/IEC 14598 for acquires is also divided in four stages:

Establishment of requirements. In this first stage, it is necessary to define what is the scope of the evaluation. The quality model defined in ISO/IEC 9126 can be used to determine the quality factors that affect the quality of the software product, but can also be defined other factors like cost or regulatory compliance.

Evaluation specification. At this stage, an specification of the evaluation is drawn by analysing the software product so its key components can be identified. To each component, quality-in-use and external metrics are specified in order to evaluate the quality factors established in the previous stage. For each metric it is defined the level of priority and thresholds. The methods for measurement collection and analysis are also documented.

Evaluation design. Establishing an evaluation plan can be difficult, because it depends on the type of software product under evaluation. For example, it is possible to evaluate a project still under development at various stages of its life-cycle and have access various types of data, whereas an off-the-shelf software product is more difficult to evaluate. Non the less, it is necessary to establish an evaluation plan, and for this it must be taken into account the need to access the software product's documentation, development tools and personnel, evaluation schedules, contingency arrangements, key milestones,

criteria for evaluation decisions, reporting methods and tools, procedures for validation and standardization over future projects, in order to make the most complete evaluation possible. The ISO/IEC 14598 provides the necessary information and support material to make create this evaluation plan.

Evaluation execution. At the end, the evaluation needs to be recorded. This could be anything from a simple logbook, to a full report that contains the results, the analysis, decision records, problems encountered, measurement limitations, any compromises made in relation the original objectives, and conclusions about the evaluation results obtained and the methods used.

3.5.3 Process for Evaluators

The main objective of an evaluator is to assess software products in an impartial and objective way, so that results of an evaluation can be always reproduce by using the same measurement criteria. The process defined in ISO/IEC 14598 for evaluators is also divided in four stages:

Evaluation requirements. In the first stage, like in the process for acquires, it is necessary to define what is the scope of the evaluation. The quality model defined in ISO/IEC 9126 can be used to determine the quality factors that affect the quality of the software product, but it can also be defined other factors like cost or regulatory compliance.

Evaluation specification. In this stage, an specification of the evaluation has to be drawn. This is done by analysing the software product and identifying its key components. To each component the metrics used to evaluate the quality factors established in the previous stage are specified. The specification is basically formed by the formal specification of each metric and the instructions on how to report the results, and a formal description of each component and the quality factors used to evaluate them.

Evaluation plan and evaluation modules. At this point an evaluation plan must be created. It is necessary to document the evaluation methods used to implement the metrics defined in the previous stage. Then the plan must be optimized by relating the evaluation methods to the elements (metrics and components) in the evaluation specification. This elements are already related to the quality factors chosen in the first stage. It is also necessary to define evaluation activities by taking into account available

human resources and components to evaluate. The ISO/IEC 14598 provides evaluation modules in order to create reports in a consistent and repeatable format.

Evaluation results. In this last stage, all the evaluation activities defined in the evaluation plan are executed. After the evaluation, reports are elaborated and results are documented.

3.6 Software Metrics Tools

Nowadays, we can easily find tools capable of measuring all the software metrics mentioned previously. These tools range from the simple command line tool that only outputs numerical results, to the more complete tool with graphical user interface that displays the results using graphs, in order to optimize the information that is passed to the user. Within this type of tools, there are also those that are only capable of calculating one or two simple metrics and tools capable of measuring more than 20 software metrics.

The remainder of this chapter will be devoted to a survey on software metric tools, where it is used Freecss[2], an open source chat server written in Java, as an example.

3.6.1 CyVis

CyVis[3] measures the complexity of Java programs by using simple source code metrics. It can graphically display the results obtained and also generate reports, in Html or Text format. The set of metrics used by CyVis is measured by gathering data from the project class files (bytecode). These metrics are divided in three levels, project, package and class level: Number of Packages (NOP); Number of Classes (NC); Number of Methods (NOM); Class Size (CS) (instruction count); Method Size (MS) (instruction count); and McCabe Cyclomatic Complexity (VG).

Demonstration

The results from the example FreeCs, obtained using CyVis, can be seen in Figure 3.2. More specifically, the results of its *TrafficMonitor* class.

[2]http://freecs.sourceforge.net/
[3]http://cyvis.sourceforge.net/

All the seven methods from class *TrafficMonitor* are represented in the bar chart. The size of each bar changes depending on the size of the method it represents. And its position varies depending on the complexity of the method. The ones on top have higher complexity.

The greater the complexity of a method, the harder it is to understand and test it, therefore, each bar is coloured with three possible colours, green, yellow or red, as a way of warning.

As can be seen in the results table from Figure 1, method *run* is the largest with a instruction count value of one hundred twenty one and the most complex with a complexity value of twelve.

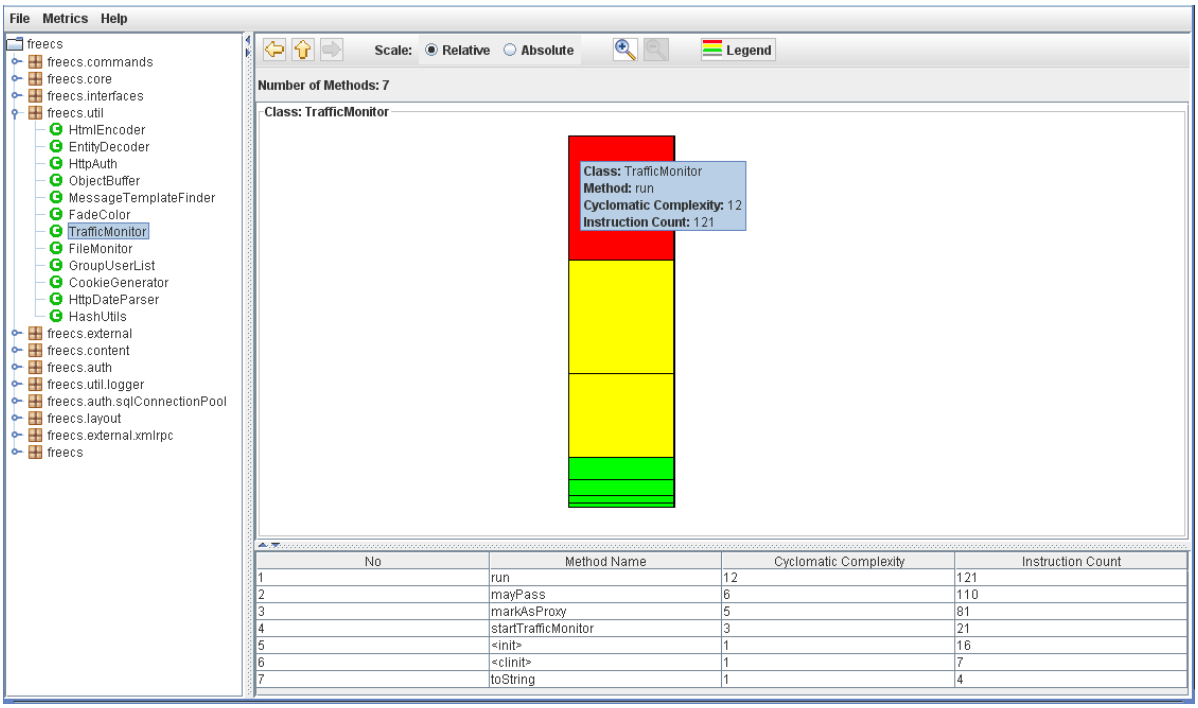


Figure 3.1: FreeCS example in CyVis

3.6.2 JavaNCSS

JavaNCSS[4] is a simple command line tool for analysing Java programs. It does this by measuring two of the most known and used source code metrics, non commented source statements (NCSS) and cyclomatic complexity (CC).

[4]http://www.kclee.de/clemens/java/javancss/

The definition of statement in JavaNCSS is broader than in the Java language specification. Besides the normal statements, like *if*, *while*, *break*, *return*, *synchronized*, it also considers as statements the *package*, *import*, *class*, *interface*, *field*, *method*, *constructor* declarations and the *constructor* invocation.

JavaNCSS also counts the number of packages, classes, methods, and Javadoc comments (JVDC) per classes and methods.

All the results can also be displayed in a simple graphical user interface, or in a generated XML file.

Demonstration

As can be seen in Figure 3.3, JavaNCSS presents the results from the example FreeCs by using a simple graphic interface with no type of charts and just using numerical values.

For example, in the case of the *TrafficMonitor* class, it returns a list of all the methods in the class with information regarding the NCSS, the CC and the number of JVDCs. It can be seen that method *run* is the biggest and most complex method, in class *TrafficMonitor*, with a NCSS value of twenty seven, a CC value of twelve and one JVDC. In the end, JavaNCSS also gives an average value of the NCSS, the CC and the number of JVDC per method.

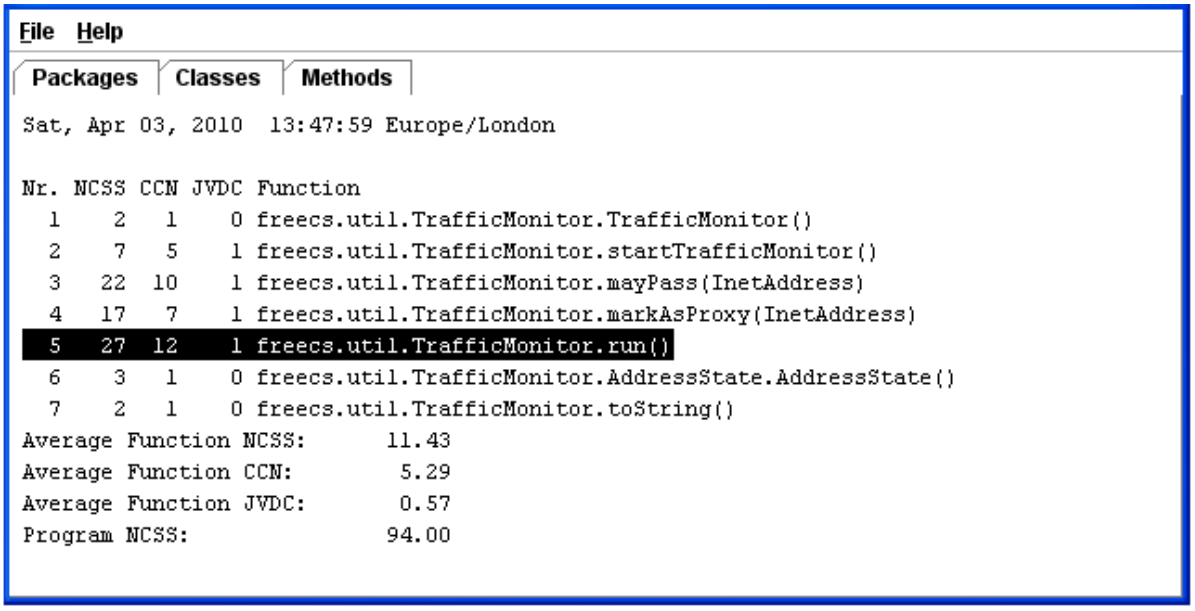


Figure 3.2: FreeCS example in JavaNCSS

3.6.3 JDepend

JDepend[5] is a tool that generates design quality metrics for each Java package in a Project. The design quality is evaluated based in its extensibility, re-usability, and maintainability. This is done by using the following design quality metrics: Number of Classes and Interfaces (CC); Number of Abstract Classes (and Interfaces) (AC); Afferent Couplings (CA); Efferent Couplings (CE); Abstractness (A); Instability (I); Distance from the Main Sequence (D); and Package Dependency Cycles (Cyclic).

JDepend can be used to analyse package abstractness, stability and dependencies, with the objective of identifying and inverting dependencies between high-abstraction stable packages and low-abstraction instable packages. This makes packages with high-abstraction level reusable, easy to maintain and extensible to new implementations. It can also be used to identify package dependency cycles that negatively influence the re-usability of packages involved in these cycles.

One nice feature of JDepend, is that it can be used with JUnit[6], a framework for writing and running repeatable tests for Java. Tests can be written to automatically check that metrics are in conformance with desired result, or, written to fail if a package dependency, other than the ones declared in a dependency constraint, is detected. Package dependency cycles can also be checked using JUnit tests.

All the results can be displayed in a simple graphical user interface, or by generating a text file or a XML file.

Demonstration

In Figure 3.4, it can be seen a example of JDepend graphical user interface. It is divided in two parts that represent the afferent and efferent couplings of each package, in the FreeCs example.

For each package, it is displayed all the metric results, obtained. By clicking , for example, on the *freecs.util* package, it opens a list containing the packages it depends upon (in the efferent coupling section), or the list of packages that use it (in the afferent coupling section).

Focusing on the *freecs.util* example, this package is formed by thirteen classes, it is totally concrete ($A = 0$) and it is either stable, or instable ($I = 0.55$), witch makes this package very difficult to manage. Since concrete packages are more affected by changes made to their afferent couplings, they should be instable. Therefore, one can

[5]http://clarkware.com/software/JDepend.html
[6]http://www.junit.org/

conclude that it is undesirable to add dependencies to the *freecs.util* package. However, *freecs*, *freecs.auth*, *freecs.sqlConnectionPool*, *freecs.commands*, *freecs.content*, *freecs.core*, *freecs.external*, *freecs.layout* and *freecs.util.logger* all depend upon *freecs.util* ($CA = 9$).

The *freecs.util* package also has, at least, one dependency cycle, flagged by the word *Cyclic* in the list of metric results. These cycles can be viewed by clicking on the packages that depened on *freecs.util* and drilling down its tree of efferent couplings. Yet, it is easier to viewed them by generating a text file with the results.

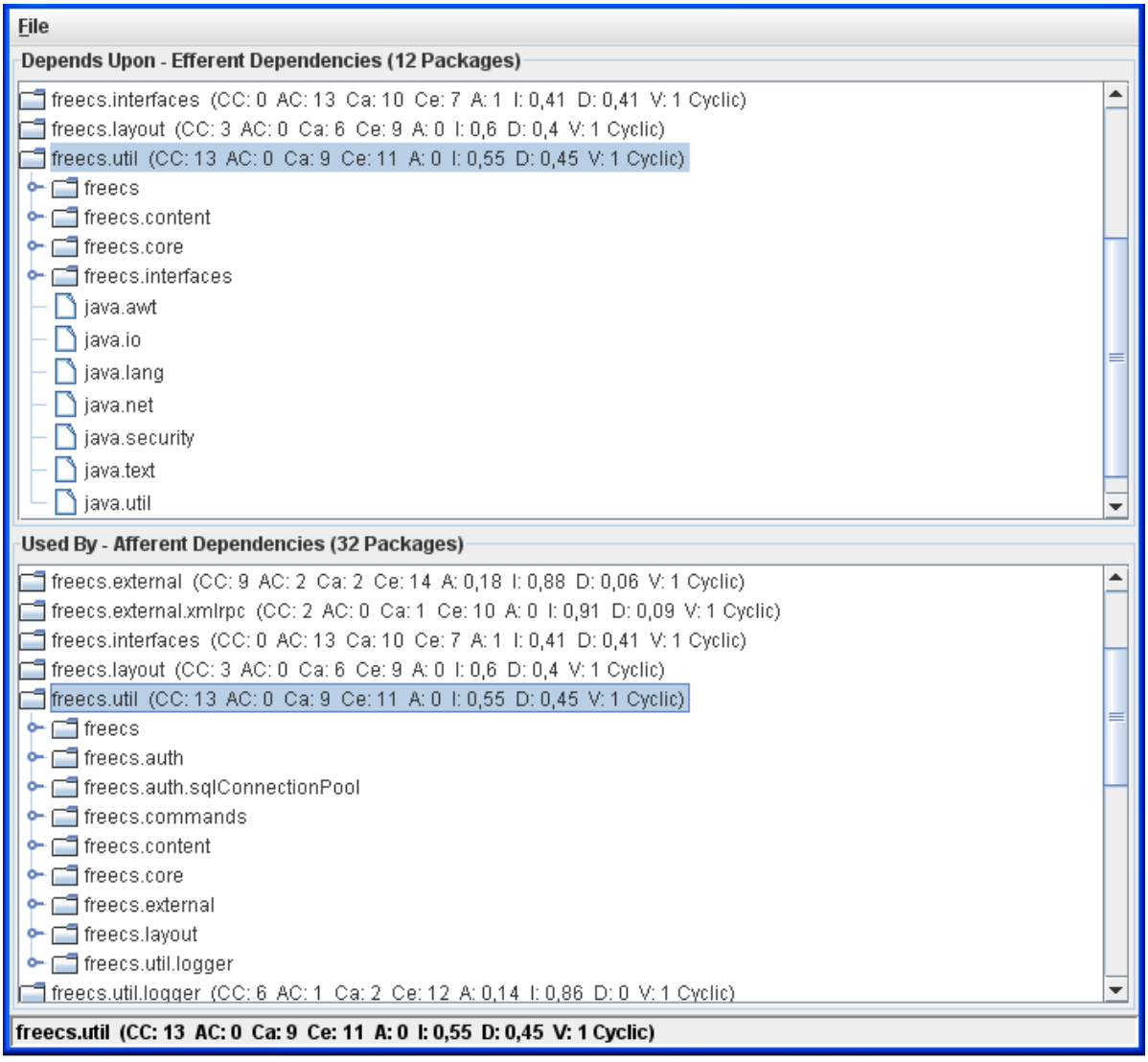


Figure 3.3: FreeCS example in JDepend

3.6.4 CKjm

CKjm is a simple command line tool that calculates Chidamber and Kemerer object-oriented metrics: Weighted methods per class (WMC); Depth of Inheritance Tree (DIT); Number of Children (NOC); Coupling between object classes (CBO); Response for a Class (RFC); Lack of cohesion in methods (LCOM); Afferent coupling (CA); Number of Public methods for a class (NPM)

This tool neither has a graphical user interface, or generates output files with the results, it only calculates two metrics besides the Chidamber and Kemerer metrics, the NPM and CA metrics. However, it does this in a efficient and quick way.

To run this tool, one just has to specify the class files on its command line.

Demonstration

The measures obtained for the example FreeCs, more specifically the *freecs.util* package, can be seen in Figure 3.5.

For each class that forms *freecs.util*, CKjm presentes all the results from the eight metrics calculated in the following order: WMC, DIT, NOC, CBO, RFC, LCOM, CA, and NPM.

For example, the *TrafficMonitor* class obtained a seven WMC value witch means that it has seven methods (because CKjm assigns one complexity value to each method) in which, by the NPM result, five of them are public. By the DIT and NOC values, one learns that *TrafficMonitor* has two superclasses and zero subclasses. By the CBO and CA values, one learns that this class has two classes that use it and one class that it depends upon. From the RFC value, one understands that twenty nine different methods can be executed, when a object from *TrafficMonitor* receives a message (note that CKjm only calculates a rough estimation) and, from the LCOM value, one realizes that there are seven pairs of the class's methods that do not share a instance variable access.

3.6.5 Eclipse Plugin

This last tool is the most complete of them all, it is a open-source Eclipse plug-in[7] and calculates twenty three metrics at package, class and method level: Lines of Code (LOC);

[7] http://metrics.sourceforge.net/

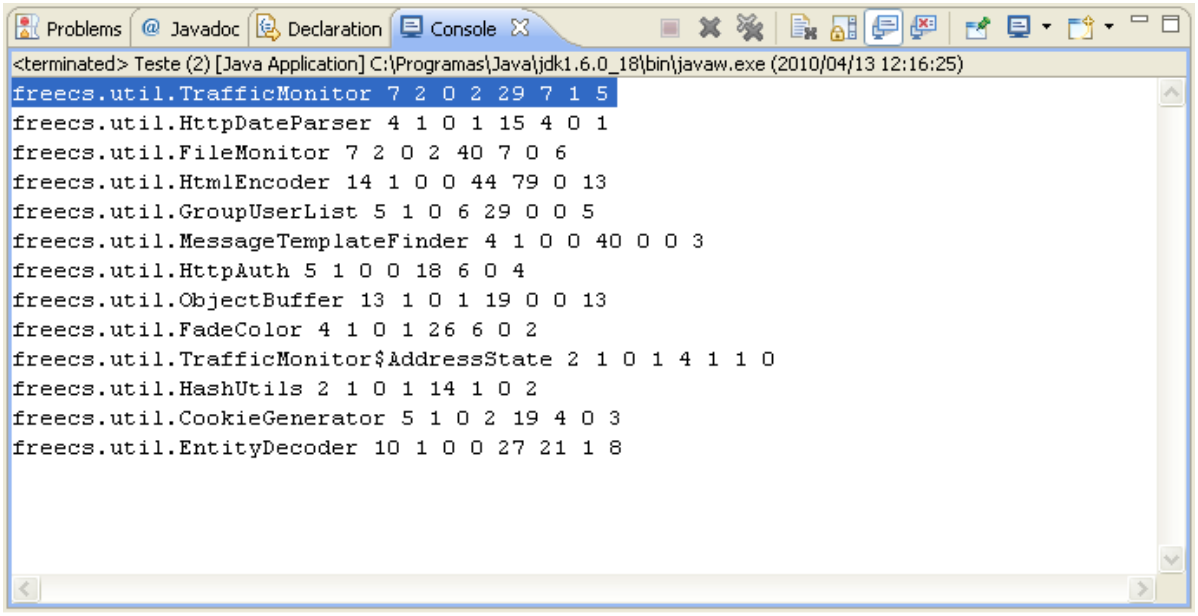


Figure 3.4: FreeCS example in CKjm

Number of Static Methods (NSM); Afferent Coupling (CA); Efferent Coupling (CE); Instability (I); Abstractness (A); Normalized Distance (D); Number of Attributes (NOF); Number of Packages (NOP); Number of Classes (NOC); Number of Methods (NOM); Method Lines of Code (MLOC); Number of Overridden Methods (NORM); Number of Static Attributes (NSF); Nested Block Depth (NBD); Number of Parameters (PAR); Number of Interfaces (NOI); Number of Children (NSC); Depth of Inheritance Tree (DIT); Lack of Cohesion of Methods (LCOM); Weighted Methods per Class (WMC); McCabe Cyclomatic Complexity (VG); Nested Block Depth (NBD)

Naturaly it has a graphical user interface where the metrics results are displayed. These can also be exported to a XML file. It has the capability to trigger a warning (indicating the package, class or method) whenever a metric threshold is violated. And these thresholds can be changed in the plugin preferences.

One of the nicest features of this eclipse plugin is the option of viewing the packages dependency Graph. This option can be used to identify dependency cycles between packages, since these are coloured red. In greater detail, one can also view the classes from those packages that are creating the dependencies.

Besides helping to identify package dependency cycles, it also has the option of finding the shortest path between two nodes that can be used to better understand the connection between all the packages in large dependency graphs.

Demonstration

Figure 3.6 shows the results from the FreeCs example. All the twenty three metrics are displayed along with its total results, mean values, standard deviations, maximum values and the resources that achieved the maximum values.

If one selects a metric it will display the results obtained for each package. If we continue selecting a package and continuing drilling down, it will display the results at the class level and method level.

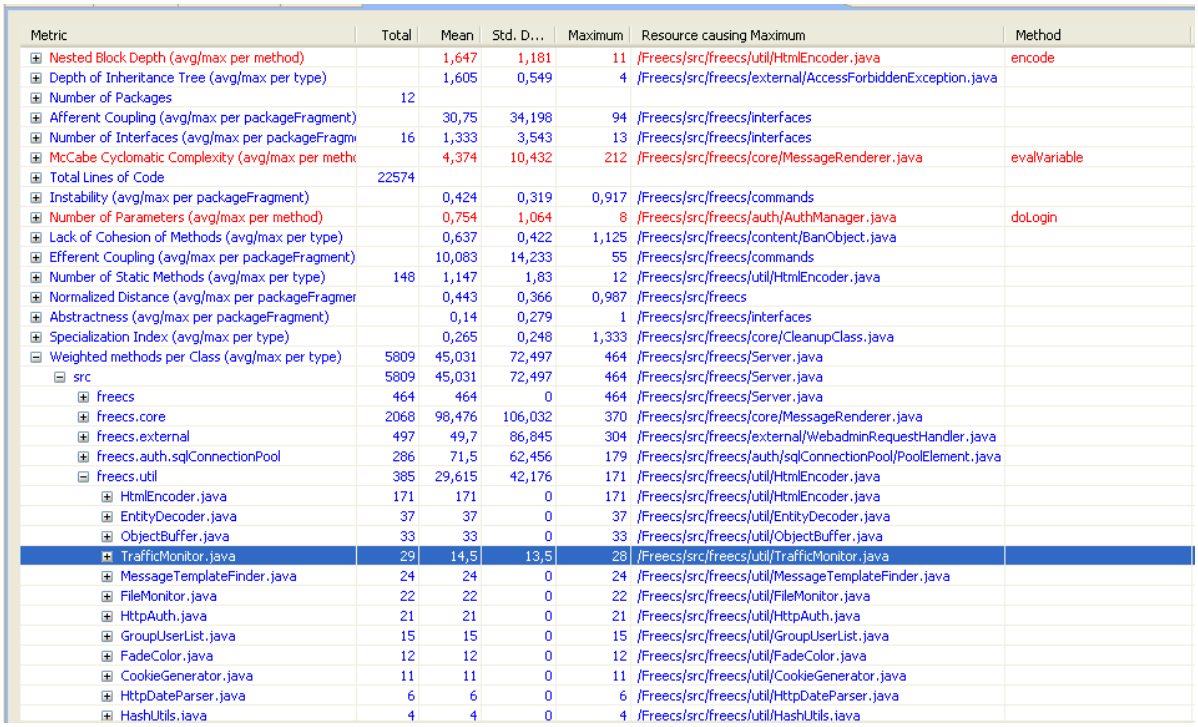
For example, the package *freecs.util* has a method with a maximum complexity value of 171 and it belongs to the class *HtmlEncoder*, which is very high for a method. In the case of the class *TrafficMonitor*, it has a WMC value of twenty nine, with a method achieving the maximum complexity value of twenty eight, witch is still a very high value. Note that the WMC value obtained by *TrafficMonitor* differs from the result obtained with JDepend, because the complexity of a method is calculated using the McCabe cyclomatic complexity metric.

It is also possible to see that the thresholds from the metrics, nested block depth, McCabe cyclomatic complexity and number of parameters where violated, because they are coloured red. However, by drilling down the levels (package, class, method), one can pin point the origin of the warnings.

3.6.6 Survey Results

Almost all of the tools in the survey have some kind of GUI to better present the results obtained and also have the option of generating an output file to facilitate integration with other tools. However the most important thing is that they are all capable of measuring different software metrics, more specifically OOD metrics, and thus complement themselves.

Obviously, as can be seen in Table 3.1, the tool that stood out the most was the Eclipse Plugin, because of its ability to measure all the metrics that were referenced in this chapter. And, because it is a plugin for a integrated development environment (IDE), it allows the user to get immediate feedback whenever there is any alteration in the source code. Having a GUI with the option for generating and viewing packages and classes dependency graphs is also a nice feature.



Metric	Total	Mean	Std. D...	Maximum	Resource causing Maximum	Method
Nested Block Depth (avg/max per method)		1,647	1,181	11	/Freecs/src/freecs/util/HtmlEncoder.java	encode
Depth of Inheritance Tree (avg/max per type)		1,605	0,549	4	/Freecs/src/freecs/external/AccessForbiddenException.java	
Number of Packages	12					
Afferent Coupling (avg/max per packageFragment)		30,75	34,198	94	/Freecs/src/freecs/interfaces	
Number of Interfaces (avg/max per packageFragm	16	1,333	3,543	13	/Freecs/src/freecs/interfaces	
McCabe Cyclomatic Complexity (avg/max per meth		4,374	10,432	212	/Freecs/src/freecs/core/MessageRenderer.java	evalVariable
Total Lines of Code	22574					
Instability (avg/max per packageFragment)		0,424	0,319	0,917	/Freecs/src/freecs/commands	
Number of Parameters (avg/max per method)		0,754	1,064	8	/Freecs/src/freecs/auth/AuthManager.java	doLogin
Lack of Cohesion of Methods (avg/max per type)		0,637	0,422	1,125	/Freecs/src/freecs/content/BanObject.java	
Efferent Coupling (avg/max per packageFragment)		10,083	14,233	55	/Freecs/src/freecs/commands	
Number of Static Methods (avg/max per type)	148	1,147	1,83	12	/Freecs/src/freecs/util/HtmlEncoder.java	
Normalized Distance (avg/max per packageFragmer		0,443	0,366	0,987	/Freecs/src/freecs	
Abstractness (avg/max per packageFragment)		0,14	0,279	1	/Freecs/src/freecs/interfaces	
Specialization Index (avg/max per type)		0,265	0,248	1,333	/Freecs/src/freecs/core/CleanupClass.java	
Weighted methods per Class (avg/max per type)	5809	45,031	72,497	464	/Freecs/src/freecs/Server.java	
src	5809	45,031	72,497	464	/Freecs/src/freecs/Server.java	
freecs	464	464	0	464	/Freecs/src/freecs/Server.java	
freecs.core	2068	98,476	106,032	370	/Freecs/src/freecs/core/MessageRenderer.java	
freecs.external	497	49,7	86,845	304	/Freecs/src/freecs/external/WebadminRequestHandler.java	
freecs.auth.sqlConnectionPool	286	71,5	62,456	179	/Freecs/src/freecs/auth/sqlConnectionPool/PoolElement.java	
freecs.util	385	29,615	42,176	171	/Freecs/src/freecs/util/HtmlEncoder.java	
HtmlEncoder.java	171	171	0	171	/Freecs/src/freecs/util/HtmlEncoder.java	
EntityDecoder.java	37	37	0	37	/Freecs/src/freecs/util/EntityDecoder.java	
ObjectBuffer.java	33	33	0	33	/Freecs/src/freecs/util/ObjectBuffer.java	
TrafficMonitor.java	29	14,5	13,5	28	/Freecs/src/freecs/util/TrafficMonitor.java	
MessageTemplateFinder.java	24	24	0	24	/Freecs/src/freecs/util/MessageTemplateFinder.java	
FileMonitor.java	22	22	0	22	/Freecs/src/freecs/util/FileMonitor.java	
HttpAuth.java	21	21	0	21	/Freecs/src/freecs/util/HttpAuth.java	
GroupUserList.java	15	15	0	15	/Freecs/src/freecs/util/GroupUserList.java	
FadeColor.java	12	12	0	12	/Freecs/src/freecs/util/FadeColor.java	
CookieGenerator.java	11	11	0	11	/Freecs/src/freecs/util/CookieGenerator.java	
HttpDateParser.java	6	6	0	6	/Freecs/src/freecs/util/HttpDateParser.java	
HashUtils.java	4	4	0	4	/Freecs/src/freecs/util/HashUtils.java	

Figure 3.5: FreeCS example in Eclipse Plugin

Table 3.1: Tools results

	N° of Metrics	OOD Metrics	GUI	XML/Txt Output
Eclipse Plugin	23	20	YES	YES
JDepend	8	8	YES	YES
CKjm	8	8	NO	NO
CyVis	6	3	YES	YES
JavaNCSS	6	3	YES	YES

4 Complementing Software Metrics Information

Although software quality metrics are the most direct way to analyse the various factors that constitute the quality model of a software product., decisions should not be solely based on the information obtained by metrics, because software metrics are not enough to evaluate all aspects of a software product's source code (design, architecture, complexity, coding rules, tests).

Next we present two of the most widely used techniques to analyse and improve the quality of a software product and also complement the results obtained through software quality metrics: unit testing and static analysis.

There are other techniques, like model checking, that are used by tools like Java Pathfinder[1]. This tool is an extensible virtual machine framework that verifies Java programs bytecode for violations of properties like deadlocks and race conditions. However we think this technique does not enter the scope of this thesis, so we will not going to include it in this chapter.

4.1 Unit Testing

Testing plays a major role in software development, it is used to verify software's behaviour and find possible bugs. Testing can be used to measure and improve the quality factors of a software product (like reliability and maintainability) since de earlier stages of development, so it has a big impact on the quality of the final product [43, 22].

In *The Art of Unit Testing*, by Roy Osherove [52], *unit testing* is defined as being an "automated piece of code that invokes the method or class being tested and then checks some assumptions about the logical behaviour of that method or class. A unit test is almost always written using a unit testing framework. It can be written easily and runs quickly. It is fully automated, trustworthy, readable, and maintainable". This concept

[1] http://babelfish.arc.nasa.gov/trac/jpf/wiki

first appeared for the programming language Smalltalk by the hand of Kent Beck, and later spread to almost every known programming language, making unit testing one of the best techniques to improve code quality while learning about the functionality of the class and its methods. Unit testing is now used in several popular software development methodologies like extreme programming, test driven development, continuous testing, and efficient test prioritization and selection techniques.

4.1.1 Different Types of Tests

Unit testing is used to test software modules that later will be combined and tested as a group, in the *integration testing* phase. This occurs because each module may work correctly during unit testing phase, but it may fail when interacting with other modules. After the integration testing phase, it comes the *system tests* that group all the modules together and test the software's functionality from the user's point of view and determines the readiness of a software product for release. The process of testing finishes with the *acceptance tests* as a way of validating the costumer acceptance [56].

However using integration occupy much of the effort of testing a software product. It is normal for a software project to have 50% to 70% of the development effort spent on testing, and 50% to 70% of the testing effort on integration testing [58]. This is why unit testing is so important, especially in the earlier phases of the development process, when bugs are smaller and easier to find, thus reducing the effort in the integration stage.

4.1.2 Java Unit Testing Framework (JUnit)

Normally a unit testing framework is known as a XUnit framework, being X the name of the language for which the framework was developed. Today there are several unit-testing frameworks like JUnit for Java, CppUnit for C++, NUnit for .NET, PyUnit for Python, SUnit for Smalltalk, and HUnit for Haskell [20].

Unit testing frameworks are libraries and modules that help developers create unit test, they provide a test runner (in the form of a console or a GUI) that lists the created unit tests, runs the tests automatically and indicates the tests status while running. Test runners will usually provide information such as how many tests ran, which tests and reason they failed, the code location that failed. It is also possible to create *test suites* that are basically collections of test cases.

JUnit[2] is an open source unit testing framework for Java that was developed by Kent

[2]http://www.junit.org/

Beck and Erich Gamma. This framework is based one a similar framework for Smalltalk, named SUnit. The JUnit framework has become the reference tool for Java unit testing, this is proven by the large number of introductory and advanced articles, and the large number of Open-Source projects that use it.

Creating a unit test with JUnit

The example used is a very simple project with three classes; class *Game* which consists of a list of players and teams where players will be assigned randomly, through the method *generate()*; class *Team* is formed by an *Integer n* that identifies the team and the list of players that will play for team n; and class *players* which consists of a *String* that represents the name of the player and an *Integer* that represents its number. Below, class *Player* is shown:

```
public class Player {

    private String name;
    private int number;

    public Player(String name, int number){
        this.setName(name);
        this.setNumber(number);
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public int getNumber() {
        return number;
    }

    @Override
    public String toString() {
        return "Player " + name + " nº " + number;
    }
}
```

As can be seen, class *Player* is formed by the method constructor, the getters and the setters. This class also has implemented the method *toString()* that returns the *String "Player " + name + " nº " + number* containing the name and number of the player. It is for this method that we will create our first test:

```
public class PlayerTests{

    @Test
    public void testToStringPlayer(){

        Player player = new Player("John",1);

        //assertTrue(
        // player.toString().equals("Player John nº 1")
        //);

        assertEquals(
            "Player John nº 1",
            player.toString()
        );
    }

}
```

Normally when creating unit tests, a separate test class is created for each class that is tested and for each method of this class at least one test method is created. The unit test structure usually is divided in three parts, the creation of objects to perform the test, executing the method one wants to test and verify (assert) that everything occurred as expected. As can be seen in the example, in order to create the test class *PlayerTests* each test method has to be signalled with the annotation *Test*, like in the case of the method *testToStringPlayer()*. However the most important part of a test is the *assertTrue()* method. These assertions (*assertTrue()*, *assertEquals()*, *assertNull()*, *assertFalse()*, etc) are static methods belonging to JUnit's class *Assert* that receive one or two parameters and verifies a boolean expression, if it is *false* the test is considered a failure and a message is returned, otherwise the test case continues normally. In the *TestToStringPlayer* example there are two ways of using assertions to verify that *toString()* returns *String* "Player John nº 1", as expected. The difference between the two is that the failure message returned by *assertEquals()* is more specific than the one returned by *assertTrue()*.

To run the unit test just use a IDE like Eclipse and run *PlayerTests* as a JUnit Test. Instead of the Package Explorer, the JUnit window should open and display all the information related to the execution of the test, as seen in Figure 4.1.

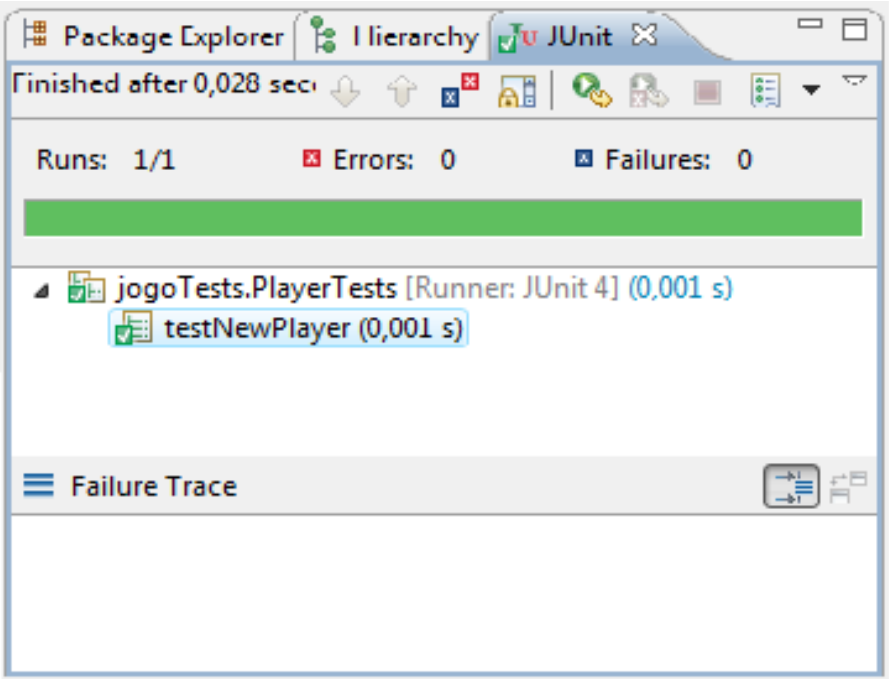


Figure 4.1: Running *testToStringPlayer* with JUnit in Eclipse

Setup and teardown methods

As more and more unit tests are being created for the same test case, one begins to see that the same (or similar) state is always created in the beginning of each test, this is known as a test *fixture*. With JUnit, it is possible to create for each test case the special methods *setUp()* that is executed before each test method, and *teardown()* that is executed after each test. These methods can be used to create an instance of the fixture, for each test and in the end of the test, destroy the instance. This ensures that the changes made to the fixture by previous tests does not influence the ones that have not yet been executed, thus creating more independent tests. This also allows to decrease the percentage of duplicated code. Consider the follw example:

```
private Player player;

   public class PlayerTests {

      @Before
      public void setUp() throws Exception {
         Player player = new Player("John",1);
      }

      @Test
      public void testToStringPlayer(){
```

```
        //assertTrue(
        // "Strings equal",
        // player.toString().equals("Player John nº 1")
        //);

        assertEquals(
           "Strings equal",
           "Player John nº 1",
           player.toString()
        );

     }

     @After
     public void tearDown() throws Exception {
        Player player = null;
     }

  }
```

In this case it can be seen the use of methods *setUp()* and *teardown()* for the example *tesToStringPlayer*. It is also worth noting that for every assertion it was defined a failure message *Strings equal*, in order to describe the failure in a more user friendly way.

Testing exceptions

Not all tests are used to verify if every thing is running according to plan, it is necessary to also test worst case scenarios. With JUnit, it is possible to test for exceptions thrown by the tested methods. Consider the follow example:

```
private Player player;

   public class PlayerTests{

      @Before
      public void setUp() throws Exception {
         Player player = null;
      }

      @Test(expected=NullPointerException.class)
      public void testToStringPlayer(){

         boolean test = player.toString().equals("Player John nº 1");

      }

      @After
      public void tearDown() throws Exception {
         Player player = null;
```

```
    }

}
```

As can be seen in this example, *testToStringPlayer()* throws a *NullPointerException* because it is trying to compare 2 *Strings* and variable *player* is initialized to *null*. However this test will pass because test *testToStringPlayer()* is expecting an exception as can be seen by the annotation *@Test(expected=Null...)*.

4.1.3 Stubs and Mock Objects

An external dependency is an object in the software that the method under test interacts with, but does not have control over it, like for example file systems, threads, memory, time, and etc. It is important to be able to control external dependencies when creating unit tests and this controlling can be done by using techniques like stubs and mock objects.

A *stub* is a replacement for the existing external dependency in the software that can be controlled. By using a stub, unit tests can be done in isolation and the method under test still receives all the inputs it needs so that it can be executed in the test. Common techniques for using stubs involve the creation of an interface to allow replacing the external dependency for the stub. The interface is in a constructor, in a get (or a set) or just before the method is called, so it can be used in the tests.

Mock objects is a technique proposed by Mackinnon et al. [42]. The difference between mock objects and stubs although very small. Unlike stubs, mock objects can cause failures in tests. The method under test interacts with the mock object and later the assertion is made against the mock object (instead of the method under test) to see if the method interacted with the mock object as expected. The advantage of using mock objects is that it is possible to test at a finer level of granularity than with stubs.

4.1.4 Unit Tests and Code Coverage

Code coverage was defined by Miller and Maloney [49] in the 60's and can be described as the the percentage of source code that has been tested. As the main objective of testing is to find the greatest number of bugs, code coverage is the best way to know if more unit tests are necessary. It is possible to improve the quality of a software product by achieving the highest code coverage possible.

White-box testing [56] (also known as implementation-based or structural tests) is a

method of testing that uses the internal structure of a program to determine the test cases. Normally, the control flow or data flow analysis of a program is made in order to derive white-box tests. Opposed to *black-box testing* (also known as specification-based or functional tests) that is a method of deriving tests based on the functionality of a program.

Tests used to obtain code coverage fall in the category of white-box testing. Many code coverage metrics have been proposed [9] and the must commonly used are based on control flow analysis, as seen above:

Statement coverage

This measure is generated by determining the statements of the source code that were executed by a test suite. It is the weakest form of a code coverage metric and it is the one used by many commercial and open-source tools.

Decision (or branch) coverage

This measure gives the number of executed branches represented in the control flow graph of a program, during test. This is done by determining if the boolean expressions in control structures are evaluated to both true and false.

Condition coverage

This measures is similar to decision coverage, but instead of counting the results of every boolean expressions in the control structures of a program, it counts the outcome of each condition in the boolean expressions, during the execution of the test suite. A *condition* is an operand of a logical connective that does not contain logical connectives. For example:

```
decision:    if (min < X && X <= max)
             ...
         else
             ...

conditions: (min < X) and (X <= max)
```

Modified condition/decision coverage

This metric was developed for safety critical aviation software at Boeing by the RCTA/DO-178B. With this metric the number of necessary tests for full coverage is determined so

that every boolean expressions in the control structures of a program has taken all possible outcomes at least once, the same must happen for the conditions, and every condition in a boolean expression has to independently affect that decision's outcome at least once, during test. A condition independently affects a decision outcome by varying that condition while the other conditions in the expression are been holding fixed [12].

Multiple condition coverage

The number of unit test necessary with multiple condition coverage is determined by every possible combinations between the conditions of each boolean expressions in the control structures of a program. In other words the test cases required for full multiple condition coverage are given by the truth table of every boolean expression in the program.

Path coverage

In the case of path coverage, the number of tests are determined by the number of paths (every possible sequence of statements) in a program. This type of coverage is the strongest one based on control flow analysis, but normally is impossible to use this method because of infinite number of paths introduced by loop statements [51].

Tools for calculating code coverage

Today, there are many tools capable of measuring and reporting code coverage. Open-source tools like EMMA[3] and Cobertura[4], or commercial tools like Clover[5] determine the percentage of code accessed by unit tests through the use of code coverage metrics such as statement and branch coverage. In Figure 4.2 it can be seen that the percentage of code covered by *testToStringPlayer()* Only class *Player* was covered by the test (83.8%) resulting in only 25% of the code in the project. It can also be seen in class *Player* the statements that were executed by the test, the lines represented in red show that only the methods *getName()* and *getNumber()* were not executed during .

[3]http://emma.sourceforge.net/index.html
[4]http://cobertura.sourceforge.net/download.html
[5]http://www.atlassian.com/software/clover/

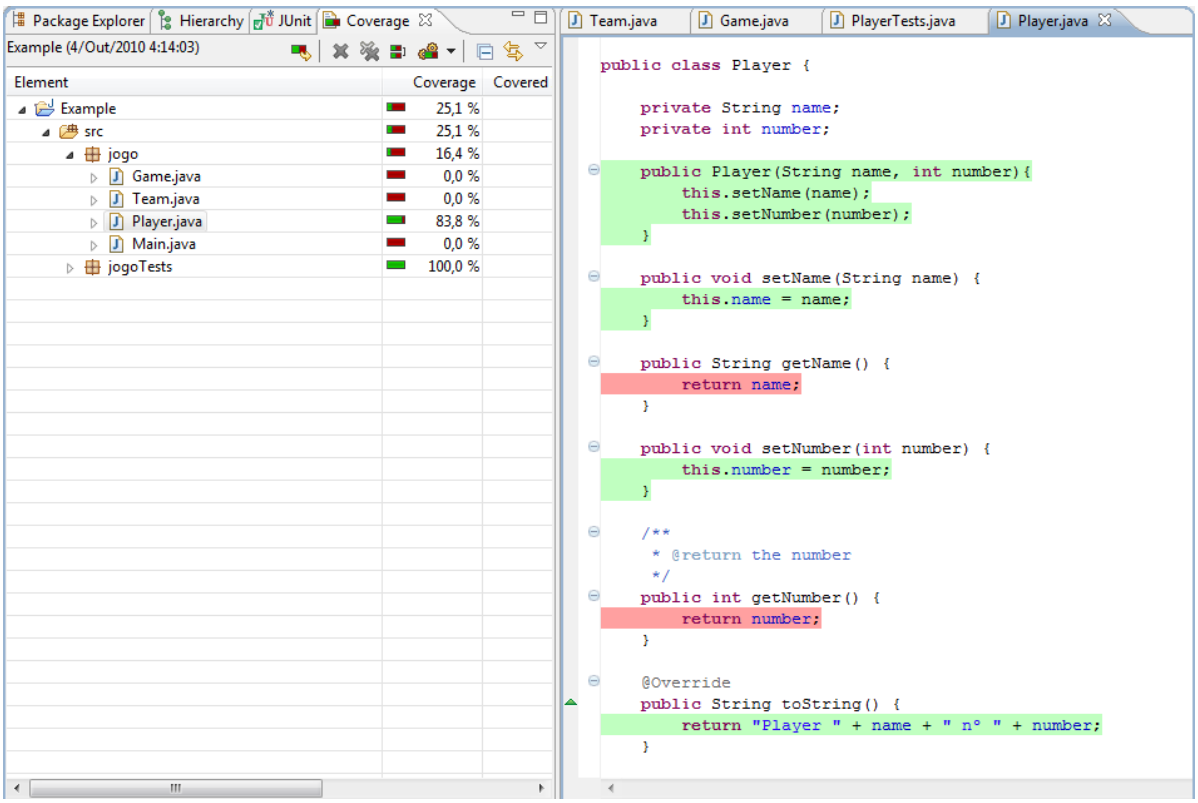


Figure 4.2: Running *testToStringPlayer* with EMMA in Eclipse

4.2 Static Analysis

Static code analysis consists in the verification of source code or object-code of a software product against bugs, without the hassle of having to executing it, as opposed to dynamic analysis.

The first of static code analysers represented by *Lint* appeared in the 70's and were difficult use and the number of bugs that could identify was very limited. Besides that the *noise* rate or the amount of irrelevant results (although correct) of these tools were extremely high. Nowadays, there are many fully automated static code analysers available that are capable of identifying various categories of bugs, these categories range from safety and security to bad programming practices and coding conventions and produce highly accurate results with few incorrect bug alerts also known as *false positives* and low noise rate [4]. To achieve this, static code analysers rely on techniques like semantic checking, strong type checking, memory allocation checking, logical statement checking, interface and include problem checking, security checking and metrics analysis [38].

Studies show that the number of bugs in software products can be reduced to up to

60% through the use of static analysis [38], and thus leading to the improvement of functionality (security) and reliability these software products [10, 54]. However, with the wide range of rules static analysers are capable of verifying, like for example bad programming practices or coding conventions, these tools can also be used to improve other quality factors like maintainability and re-usability.

Next three well known open source static analysis tools, Findbugs, PMD and Checkstyle, are presented. The example used to demonstrate these tools is the same used in the *unit testing* section. It is a very simple project with three classes; class *Game* which consists of a list of players and teams where players will be assigned randomly, through the method *generate()*; class *Team* is formed by an *Integer n* that identifies the team and the list of players that will play for team *n*; class *players* consists of a *String* that represents the name of the player and a *Integer* that represents its number. Some bugs were added to the example in order to demonstrate the usage of these tools.

4.2.1 FindBugs

Findbugs[6] [3] is an open-source static analyser that verifies Java source code against potential bugs.

It has an architecture based on plugins also known as bug detectors that are written in Java and are capable of verifying several different types of bugs. Currently, Findbugs is capable of finding nearly 300 bug patterns. These patterns are divided into various categories like correctness, bad practice, performance, security, internationalization and malicious code vulnerability. Each bug pattern is also determined a different priority, high, medium or low.

Findbugs can be executed as a plugin in an IDE like Eclipse or NetBeans, it can be used through the command line or it can be integrated into project management tools like Ant or Maven. The analysis results can also be saved in an XML file.

Using Findbugs

After running this tool over the example, 2 bugs were reported. The first one was found in class *Game*, in method *generate()* and it is related to the bug pattern *Method invokes inefficient new String() constructor*:

```
Random randomGenerator = new Random();
```

[6] http://findbugs.sourceforge.net/

```
String players = new String();

ArrayList<Player> aux = new ArrayList<Player>();
```

As can be seen, the string variable *players* is initialized by using the inefficient string no-argument constructor. This leads to the waste of memory because the new object is not recognise as being the constant "" and, in Java, identical string constants are represented by the same object. This leads to a loss of performance.

The second bug was also found in class *Game*, in method *generate()* and it is related to the bug pattern *Method concatenates strings using + in a loop*:

```
for(int i = 1; this.lineup1.size() < test/2; i++){

   int randomInt = randomGenerator.nextInt(test-i);

   this.lineup1.add(aux.get(randomInt));

   if(aux.get(randomInt).getName().equals("Mike") == true)
      players += aux.get(randomInt).getName();

   aux.remove(randomInt);

 }
```

In this case the string *players* is being constructed by using concatenation (+) inside the for loop. The problem is that strings in Java are immutable, so in each iteration a new string is in fact being created and this leads to a loss of performance. To circumvent this problem it is recommended the use of *StringBuffers*.

4.2.2 PMD

PMD[7] is another open source static analysis tool that scans java source code for potential bugs.

In these tool, bug patterns are described using Java or XPath expressions. Almost all of the patterns in PMD are related to bad programming practices like dead code, suboptimal code, overcomplicated expressions or duplicate code.

PMD can be executed as a plugin in an IDE like Emacs, BlueJ, Eclipse or NetBeans, it can be used through the command line or it can be integrated into project management tools like Ant or Maven. The analysis results can also be saved in an XML file.

[7]http://pmd.sourceforge.net/

Running PMD

After running this tool over the example, 27 bugs were reported. One them was related to the bug pattern *Simplify Boolean Expressions* and was found in class *Game*, in method *generate()*:

```
if(!aux.get(randomInt).getName().equals("Mike") == true)
    players += aux.get(randomInt).getName();
```

In this case, string *players* is concatenating the name of all players except the ones named "Mike". The problem is that the *if* statement contains an boolean expression where the result of the method *equals()* is compared with the boolean *true*, this is unnecessary because *equals()* already returns an boolean result, so the boolean expression can be simplified. Complicated boolean expressions increases complexity and affects maintainability.

Another bug verified by PMD was found in all classes, *Player*, *Team*, *Game*, and it is related to the bug pattern *Loose Coupling*. For example, in class *Team*:

```
public class Team {

   private int team;
   private ArrayList<Player> players;

}
```

As can be seen all the players on the team are stored in an *ArrayList*, however in these cases it is proposed the use of interfaces (in this case, the interface *List*) instead of implementation types like *ArrayList*. This promotes loose coupling which allows changes to be made to the structure of a class without affecting other classes that depend on it. This also affects the maintainability of a software product.

4.2.3 CheckStyle

Checkstyle[8] is an open source static code analyser that checks Java source code against rules related to coding conventions. This tool also has an architecture based on plugins or modules and is capable of checking more than 100 rules. And almost all of these rules are related to the use of Javadoc, naming conventions and programming styles.

Checkstyle can be executed as a plugin in an IDE like Vim, IntelliJ, Eclipse or NetBeans, it can be used through the command line or it can be integrated into project

[8]http://checkstyle.sourceforge.net/index.html

management tools like Ant or Maven. The analysis results can also be saved in an XML file.

Running Checkstyle

After running Checkstyle over the example, the *Design For Extension* rule turns out to be one of the most violated and was found in all classes. With this rule Checkstyle checks if public, protected, nonstatic methods of classes that can be extended are final, abstract or have an empty implementation. This style of programming avoids superclass's functionality from being affected by their subclasses. This rule relates to the reliabilty of a software product.

Another rule that was often flagged by Checkstyle was *Member Name*, as can be seen in class *Game*:

```
public class Game {

   private ArrayList<Player> team;
   private ArrayList<Player> lineup\$1;
   private ArrayList<Player> lineup\$2;
   private Date date_of_game;

}
```

In this case, variables *lineup$1*, *lineup$2* and *date_of_game* do not comply with the format $\$\hat{}[a-z][a-zA-Z0-9]*\$$ as defined in the Java language specification and the Sun coding conventions. This rule can be related to the usability of a software product.

5 Sonar: A Platform For Source Code Quality Management

Sonar [19] is an open source tool used to analyse and manage source code quality in Java projects. It evaluates code quality through seven different approaches: architecture & design, complexity, duplications, coding rules, potential bugs, unit tests and comments. Sonar groups a set of well-known code analysers such as EMMA, Cobertura, Clover, PMD, FindBugs, CheckStyle and JavaNCSS, which allows it to present features like:

- listing of all evaluated projects and its results;
- drill down to see the results at package, class and source code level;
- coding rules violation report (Sonar provides over 600 coding rules);
- classical and object-oriented design metrics measurement;
- unit tests results and code coverage;
- a time machine that shows the evolution of different quality metrics through out time;
- a dependency structure matrix (DSM) that represents dependencies between components (Maven modules, packages or files) in a compact way;
- a plugin mechanism that enables users to extend the functionalities of Sonar.

Sonar runs as a server and uses a database to persist the results from projects analysis and global configuration. It comes with an internal database (Apache Derby), however it can be configured to use other databases, such as MySQL, Oracle, PostgreSQL, or Microsoft SQL. Sonar uses Maven a software tool for building and managing Java projects. Analysis is done through a Sonar Maven plugin that executes a set of code analysers and stores the results in the database. Although Sonar uses Maven it also can analyse

non-Maven projects. This Maven plugin uses PMD and Checkstyle to find violations of coding rules like coding conventions, design problems, duplicate code and dead code. It uses FindBugs to detect potential bugs. Measurement of code coverage by unit tests is done with Cobertura and Clover. Sonar uses JavaNCSS to measure the source code metrics, LOC and CC, but it also has its own costume made code analyser named Squid that, among other things, generates the C&K and some of the R.C. Martin object-oriented design metrics and signals dependency cycles between packages.

5.1 Sonar Functionalities

Sonar is a web-based application. When accessing its web site, it is possible to see the complete list of projects that are in the data base of Sonar. Once a project is chosen, the dashboard of its page is displayed, as seen in Figure 5.1. This dashboard is the starting point to analyse the project, because it allows the user to have an idea of the overall quality.

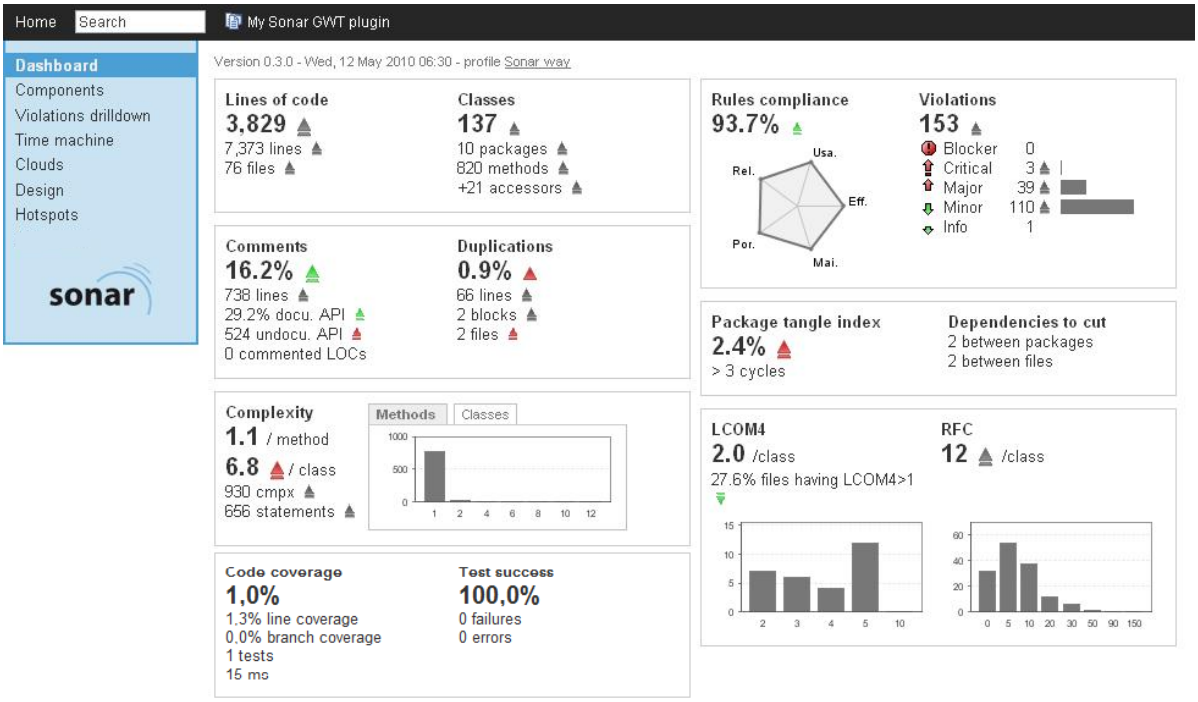


Figure 5.1: Example of a dashboard of a project in Sonar

The information displayed in the dashboard is divided by several widgets. The first three widgets display results from standard metrics like the number of LOC, packages, classes, methods, the percentage of comments in the code, the percentage of duplicated

code, and the average complexity per method and class that is complemented with a bar chart where each bar represents the number of methods/classes with a certain complexity.

Beside each metric, are arrows of various sizes and colours, these are called *tendencies*. These are calculated through the use of an algorithm that basically takes all the measures taken in the last days and determines whether the metric results have been increasing or decreasing. Gray arrows are used in quantitative metrics, while the coloured arrows are used in qualitative metrics to represent how they are affecting the quality of the project.

5.1.1 Violations Drilldown

One widget in particular displays measures related to coding rules compliance. It presents the number of coding rules violations in the project, as well as the number of level blocker, critical, major, minor and info violations. These priority levels represent all the priority levels from the three coding rules engines used by Sonar. All FindBugs rules with priority level 1 are consider blocker, level 2 are consider major and level 3 are consider info. In the case of Checkstyle, all error level rules are consider blocker, all warning rules are consider major, and all info rules are consider info. For PMD, all its 5 levels of priority are mapped to Sonar's five priority levels. Sonar also uses the ISO/IEC 9126 quality model to divide the rules 5 categories: maintainability, usability, efficiency, portability and reliability.

This widget also displays the *rules compliance index* (RCI) metric that gives the percentage of lines of code that are in compliance rules checked by Sonar, and a graph that represents the RCI for each category.

It is possible to access the *Violations Drilldown* feature through the menu to the left of the project's dashboard. This feature displays a list of all rules violations that can be sorted by priority level or category. It is also possible to list only the rules violations of a specific category or level of priority. By choosing a specific violation, this feature also displays all the classes where such violation occurs. It is also possible click on a class, to open a new page containing the source code of the class where all the violations are flagged.

5.1.2 Dependency Structure Matrix

The dashboard also contains two widgets related to the design and architecture of the project. The first widget displays the *package tangle index* which gives the level of

tangle of the packages, as well as the number of packages cycles and information in how to eliminate these cycles. The second widget displays the average value of object-oriented design metrics from the C&K metrics suite, LCOM and RFC and two bar charts where each bar represents the number classes with the specific metric result.

It is also possible to access the *dependency structure matrix* feature through the menu to the left of the project's dashboard. This feature represents all dependencies between components, Maven modules, packages, or files, depending on the navigation level. The dependency information is displayed in a matrix where each column and row header represents a different component. By clicking in a component's row, all the components that depend on it are highlighted. The component's row is also highlighted displaying all components it depends on. The cells in the matrix have numeric values that represent the number of dependencies between the component represented in the row and the component represented in the column. The cells that are coloured red represent dependencies to be eliminated in order to remove the cycles in the project. By clicking on a cell a list containing all dependencies between the two components are displayed. Ahead, it will be presented examples with this feature.

5.1.3 Coverage Clouds

The last widget in the dashboard is related with unit testing. It displays the percentage of code covered by unit tests, this metric is generated by mix the line coverage and branch coverage metrics, that are displayed in this widget. It also gives the percentage of tests success, as well as the number of failures and errors.

Once again it is possible to access the *coverage clouds* feature through the menu to the left of the project's dashboard. This feature has two tab, the *Quick Wins* tab displays all classes (represented by their name) of the project in different sizes and colours. The size represents the number of LOC in the class, the color represents the code coverage or the Rules Compliance Index, it is possible to choose either one. In the *Project Risks* tab the only difference is that the size represents a class complexity.

5.1.4 Hotspots

But Sonar also has other features in its menu, the *hotspots* feature is composed by widgets that present several top fives like most violated rules, longest unit test files, most complex files, most duplicated files, most violated files, highest average method complexity files, most undocumented file APIs.

5.1.5 Components

The feature *components* in the left menu displays the list of components that form the project or a chosen package of the project. For each component it can be seen some metrics like RCI, code coverage and also the build time.

In this feature it is also presented a tree map graph where each components is represented by a rectangle. The size and color of each rectangle represent different metrics.

5.1.6 Time Machine

The *time machine* feature is used to analyse the evolution of the metrics results obtained by a project, through the analysis made of their different builds. This feature contains a chart that displays the results of a set of metrics which were obtained from the analysis of the first down to the final build of the project.

This feature also has a table where are listed all the metrics calculated. Each column of the table represents a measure for the metric, obtained on a given build. The last column also displays a sparkline that shows the measurements evolution along the various builds. Ahead, it will be presented examples with this feature.

5.1.7 Quality Profiles

Sonar also allows to create quality profiles. In this profiles it is possible to define the name of the profile, activate/deactivate weight coding rules, define metrics thresholds (but only at Project level) and define the projects associated to the profile. With these profiles Sonar can be adapted to the requirement and quality level of each type of project.

5.2 Sonar Plugins

It is possible to add new features to Sonar. A Sonar plugin can define new extensions like new metrics to be calculated and collected and new widgets with *Ruby on Rails*[1], an open-source framework for developing database-driven web sites, to display the new metrics results in the dashboard. It is possible to create *sensors* and *decorators* to gather and process all the new defined metrics. A sensor collects and analyses data however has no access to other plugins collected data. A decorator in addition to collect and analyse data, also makes cross reference with data collected from other plugins. It is also possible

[1] http://rubyonrails.org

to add new features to Sonar with *Google Web Toolkit*[2] (GWT), a framework used to create complex browser-based applications.

Sonar plugins forge is currently hosting more than 40 plugins. There are plugins that add additional metrics to Sonar, plugins that extend Sonar to new languages (Cobol, Flex, Groovy, PHP, .NET), plugins that allow to use Sonar with integration tools like Hudson, Bamboo, AnthillPro, and Piwik, and also plugins who extend the visualization and reporting capability of Sonar. Some of these plugins are described below:

Security Rules. This plugin adds to the dashboard of projects a new widget that monitors and reports a set of coding related to security. Its shows the number of security violations, the percentage of the project in compliance the rules, and a pie chart that represents the amount of critical and major violations. The plugin is configurable, so it is possible to add new rules the existent set.

PDF Report. This plugin generates reports of projects in PDF format. These reports contain the information shown in the projec's dashboard, the coding violations by category, most violated rules, most violated files, most complex classes, most duplicated files. These reports can later be used as a part of the project's documentation.

Total Quality. This plugin also adds a new widget to the dashboard. It displays the percentage of *total quality* of the project based on its architecture (package tangle index), design (C&K metrics), code (rules compliance index) and tests (code coverage and test success).

5.3 The TreeCycle Plugin

We all have heard the phrase "A picture is worth a thousand words". This maxim can be also applied to software engineering especially in the domains of software maintenance, reverse engineering, and re-engineering where it is necessary to understand large amounts of complex data. *Software visualisation* can be seen as "the mapping from software artefacts, including programs, to graphical representations" [39]. Studies [39] show that maintenance programmers spend 50% of their time just figuring out the software to be changed. However, many researchers believe software visualisation may be one of the solutions to minimize this problem.

[2]http://code.google.com/webtoolkit

There are several projects that attempt to combine software visualisation and software metrics. For example, Holten et al. [23] propose a visualisation approach that uses treemaps to represent hierarchically organized components of a software system. Software metrics are visualised by using different computer graphics techniques like cushions, colours, textures and bump mapping.

Lanza et al. [40] propose a software visualisation technique complemented with software metrics information named polymetric views. This technique uses different layouts (trees, scatterplots, checkers and stapleds) to represent the relation between entities of a software system. The most interesting aspect of this technique is the representation of up to 5 different metrics on each node. The size of each node (width and height) represent 2 different metrics, the position of the node (axis X and Y) also represent 2 different metrics, while the color of the node represent a fifth metric.

Wettel et al. [59] propose a 3D visualisation approach which represents object-oriented software systems as cities. Classes are represented as buildings located in city districts which in turn represent packages. This approach represents metrics by the size (width and height) and color of both buildings and districts.

Sonar gathers much information in its database from various well-known code analysers. And with version 2.0 of Sonar, source code quality started to be also evaluated trough its design and architecture with the introduction of object-oriented design metrics and report of dependency cycles. Sonar plugins forge is currently hosting more than 40 plugins. However, only 4 are devoted to visualisation and reporting.

TreeCycle is a plugin for visualisation of information that concerns to the design quality of Java projects and is one of results of the work done in the elaboration of this thesis. It represents dependencies between packages in tree graphs highlighting it's dependency cycles (this is why we name it TreeCycle). Moreover, the plugin represents in a graphical way the results of the C&K metrics for the classes of each package. We think these features give an overall image of the design quality of a project and make TreeCycle a good complement to the DSM feature of Sonar.

5.3.1 How It Works

TreeCycle starts by soliciting Sonar web server the data relative to all project's packages and its dependencies. This data will be used to create a tree graph where the nodes represent packages and the edges represent the dependencies between packages.

Why a tree graph? TreeCycle displays package dependencies using graph trees because these type of graphs are useful for the display of hierarchical structures like inheritance or dependency between entities (packages, classes, methods). Nodes represent entities, while the edges between the nodes represent hierarchical relationships. The advantage of this type of graphs is that they are able to render a complex system in a very simple way. However, tree graphs of big systems tend be very large and sometimes not even fitting on one single screen.

TreeCycle uses organizational charts provided by Google Chart Tools (a.k.a. Visualisation) 1.1 library, for GWT, to generate dependency tree graphs.

Reading a tree. In Figure 5.2 it can be seen an example of a tree generated by the TreeCycle plugin. The dependencies in a tree graph are read from top to bottom, i.e., a node depends (directly and indirectly) on all the nodes standing bellow it. For instance, node 0 depends on all of the 34 nodes in the tree.

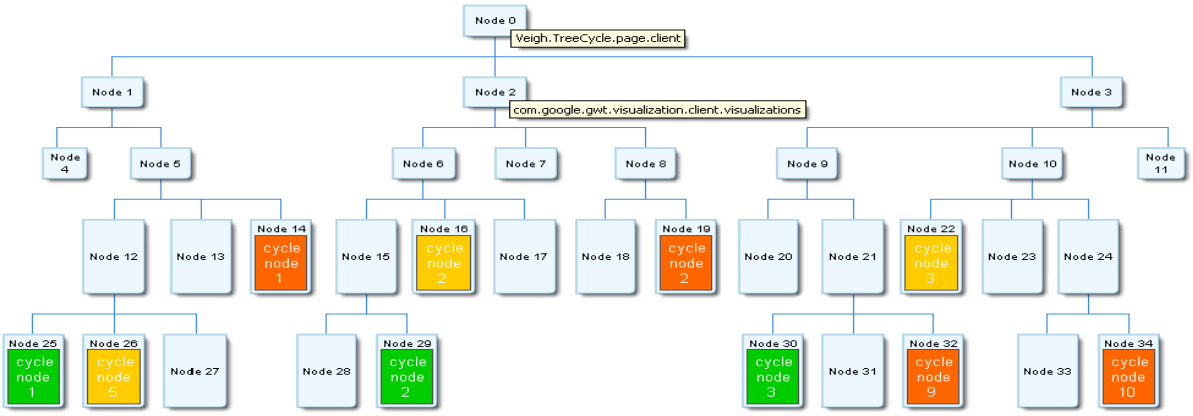


Figure 5.2: TreeCycle: package dependencies tree graph

Choosing the tree root. Initially, all packages are candidates to be the roots of the tree graphs that will be generated. What TreeCycle does to narrow this list is a test run, where for each candidate package is calculated the number of nodes that it's tree will contain. The list of packages (root candidates) is then ordered from the package with the highest count to the package with the lowest count. Next the plugin begins to build the tree graphs for the package with the highest count, till the list of root candidates is empty. Meanwhile, all packages that were used as nodes to determine the result of another package X, will be excluded from the list of root candidates, because their trees

will appear as a subtree of the dependency tree generated for X. This algorithm makes it possible to generate the minimum possible number of trees.

Identifying dependency cycles. Package structures with many cycles are in general more difficult to understand and to maintain [45], because these tend to generate spaghetti code. An important feature of the TreeCycle plugin is the highlighting of dependency cycles between packages in the tree graphs. In Figure 5.2 it can be seen some leafs highlighted with different colours. Each color identifies a different cycle, which is represented on the tree as a path that ends in the highlighted leaf and begins in the node that is identified on same leaf. Note that a dependency cycle can be reflected in a tree several times. To isolate cycles TreeCycle identifies each cycle with a different color (for instance, the tree in Figure 5.2 captures three different cycles corresponding to the three colors that appear in its nodes). Alternatively, it is possible to see a list of all dependency cycles in the design and also the information about witch packages are involved in each cycle. This can be done by selecting the *Cycle List* tab, as can be seen in Figure 5.3.

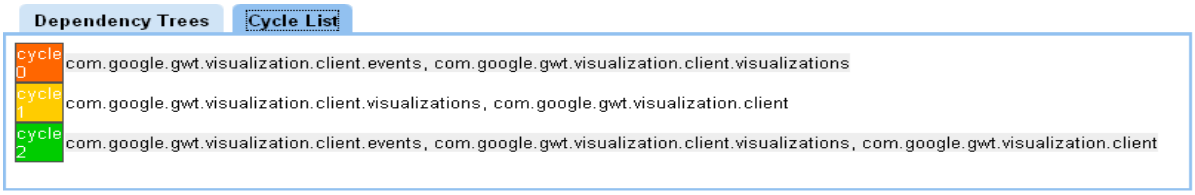


Figure 5.3: TreeCycle: list of dependency cycles

Isolating components. Besides giving a overall image of the package structure of a project and detecting dependency cycles, the TreeCycle plugin can be used to identify components that can be reused. For example, we know that in order to reuse the package represented by node 2 it is necessary to also include all the nodes standing bellow it (nodes 6, 7, 8, 15, 16, 17, 18, 19, 28 and 29).

Drilling packages. The TreeCycle plugin not only serves to generate dependency tree graphs. By clicking in a node it is possible to drill-down into the package (represented by the node) where the C&K metrics results will be displayed for all the classes that compose that package. These results are graphically represented in pie charts also provided Google Chart Tools 1.1 library. For example, in Figure 5.4 it can be seen the metrics results for all the classes from package represented by node 0.

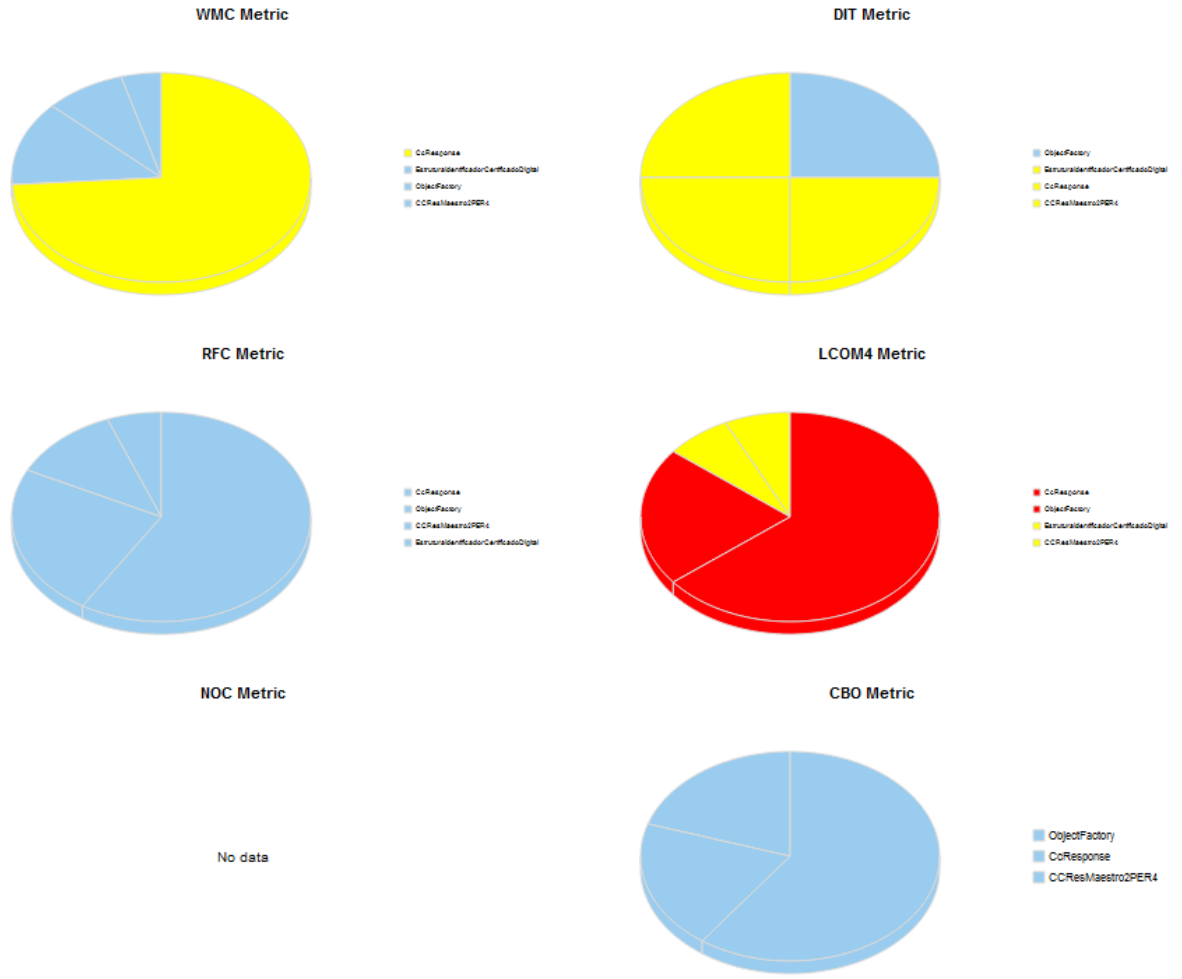


Figure 5.4: TreeCycle: C&K metrics

Using Thresholds. One of the features of TreeCycle is the option to define thresholds for each C&K metric. Just go to Sonar configurations, then settings, and then select the TreeCycle tab and enter the thresholds for each metric. Whenever a metric result exceeds a threshold an alert is issued in the pie chart for the corresponding metric and in the slice representing the faulty class. These alerts are represented by different colours (for example blue, yellow or red) depending on the degree of risk.

5.3.2 Assessing TreeCycle with Sonar

After selecting the TreeCycle project and open the dashboard of the project page, it can be seen some general information. The widgets in Figure 5.5 show that the project contains 1231 lines (3.8 % being comments), 844 lines of code (blank lines, comments,

source code), 10 classes, 3 packages and 74 methods and that almost all these results have increased from one build to a more recent one. There is no duplicate code. Despite the architectural and design quality being high (100% and 86,7%, respectively), the overall quality of this project is 70,8%, because of code quality (influenced by the high number of coding rules violations) but mainly the low percentage of unit tests.

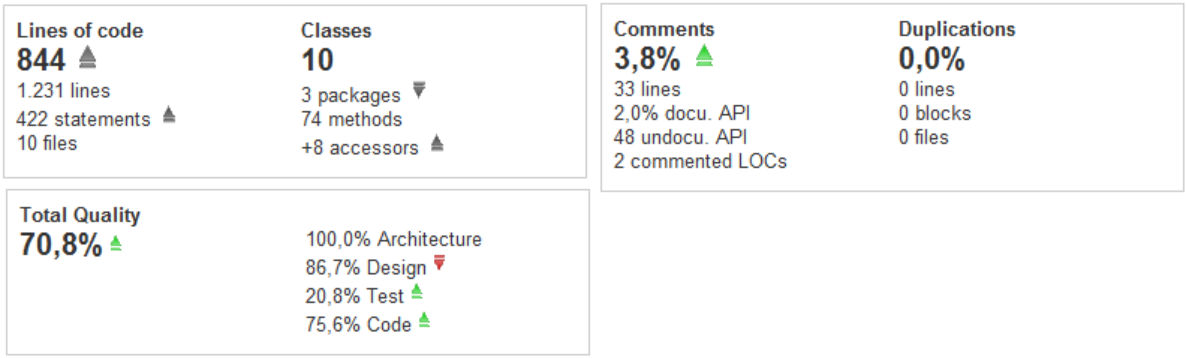


Figure 5.5: TreeCycle: general information

By choosing the *Time Machine* in the left menu of Sonar, it is possible to analyse the evolution of the results obtained by a project, through the analysis made of their different builds. In Figure 5.6 the chart shows that the complexity, of the project has grown from the first build to the second, but it dropped with the third build. Code coverage has been increasing, though very slowly, in every new build. The chart also shows that the code quality and design quality of the project dropped from the second build to the third build. On the other hand, architecture has improved.

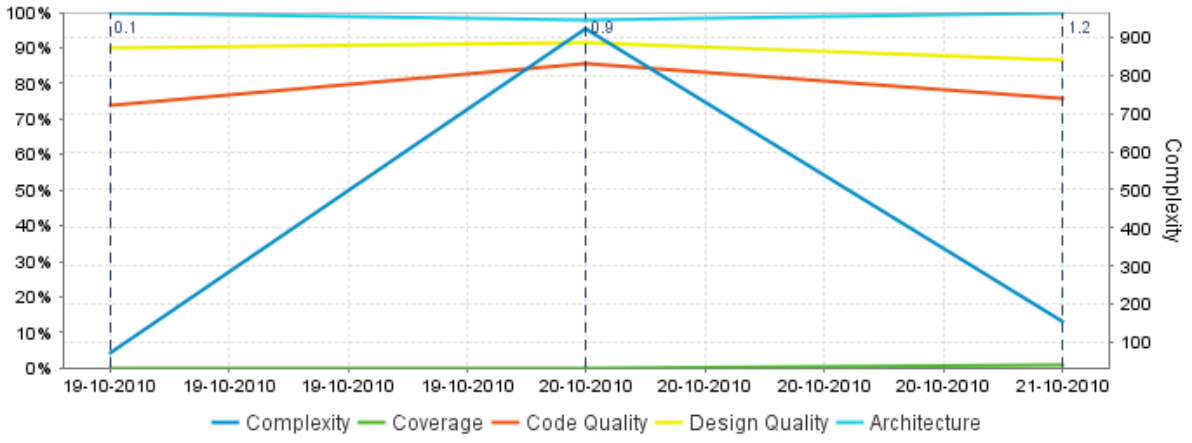


Figure 5.6: TreeCycle: quality evolution

Statical analysis (Rules Compliance)

In Figure 5.7, it can be seen the values associated with the coding rules, 78.4% of the lines of code are in compliance with the 480 rules checked by Sonar. There are 87 rules violations (which is not high), 48 are considered major 38 are considered minor.

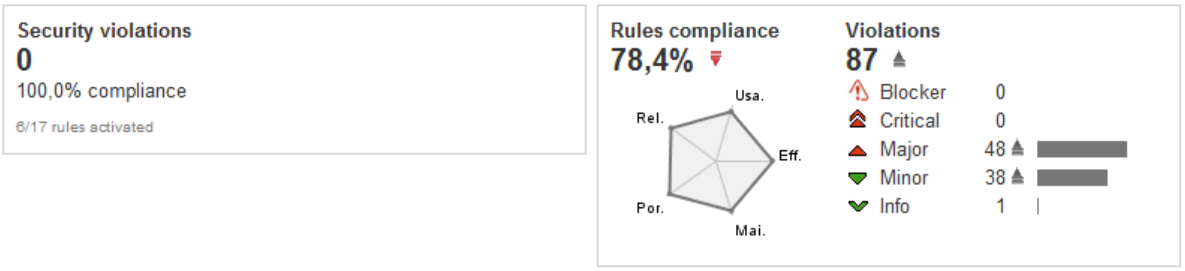


Figure 5.7: TreeCycle: rules compliance info

By clicking in the number of violations, the *Violations Drilldown* feature is opened. As can be seen in Figure 5.8, most of the violations affect the quality factors usability (33), reliability (31) and maintainability (22). The package with must violations is *client* (74) and the most violated class is *orgChart* (31).

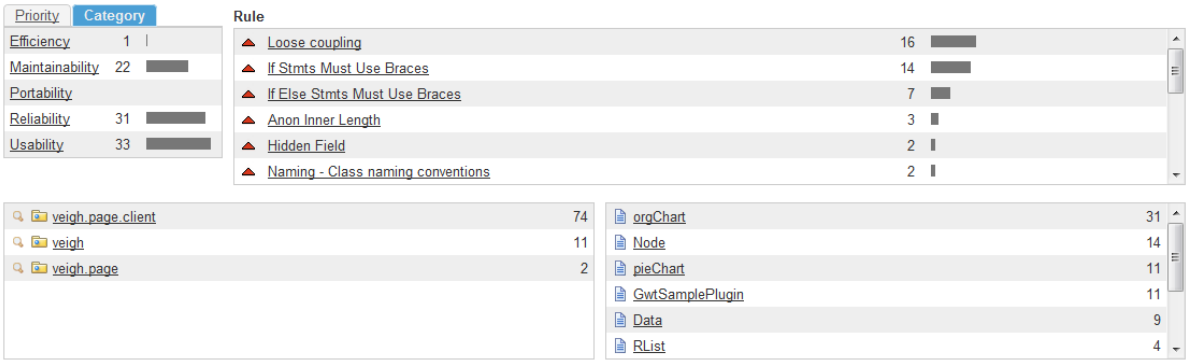


Figure 5.8: TreeCycle: rules compliance drill-down

Most of the major rules violations are related to coding conventions like absence of braces on *if* and *for* statements, or the use of interfaces instead of implementation types (e.g. the use of List instead of ArrayList), and naming conventions of classes. Most of the minor rules violations are also related to coding conventions like design for extension, constant naming conventions, and the use of numeric literals that are not defined as constants.

OOD Metrics (Design & Architecture)

In the dashboard of the project page, it can also be seen some results related to complexity and object-oriented metrics. The widgets in Figure 5.10 show that the average cyclomatic complexity per method (2.1), class (15.7) which are low, considering that the threshold normally considered for CC for methods is 10. It can be seen that the average LCOM value per class is 1.7, which means that there are classes with more than 1 set of related methods and fields, in fact the graph show that there are two classes with LCOM value of two, therefore these classes must be reviewed. It can also be seen that the average RFC value per class is 26. In relation to package dependencies, the package tangle index is 0% which means that the project's design is good and does not have dependency cycles.

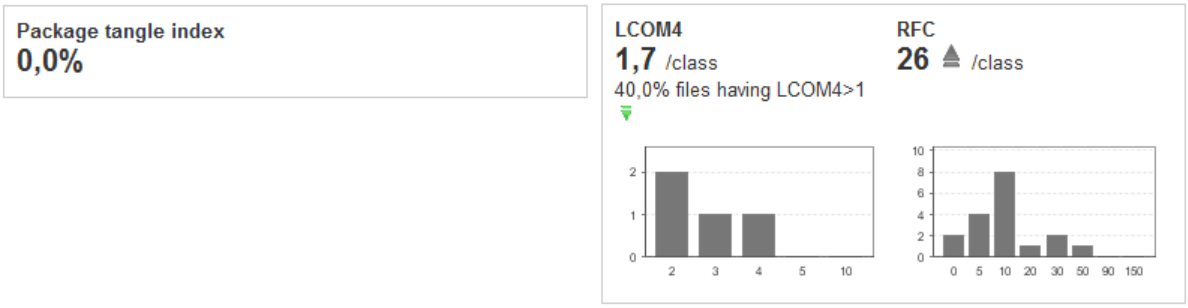


Figure 5.9: TreeCycle: design & architecture

The TreeCycle plugin gives an overview of the architecture and design of our project, so it also allows to identify the dependency cycles that the project contains. In Figure 6.5,it can be seen that the dependency tree in fact does not have dependency cycles.

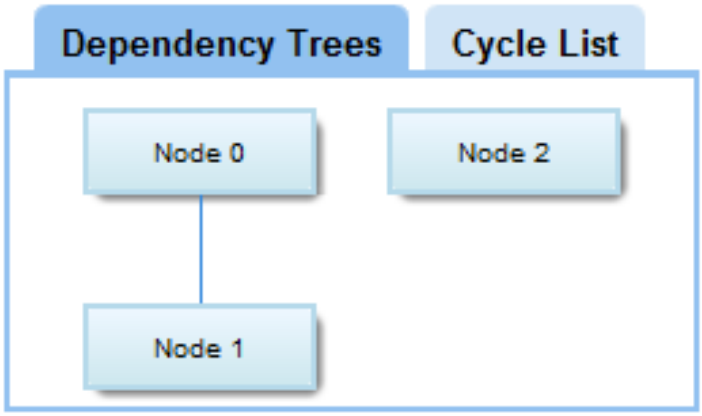


Figure 5.10: TreeCycle: dependencies tree

By continuing to use TreeCycle, it is possible to drill-down to the package level and visualize the C&k metrics results for each class of a package.The class that stands out in almost all pie charts is the *orgChart* class. This is the class responsible for arranging all data in a table that will be used for generating the dependency tree graphs. In Figure 5.11, it can be seen that *orgChart* class is the one with the higher values for WMC (54), LCOM (3), RFC (83) and CBO (3) metrics. This values tells us that this class is potentially hard to analyse, change and test. This class is the main responsible for the maintainability, re-usability and portability rates of this project, affecting its overall quality.

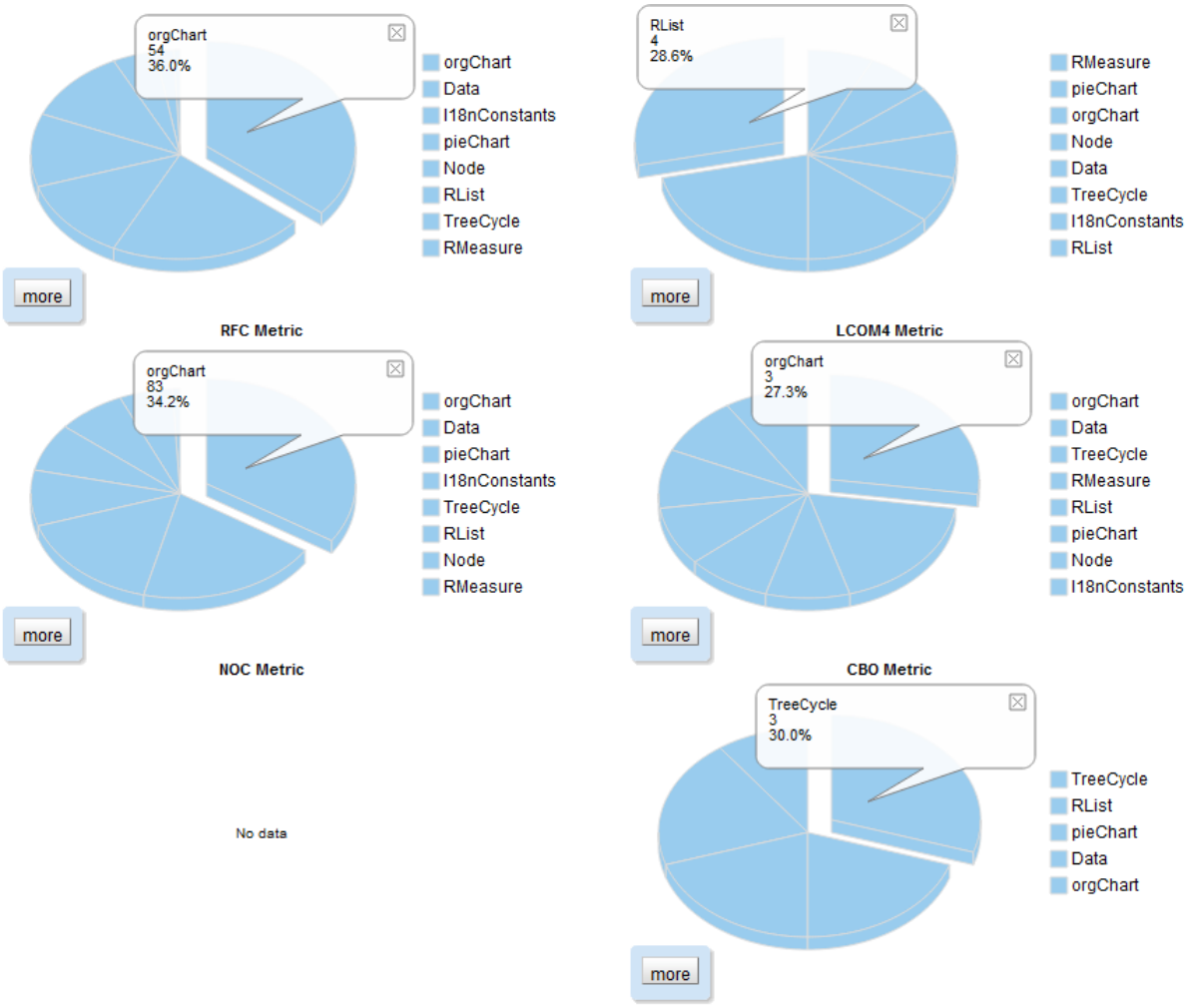


Figure 5.11: TreeCycle: C&K metrics results

Unit Testing

Finally the last widget in the dashboard displays the information related to unit tests. In Figure 5.12, it can be seen that unit tests cover 1.0% of the overall source code, which is low and has to be improved, this represents 1.3% of line coverage, and 0.0% of branch coverage. This widget also displays the success rate of the tests which is 100%.

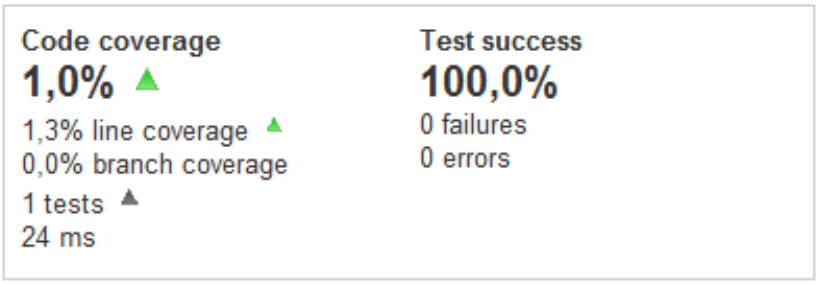


Figure 5.12: TreeCycle: unit tests

In fact, by choosing the *Coverage Clouds* feature in the left menu, it can be seen that class *orgChart* is the biiggest class in the project (in relation to LOCs) and the only class 100% code coverage is class *RMeasure*, which is also one of the smallest classes in the project with and 17 LOCs, as can be seen in Figure 5.13.

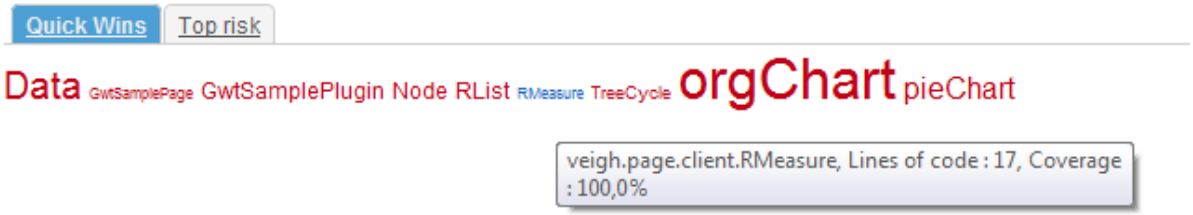


Figure 5.13: TreeCycle: coverage cloud

5.4 Sonar in the Evaluation Process

As said before, software quality assessment is on the agenda due to several factors, and has already been seen that Sonar is a tool that can be used to make this assessment. However it is important to understand how Sonar fits in the various stages of an evaluation process.

Sonar can be used in independent and external audits to software products of a company. The ISO/IEC 14598 divides the evaluation process for evaluators in four stages: evaluation requirements, evaluation specification, evaluation plan and evaluation. Sonar can be used in the *evaluation plan* stage to create a Sonar quality profile for the project and define the coding rules to be checked by taking into account quality factors that were chosen in the *evaluation requirements* stage. It is also possible to enhance the quality profile with the thresholds of the metrics chosen for each component of the project, in the *evaluation specification* stage. In the *evaluation results* stage, it is obvious that Sonar can be used in the activities defined in the *evaluation plan* to evaluate the project, but it can also be used to elaborate reports and document the results.

However Sonar reaches its full potential when used as a shared central repository for quality management enabling to improve the quality of a software in a continuous and supported manner. With Sonar, stakeholders have facilitated access to information that enables them to manage risks, reduce maintenance costs and improve agility, during a project's development life cycle. The ISO/IEC 14598 divides the evaluation process for developers in four stages: organization, project planning and quality requirements, specifications, design and planning, and build and test. Sonar can be used in this process in the same way described for the evaluators, because both processes are very similar. In the *design and planning* stage, Sonar can be used to create a Sonar quality profile for the project and define the coding rules to be checked by taking into account quality factors that were chosen in the *project planning* stage. It is also possible to enhance the quality profile with the metrics thresholds defined in the *specification* stage. In the *build and tests* stage, it is also obvious that Sonar can be used to collect and analyse the metrics results in order to evaluate the project's quality, but it can also be used to elaborate reports and document the results.

6 Case Studies

So far, it was seen how Sonar worked, along with TreeCycle, and how it can be used in the process of developing a software product. Now, it will be shown the application of this tool in two industrial Java projects of medium size. The source code of these projects were provided by Multicert[1], a company that develops complete security solutions focused on digital certification for all types of electronic transactions that require security. For each project, two different builds were made available to compare and evaluate the quality the projects in different stages.

After determining the projects results with Sonar, the next step was to estimate the thresholds for all the software metrics from the C&K metrics suite. Since Sonar does not determine thresholds for packages and classes (only project measures), the C&K metrics thresholds were calculated through the use of a simple methodology proposed by Erni et al [15] that is based on the use of statistical methods.

Basically, the results of the C&K metrics of each project were grouped together, and for each metric was calculated its average value and standard deviation. Because the amount of results was huge, we decided to implement a simple plugin that collects the results of the Sonar database and uses these to calculate the mean and standard deviation of each metric. The calculated Thresholds as well as the mean and standard deviation obtained for each metric can be seen in Table 6.1.

Next, we will start by analysing the general results displayed on each project dashboard like the number of LOC, packages, methods and classes, the percentage of duplicated code, and total quality percentage of the project evaluated through 4 categories (architecture, design, code and test). Then we take a look at the results related with rules compliance and analyse rules violations by taking into account its importance (blocker, critical, major, minor and info) and the quality factor that affected. Finally we end with the analysis of the architecture quality by identifying the dependency cycles, and the analysis of the design quality by viewing and interpreting the C&K metrics results for one of the most complex packages in each project.

[1] https://www.multicert.com/home

Table 6.1: C&K metrics thresholds

Design Metric	Mean (average)	Standard Deviation	Threshold
WMC	14	23	37
DIT	1	1	2
CBO	2	3	5
RFC	15	22	37
NOC	0	0	0
LCOM	1	2	3

6.1 Maestro Web Service Test Project

The first project, named Maestro Web Service Test, is a test kit for another Multicert project, the Portuguese citizen card[2]. This card, in addition to physically identify a person, can also be used as a means of identification and computerized services for authentication of electronic documents.

After selecting the Maestro project and open the dashboard of the project page, we can see some general information. In Figure 6.1, it can be seen that the project contains 22239 lines (34 % being comments), 10158 lines of code (blank lines, comments, source code), 189 classes, 33 packages and 924 methods and that all these results have increased from one build to a more recent one. The percentage of duplications is low (6.1%) wich is good, because "copy-and-pasting" code can lead to spread of bugs and therefore increasing the maintenance effort, but we can see by the red arrow that code duplication has increased and is affecting the quality of the project. Despite the architectural and design quality being high (96,7% and 93,6%, respectively), the overall quality of this project is 66,4% (and decreased from one build to another), because of code quality (influenced by the high numbre of coding rules violations) and the non-existence of unit tests.

6.1.1 Statical Analysis (Rules Compliance)

If we look at Figure 6.2, we can see the values associated with the coding rules, 62.7% of the lines of code are in compliance with all the rules checked by Sonar. There are 2148 (151 of them relevant to security) rules violations, which is high, but the good news is that only 35 are considered critical and there are no violations considered blocker.

[2]http://www.cartaodecidadao.pt/index.php?lang=en

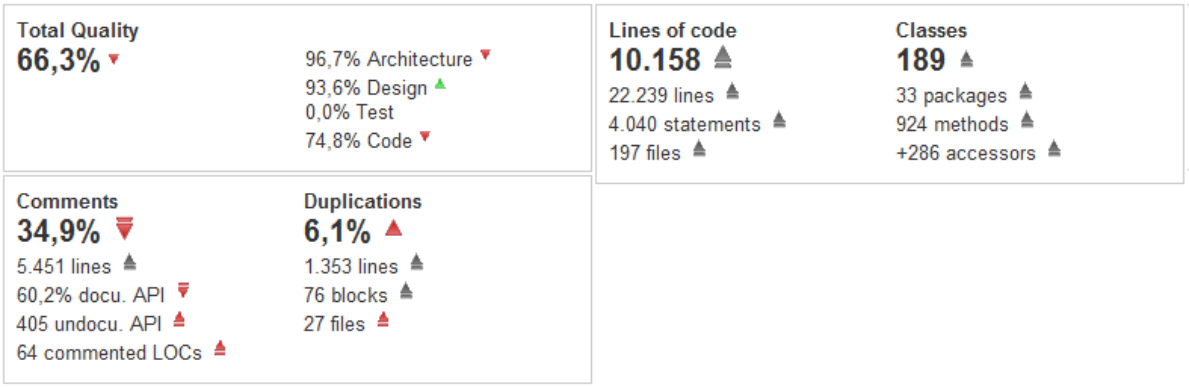


Figure 6.1: Maestro: general information

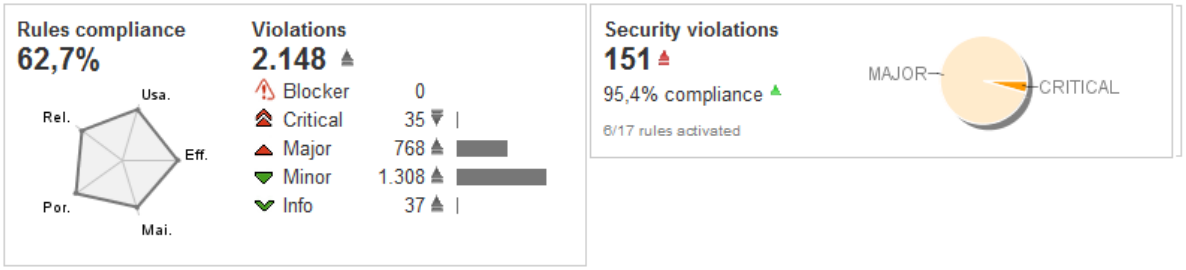


Figure 6.2: Maestro: rules compliance info

By clicking in the number of violations, the *Violations Drilldown* feature is opened. As can be seen in Figure 6.3, most of the violations affect the quality factors reliability (1276) and maintainability (503). The package with most violations is *Util* (354) and the most violated class is *MRZGenerator* (107).

Critical rules violations

Of the 35 critical violations, 28 are related to the *Security - Array is stored directly* rule. Examples of violations of this rule can be found in class *Certificates* from package *result*:

```
public void setCarootx509(byte[] carootx509) {
    this.carootx509 = carootx509;
}

public void setPkcaroot(byte[] pkcaroot) {
    this.pkcaroot = pkcaroot;
}
```

As can be seen in this examples, the arrays *carootx509* and *pkcaroot*are saved directly. This is a problem because it allows the direct manipulation of the private data structure

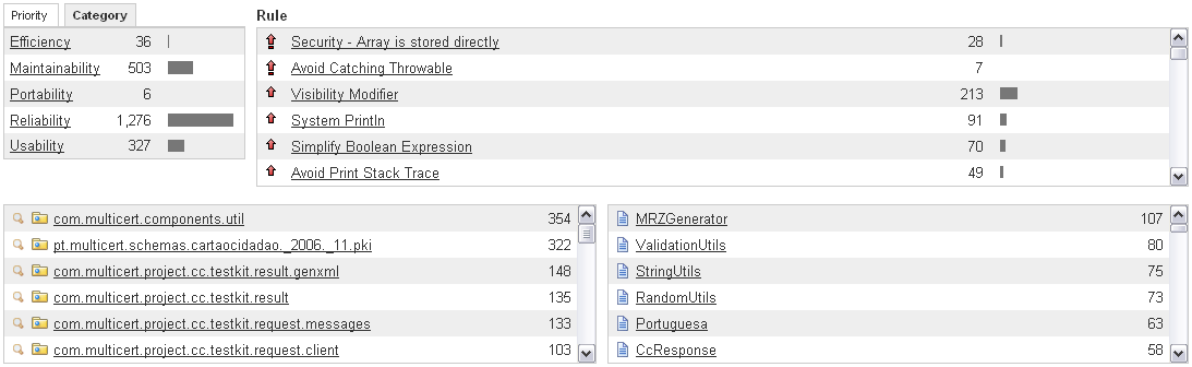


Figure 6.3: Maestro rules compliance drill-down

of objects from class *Certificates*. It is recommended instead that copy of the array is stored through the use of the *clone()* method.

The other 7 critical violations are related to the *Avoid Catching Throwable* rule. Examples of violations of this rule can be found in class *RequestPublicKey* from package *run*:

```
public void run() {
    try {
        CCGerarChavePublicaCartao c = new CCGerarChavePublicaCartao();
        c.setNumeroProcesso( Long.toString(this.procNumber) );
        c.setPrioridade(TipoPrioridade.NORMAL);

        KeyRefServiceIntf client = (new KeyRefServiceClient())
        .getKeyRefServiceHttpPort(...);

        client.keyRefSupplier( c );
    }
    catch ( Throwable t ) {
        t.printStackTrace();
    }
}
```

This is problematic because *Throwable* is to generic, it can catch, not only exceptions, but also errors like *OutOfMemoryError* and in these cases it is better to stop the application. Both the *Security - Array is stored directly* and the *Avoid Catching Throwable* rules are related to security and its violations affects reliability.

Major rules violations

Of the 768 major violations, 203 are related to the *Visibility Modifier* rule. Examples of violations of this rule can be found in class *Portuguesa* from package *messages*:

```
...
```

```
protected String codigoPais;
protected String codigoDistrito;
protected String distrito;
protected String codigoConcelho;
protected String concelho;
protected String codigoFreguesia;
protected String freguesia;
...
```

Encapsulation is one of the six fundamental concepts in OOP. It prevents object's components from being randomly accessed by code defined outside the object's class. Encapsulation ensures very important properties such as ease of reuse, error detection and modularity [2]. In the previous example, we can see that the instance variable that defines the data structure class *Portuguesa* are defined as *protected*, i.e., its structure data can be accessed directly by all other classes and subclasses belonging to the same package. This creates a fault in the encapsulation, making it more difficult to correct mistakes and re-use this class. It is recommended to change the status of the instance variables to *private* and that the instance methods are used to access the variables. The violation of *Visibility Modifier* rule affects maintainability.

There are 140 rules violations related with the *System Println* and *Avoid Print Stack Trace*. Instead of using *printStackTrace* and *println*, it is recommend the use of logging tools like Log4j to facilitate debugging. The violation of these rules affects usability.

The *Simplify Boolean Expression* and *Simplify Boolean Return* rules have 75 violations (70 and 5, respectively). Examples of violations of this rule can be found in class *Portuguesa* from package *messages*:

```
if( name.equals( "RSA" ) == true ) {
   return ( "2.5.8.1.1" );
}
else if( name.equals( "DSA" ) == true ) {
   return ( "1.3.14.3.2.12" );
}...
```

Despite the simplification of the boolean expressions (decisions) above not being too significant, simplification of boolean expressions is important because they are easier to understand and analyse. In the maintenance stage, when deriving test cases and using test coverage methodologies based on control flow analysis (condition/decision coverage, multiple condition coverage, path coverage,etc), the number of tests needed to achieve a high code coverage are determined by the number of combinations among the conditions that form the expression, so it also decreases the testing effort. The violation of the rule *Simplify Boolean Expression* increases the testing effort and consequently affects maintainability.

The *Unused local variable* rule has 43 major violations. Examples of violations of this rule can be found in class *StringUtils* from package *util*:

```
public static int getMaximumLength( String[] array ) {
   int res = 0;
   String first = array[ 0 ];
   for( int i = 1; i < array.length; i++ ) {
      if( first.length() < array[ i ].length() ) {
         first = array[ i ];
      }
   }
   return first.length();
}
```

As can be seen in this example, variable *res* is defined but not used. The first problem that this creates is the unnecessary allocation of resources, but in more complicated methods it also makes it harder to understand and change them, making it more difficult to maintain a class. The violation of *Unused local variable* rules affects maintainability.

As stated before, CC is a well-known metric that measures the level of complexity of methods, the higher the CC value, the harder it is to understand the source code and test a method, specially when trying to achieve a high code coverage by using test coverage methodologies based on control flow analysis (like path coverage). Sonar uses a default maximum threshold of 10 for this metric, but this can be altered. *Cyclomatic Complexity* rule has 16 major violations. Examples of violations of this rule can be found in class *MRZGenerator* from package *genxml* that contains a method (*truncate*) with a CC value of 31 and in class *ValidatonUtils* from packageutil that has one method (*getDate*) with a CC value of 27. The violation of *Cyclomatic Complexity* rule affects maintainability.

The *Unused private method* and *Simplify Boolean Return* rules have 2 violations. Method *getNextProcId* is an example of a violation of this rule and can be found in class *RequestPublicKey* from package *run*:

```
public class RequestPublicKey implements Runnable {

   private static long count = 1;
   private long procNumber;

   public RequestPublicKey(long procNUmber) {
      this.procNumber = procNUmber;
   }

   public void run() {

      try {
      CCGerarChavePublicaCartao c = new CCGerarChavePublicaCartao();
```

```
//          String proc_id = getNextProcId();
            c.setNumeroProcesso( Long.toString(this.procNumber) );
            c.setPrioridade( TipoPrioridade.NORMAL );

            KeyRefServiceIntf client = (new KeyRefServiceClient())
               .getKeyRefServiceHttpPort("...");

            client.keyRefSupplier( c );
            }
            catch ( Throwable t ) {
               t.printStackTrace();
            }

         }

         private synchronized String getNextProcId(){
            DecimalFormat df = new DecimalFormat("...");
            String id = df.format(count);

            count ++;
            return "99" + id;
         }
      }
```

As can be seen in the example, the private method *getNextProcId* is never executed and does not have any purpose. Dead code only is contributing to the increasing complexity of a class and the decline in its quality. Eliminating dead code will benefit maintainability, because it decreases code size, testing effort and makes a class easier to understand.

The rest of the major rules violations are related to naming conventions (that can be altered) of variables, methods and classes, the absence of braces in *if* statements, the use of duplicated string literals instead of declaring the string as a constant variable, and unused instance variables.

Minor and info rules violations

The other rules violations are considered of minor level (1308). They consist of violations of such as *Modifier Order* that imposes the order of modifiers (public, protected, private, abstract, static, final, transient, volatile, synchronized, native, strictfp) suggested in the Java Language specification, *Collapsible If Statements* that checks for nested *if then else* statements that can be combined, and *Magic Number* that checks the existence of numeric literals that are not defined as constants. There are also 37 violations that inform the existence of unused imports and unused modifiers.

6.1.2 OOD Metrics (Design & Architecture)

In the dashboard of the project page, we can also see some results related to complexity and object-oriented metrics. We can see in Figure 6.4 the average cyclomatic complexity per method (2.1), class (10.2) and file (9.8) which are low, considering that the threshold normally considered for CC is 10, at method level. It can be seen that the average LCOM value per class is 1.4, which means that there are classes with more than 1 set of related methods and fields, therefore these classes must be reviewed. It can also be seen that the average RFC value per class is 12. With respect to package dependencies, the package tangle index is low (5.3%) which means that the project's design is good but can also be improved by cleaning the 2 existing dependency cycles. These can be resolved by cutting 1 dependency between packages, which is formed by 3 dependencies between files.

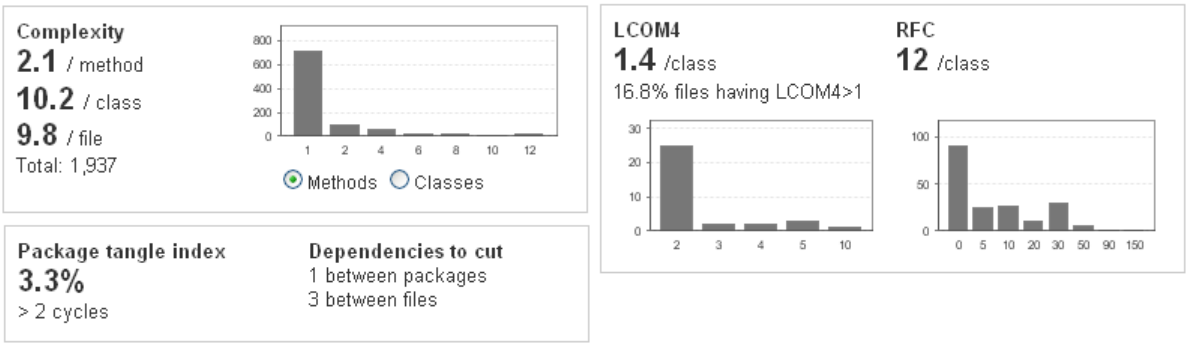


Figure 6.4: Maestro: design & architecture

TreeCycle gives an overview of the architecture and design of the project, so it also allows to identify the dependency cycles that the project contains. In Figure 6.5, we can see an extract from the dependency tree that contains two cycles and the first thing we notice is that packages *genxml* and *workfiles* are in both cycles. By choosing the *Cycle List* option in the TreeCycle plugin, we can immediately verify the sets of packages involved in each cycle.

Next, we choose the Design feature from Sonar to get a more detailed description of the dependency between the 2 packages, that needs to be cut in order to eliminate the two cycles. As can be seen in Figure 6.6, there exists one dependency from *workfiles* to *genxml* and in order to eliminate this package dependency we have to cut the 3 files dependencies between *WorkFilesProcessor* and *MRZGenerator*, *WorkFilesProcessor* and *MRZUtilConverter* and *WorkFilesProcessor* and *ResultCardBuilder*.

By continuing to use TreeCycle, we can drill down to the package level and visualize the C&K metrics results for each class of a package. In*genxml* exist some classes which

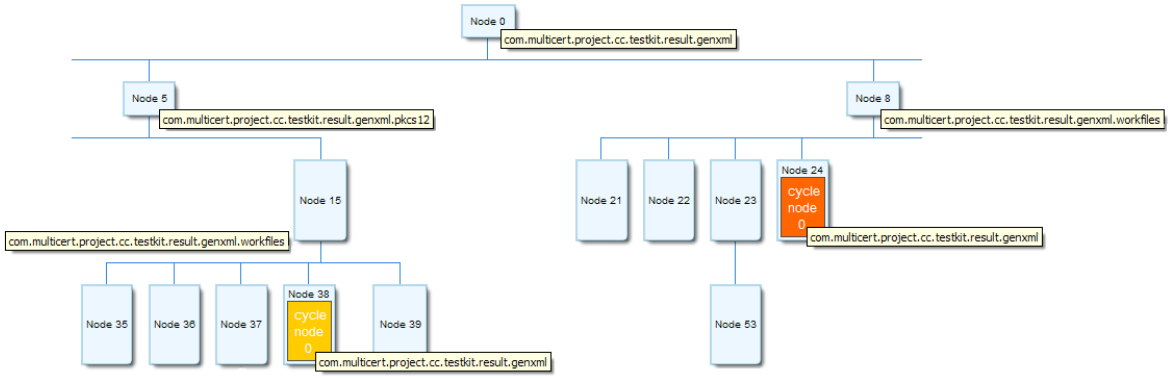


Figure 6.5: Maestro: dependencies tree

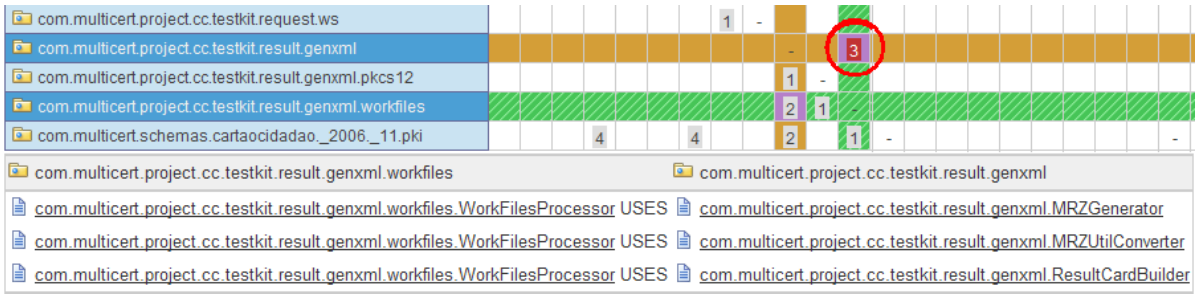


Figure 6.6: Maestro: dependency structure matrix

the results exceed defined thresholds. *ResultCardReader* is a class with only 6 methods and obtained CBO (14) and RFC (39) results that exceed the thresholds, these results show that this class is one of the most complex and fault-prone from this package, making it more difficult to debug, reuse and less adaptable to changes. Let us take the example of *addEcd2Data* method:

```
public void addEcd2Data(CCResECD2PER25 data){

        PublicKeys publicKeys = new PublicKeys();

        publicKeys.setPkAuthentication(data.getCCResponse()
            .getPKASK01AUTHENTICATION());

        publicKeys.setPkIccaut(data.getCCResponse()
            .getPKICCAUT());

        publicKeys.setPkSignature(data.getCCResponse()
            .getPKASK02SIGNATURE());

        PrivateKeys privateKeys = new PrivateKeys();
```

```
        privateKeys.setSkAuthentication(data.getCCResponse()
            .getSKASK01AUTHENTICATION());

        privateKeys.setSkIccaut(data.getCCResponse()
            .getSKICCAUT());

        privateKeys.setSkSignature(data.getCCResponse()
            .getSKASK02SIGNATURE());

        HashKeys hashKeys = new HashKeys();

        hashKeys.setHashAuthentication((ArrayUtils
            .convertToHexString(data.getCCResponse()
                .getHASHSKASK01AUTHENTICATION(), false, false))
                    .toUpperCase());

        hashKeys.setHashIccaut((ArrayUtils
            .convertToHexString(data.getCCResponse()
                .getHASHSKICCAUT(), false, false))
                    .toUpperCase());

        hashKeys.setHashSignature((ArrayUtils
            .convertToHexString(data.getCCResponse()
                .getHASHSKASK02SIGNATURE(), false, false))
                    .toUpperCase());

        resultCard.setPublicKeys(publicKeys);
        resultCard.setPrivateKeys(privateKeys );
        resultCard.setHashKeys(hashKeys);
}
```

We can see that this example is too complex. Through the use of parametrized constructors to generate the *PublicKeys*, *PrivateKeys* and *HashKeys* objects, the method emphaddEcd2Data could easily become less complex, more readable and reduce the number of methods invoked.

However, the class with the worst results is *MRZGenerator*. It obtained the higgest WMC (58), RFC (42) and LCOM (2) results, surpassing even the established thresholds, as can be seen in Figure 6.7.

This high values show us that most of the maintenance effort of the *genxml* package will be spent in this class, because complex classes are harder to analyse, test, replace or modify, therefore this class should be redesigned in order to reduce its complexity and improve the overall quality. In the case of the WMC result, this shows us that the class *MRZGenerator* is formed by a huge number of methods, by small number of highly complex methods or both. Indeed this class has 10 methods with a average of 5.8 CC and its biggest method (*truncate*) has a CC value of 31 and 123 LOCs, almost 1/3 of the class's 370 LOCs. There are two ways of reducing the WMC result and consequently

Figure 6.7: Maestro: C&K metrics results

the complexity of this class. The first approach consists in splitting the more complex methods into simpler ones. The second approach is to divide the the class in two smaller classes, and in fact the LCOM result shows us that this class has 2 sets of highly cohesive methods (instead of the ideal 1), as can be seen in Figure 6.8. This second approach also helps us to reduce the LCOM result and consequently the complexity of the class *MRZGenerator*. However it will be also necessary to apply the first approach, since one of the possible 2 classes would still contain the major method of *MRZGenerator*

Overall, this project has 33 packages all of them with similar results to the package *genxml* and all of them can be improved by using the suggestions made earlier for classes *ResultCardReader* and *MRZGenerator*. With this changes, the project's total quality will greatly increase.

Figure 6.8: Maestro: lack of cohesion methods

6.2 SMail J2EE Project

The second project, named SMail, is a recent J2EE project developed by Multicert. This project is divided in 5 modules *util*, *data system*, *mensagens*, *schedulers*, and *ejb*.

Once the SMail project is selected, we can see some initial information project's dashboard. In Figure 6.9, it can be seen that the project contains 19259 lines (21.4 % being comments), 11329 lines of code, 133 classes, 11 packages and 1.049 methods, all these results have diminished from one build to a more recent one. The percentage of duplications is low 14.0% and we can see by the green arrow that code duplication has decreased which is a good sign, because "copy-and-pasting" code can lead to spread of bugs and therefore increasing the maintenance effort and affecting the quality of the project. Despite the architectural and code quality being high (96.7% and 82.6%, respectively), the overall quality of this project is 65% (and decreased from one build to another), because of design quality (influenced by the C&K metrics results for the projects classes) and the non-existence of unit tests.

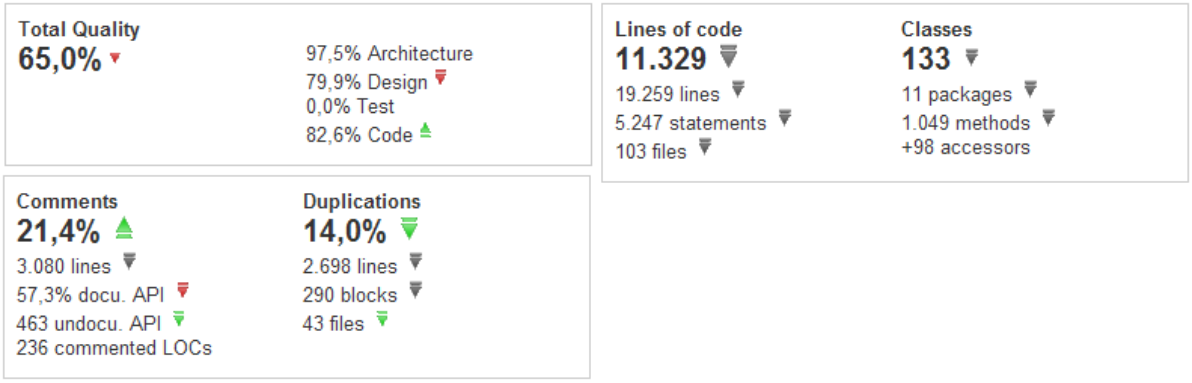


Figure 6.9: SMail: general information

6.2.1 Statical Analysis (Rules Compliance)

In Figure 6.10, we can see some of the values associated with the coding rules, 88.1% of the lines of code are in compliance with the 480 rules checked by Sonar. There are 1184 rules violations (but only 6 of them are relevant to security), which is a high number, but 908 of the violations are only considered minor, 148 are considered major and there are no blocker and critical violations.

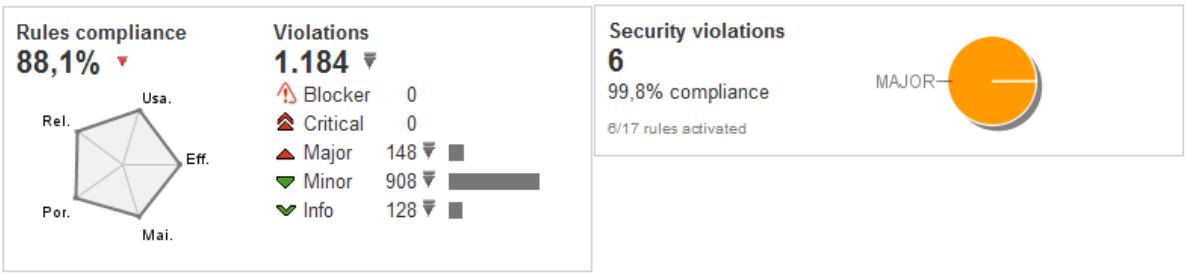


Figure 6.10: SMail: rules compliance info

By clicking in the number of violations, the *Violations Drilldown* feature is opened. As can be seen in Figure 6.11, most of the violations affect the quality factors reliability (561) and maintainability (448). The package with must violations is *ejb* (680) and the most violated class is *EMailDocumentImpl* (49), from package *impl*.

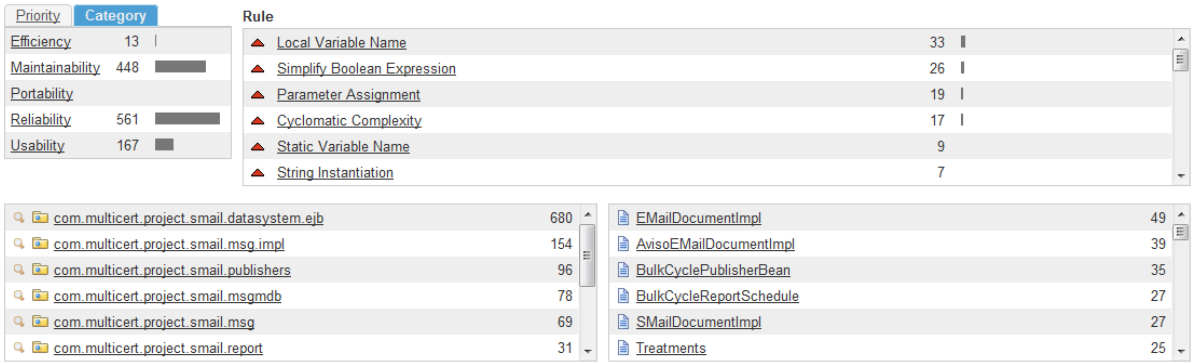


Figure 6.11: SMail: rules compliance drill-down

Critical rules violations

Of the 148 major violations, 33 are related to the *Local Variable Name* rule. Examples of violations of this rule can be found in class *MsgSenderMDB* from package *msgmdb*:

```
...
```

```
String filename_msg = error_directory + msgID + "_error";

String filename_report = report_directory + msgID + "_report";

String filename_msg = error_directory + msgID + "_error";
...
```

In this example, local variables *filename_msg*, *filename_report* and *filename_msg* do not comply with $[a-z][a-zA-Z0-9]*\$$, the naming convention defined in Sonar. Although these 3 local variables names are not extreme example of the *Local Variable Name* rule violations, the use of a naming convention can be useful to better understand the source code and to enhance its appearance. And besides, the naming format can be changed to better adapt to the specificities of the project. The *Local Variable Name* rule violations affects usability.

The *Simplify Boolean Expression* and *Simplify Boolean Return* rules have 29 violations (26 and 3, respectively). Examples of violations of this rule can be found in class *BulkCyclePublisherBean* from package *publishers*:

```
if (this.dirs.containsID(INPUT_DIR_ID) == false) {
   logger.error("...");
   throw new CreateException("...");
}

if (this.dirs.containsID(TMP_DIR_ID) == false) {
   logger.error("...");
   throw new CreateException("...");
}

if (this.treatments.containsID(OK_TREATMENT) == false) {
   logger.error("...");
   throw new CreateException("...");
}
```

Despite the simplification of the boolean expressions (decisions) above not being too significant, simplification of boolean expressions is important because the expressions are easier to understand and analyse. In the maintenance stage, when deriving test cases and using test coverage methodologies based on control flow analysis (condition/decision coverage, multiple condition coverage, path coverage,etc), the number of tests needed to achieve a high code coverage are determined by the number of combinations among the conditions that form the expression, so it also decreases the testing effort. The violation of the rule *Simplify Boolean Expression* increases the testing effort and consequently affects maintainability.

As stated before, CC is a well-known metric that measures the level of complexity of methods, the higher the CC value, the harder it is to understand the source code

and test a method, specially when trying to achieve a high code coverage by using test coverage methodologies based on control flow analysis (like path coverage). Sonar uses a default maximum threshold of 10 for this metric, but this can be changed. *Cyclomatic Complexity* rule has 17 major violations. Examples of violations of this rule can be found in class *BulkCycleReportSchedule* from package *report* that contains a method (*perform*) with a CC value of 25 and in class *MultiImpl* from package *msgmdb* that has one method (*setConfiguration*) with a CC value of 26. The violation of *Cyclomatic Complexity* rules affects maintainability.

As in project *Maestro web service test*, there are also major rules violations related to naming conventions (that can be adapted) of variables, methods and classes, to the absence of braces in *if then else* statements, the use of duplicated string literals instead of declaring the string as a constant variable, and to unused instance variables.

Minor rules violations

From the 913 minor violations, 462 are related to the *Design For Extension*. As said before, this rule forces all public, protected, nonstatic methods of classes that can be extended to be final, abstract, or have an empty implementation. This style of programming avoids superclass's functionality from being affected by their subclasses. Violations of this rule affect reliability.

The *Magic Number* rule has 99 violations. Examples of violations of this rule can be found in class *SmailReportId* from package *ejb*:

```
public int hashCode() {

    int result = 17;

    result = 37
            * result
            + (getSmailReportId() == null ? 0 : this.getSmailReportId()
                    .hashCode());
    result = 37 * result
            + (getOrigId() == null ? 0 : this.getOrigId().hashCode());

    result = 37
            * result
            + (getInternalReportId() == null ? 0 : this
                    .getInternalReportId().hashCode());
    return result;
}
```

As can be seen in the earlier example, 37 is a *magic number*, a numeric literal that is repeated several times, in method *hashCode*, instead of being defined as a constant.

The violation of *Magic Number* rule affects reliability.

The rest of the minor violations are related to rules like *Modifier Order* that imposes the order of modifiers (public, protected, private, abstract, static, final, transient, volatile, synchronized, native, strictfp) suggested in the Java Language specification, *Collapsible If Statements* that checks for nested *if* statements that can be combined, *Constant Name* related with the naming conventions of constants and *Naming - Avoid dollar signs* that advises against the use of dollar signs in variables, methods or classes names. There are also 128 violations that inform the existence of unused imports and unused modifiers.

6.2.2 OOD Metrics (Design & Architecture)

In the dashboard of the project page, we can also see some results related to complexity and object-oriented metrics. We can see in Figure 6.12 that the average cyclomatic complexity per method is 2.6, per class is 20.3 and per file is 26.2 which are high, considering that the threshold normally considered for CC is 10. It can be seen that the average LCOM value per class is 1.1, which means that there are classes with more than 1 set of related methods and fields, therefore these classes must be reviewed. It can also be seen that the average RFC value per class is 21. With respect to package dependencies, the package tangle index is low (2.5%) which means that the project's design is good but can also be improved by cleaning the 1 existing dependency cycle. These can be resolved by cutting 1 dependency between packages, which is formed by 2 dependencies between files.

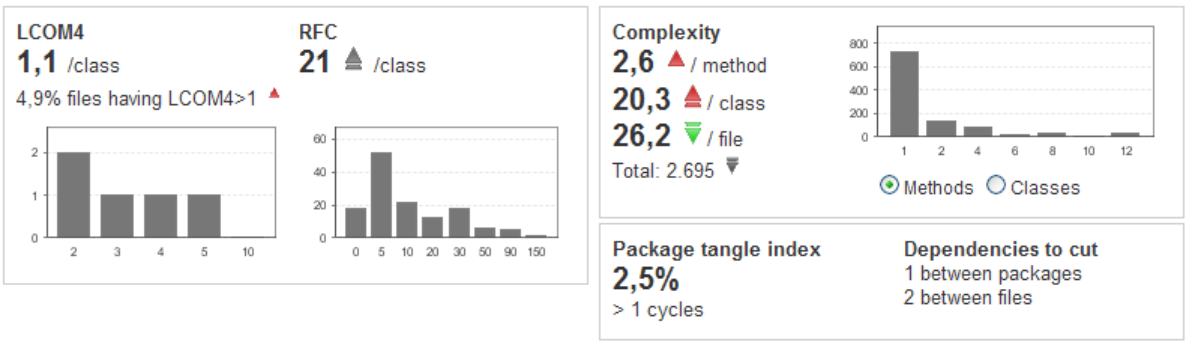


Figure 6.12: SMail: design & architecture

TreeCycle gives an overview of the architecture and design of our project, so it also allows to identify the dependency cycles that the project contains. In Figure 6.13,we can see the dependency tree that contains the cycle and the first thing we notice is that

packages *ejb* and *msgmdb* are the ones responsible for the cycle. By choosing the *Cycle List* option in TreeCycle, we can immediately verify the sets of packages involved in each cycle.

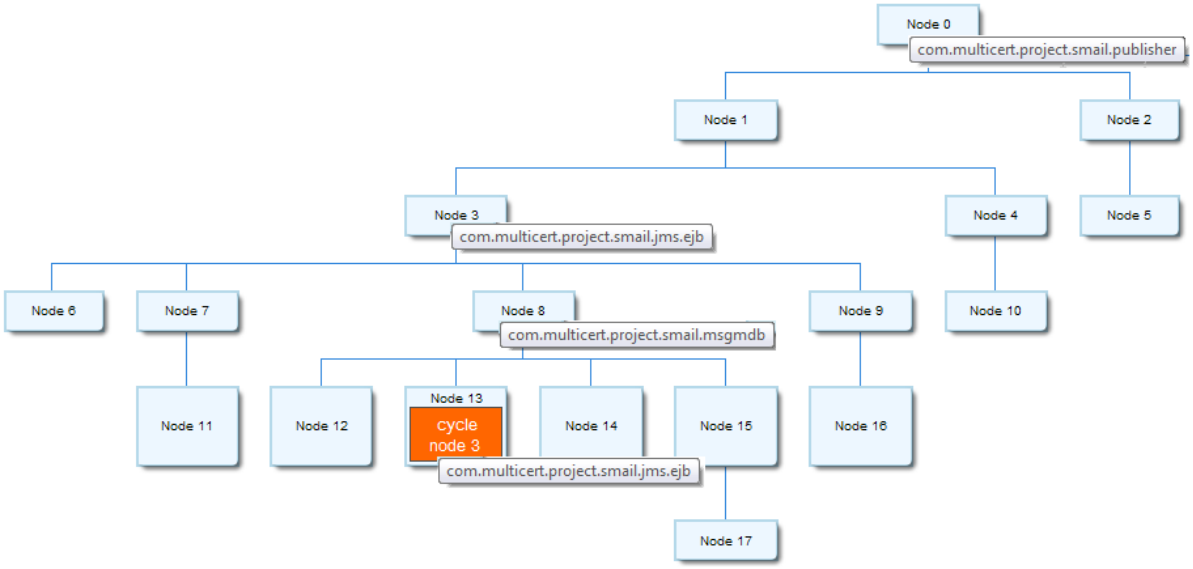


Figure 6.13: SMail: dependency tree

Next, we choose the Design feature from Sonar to get a more detailed description of the dependencies between package *ejb* and *msgmdb* that need to be cut in order to eliminate the cycle. As can be seen in Figure 6.14, there exists one dependency from *msgmdb* to *ejb* and in order to eliminate this package dependency we have to cut the 2 files dependencies between *MsgGeneratorMDB* and *MessageProcessorBean*, and *MsgGeneratorMDB* and *MessageProcessorBeanRemote*.

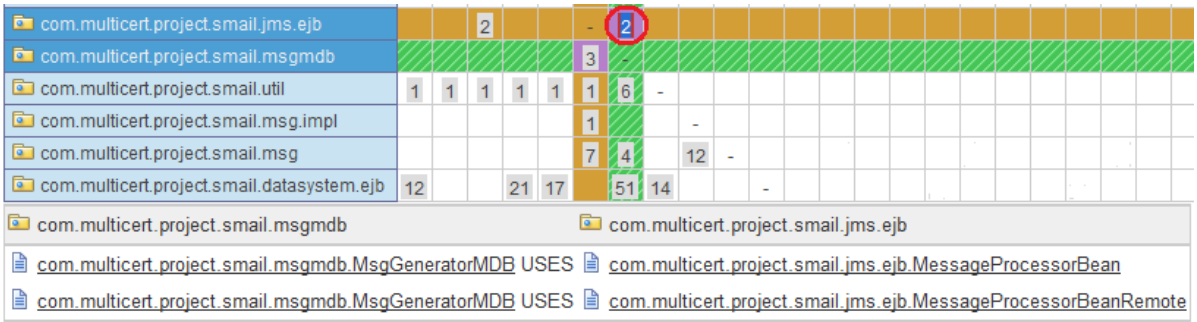


Figure 6.14: SMail: dependency structure matrix

With TreeCycle, we can also drill down to the package level and visualize the C&K metrics results for each metric for each class of a package. In *ejb* exists some classes which

the results exceed defined thresholds. The class with the worst results in this package is *MessageProcessorBean*. As can be seen in Figure 6.15, it obtained the highest WMC (75), o RFC (159), LCOM (4) and CBO (14) results, surpassing even the established thresholds.

Figure 6.15: SMail: C&K metrics results

This high values show us that most of the maintenance effort of the *ejb* package will be spent in this class, because complex classes are harder to analyse, test, replace and modify, therefore this class should be redesigned in order to reduce its complexity and improve the overall quality. In the case of the WMC result, this shows us that the class *MessageProcessorBean* is formed by a huge number of methods, by small number of highly complex methods or both. Indeed this class has 24 methods with 6.3 average CC and it has 3 huge methods, *onSMailMessage*, *setConfiguration* and *setParameters*, with

number of LOCs between 139 and 246. There are two ways of reducing the WMC result and consequently the complexity of this class. The first approach consists in splitting the more complex methods into simpler methods. The second approach is to divide the class in two smaller classes, in fact the LCOM result shows us that this class has 4 sets of highly cohesive methods (instead of the ideal 1), as can be seen in Figure 6.16. This second approach also helps reduce the LCOM result and consequently the complexity of class *MessageProcessorBean*.

Figure 6.16: SMail: lack of cohesion methods

Overall, this project has 11 packages all of them with similar results to the package *ejb* and all of them can be improved by using the suggestions made for class *MessageProcessorBean*. With this changes, the project's total quality will greatly increase.

7 Conclusions and Future Work

7.1 Conclusion

The concept of software quality is not something simple to define, quite the contrary. It is a complex concept with different possible definitions depending on the point of view and interests of each of the actors in the development process. Having identified the concept of quality from the perspective of the software product, we have described the standard ISO/IEC 9126 that identifies the set of quality characteristics and attributes for assessing software product quality and also described the McCall and Boehm quality models who were the main influences of the ISO/IEC 9126.

We have identified the concept of software quality metrics and presented four sets of metrics for object-oriented design (CK, R.C. Martin's, MOOD and L&R metrics) and its specific characteristics. We were also able to relate the object-oriented metrics from these sets to the quality characteristics and attributes identified in the ISO/IEC 9126. Moreover, we have also identified three different approaches to derive metrics thresholds.

With the standard ISO/IEC 14598 we were able to understand where all these concepts (quality models, software quality metrics, thresholds) fall into different types of procedures to evaluate the quality of a software product.

But software metrics are not enough to evaluate or improve the quality of all aspects of a software product (design, architecture, complexity, coding rules, tests). We studied two different techniques, unit testing and static analysis, and were able two understand how these techniques can be used to improve the quality of a software product.

We have made a survey of the tools available to analyse Java source code and to calculate metrics.

To put into practice all the theoretical concepts learned throughout this thesis, we chose a very recent a tool that can be used to analyse and manage source code quality in Java projects. Sonar evaluates source code quality through the use of several different approaches (architecture & design, complexity, duplications, coding rules, potential bugs, unit tests and comments).

We have developed a GWT plugin for Sonar, named TreeCycle. The TreeCycle plugin provides an overall picture of the design quality of Java projects, through the use of software visualisation techniques. It represents the dependencies between packages in a tree graph highlighting it's dependency cycles. For each package it represents in a graphical way (using pie charts) the results of a suite of metrics for object-oriented design (C&K metrics). TreeCycle also features the option to define thresholds for each metric by TreeCycle.

With the help of Sonar and TreeCycle, we assessed the quality of two different Java projects given by the industry, and were able to pin point examples of cases that were contributing to a decline in the quality of both projects. We also proposed simple solutions to resolve these cases.

The concept of software quality is becoming an increasingly important subject in software engineering. The software industry is becoming more competitive which forces software companies to create software with more and higher quality standards. We believe that in this thesis we were able to present a set of methodologies and tools that can easily be used by any software company to considerably improve the quality of its software products.

7.2 Future Work

With this thesis, we think we were able to have an overall idea of the state of the art on quality assessment of Java source code. The next step would be to in pass to the development processes of a software product and try to understand how the concepts and tools learned in this thesis can be applied in these processes and how these would be affected.

The plugins mechanism of Sonar makes it a good framework to implement new ideas. For instance, an interesting idea for a different plugin would be to use the methodology proposed in [1] by using the Sonar database as a benchmark to automatically calculate metrics thresholds.

In relation to our Sonar plugin, the works done by Holten et al. [23], Lanza et al. [40] and Wettel et al. [59] can be used or serve as inspiration for new features that can be implemented in future versions of TreeCycle.

Another interesting feature that could be added to TreeCycle would be the possibility to calculate to calculate all the missing R.C. Martin metrics (instability, abstractness and main sequence) for each package and present its results in the dependency graph.

Bibliography

[1] Tiago Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In *proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. IEEE Computer Society, 2010. To appear.

[2] Ken Arnold, James Gosling, and David Holmes. *Java(TM) Programming Language, The (4th Edition)*. Prentice Hall, 2005.

[3] Nathaniel Ayewah, William Pugh, J. Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Jun 2007.

[4] A Chou B Chelf. The next generation of static analysis: Boolean satisfiability and path simulation - a perfect match. *Coverity White Paper*, 2001.

[5] Sebastian Barney and Claes Wohlin. Software product quality: Ensuring a common goal. In *ICSP '09: Proceedings of the International Conference on Software Process*, pages 256–267, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach. The goal question metric approach. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley, 1994.

[7] Patrik Berander, Lars-Ola Damm, Jeanette Eriksson, Tony Gorschek, Kennet Henningsson, Per Jönsson, Simon Kågström, Drazen Milicic, Frans Mårtensson, Kari Rönkkö, and Piotr Tomaszewski. *Software quality attributes and trade-offs*. Blekinge Institute of Technology, 2005.

[8] P. Botella, X. Burgués, J.P. Carvallo, X. Franch, G. Grau, J. Marco, and C. Quer. Iso/iec 9126 in practice: what do we need to know? pages 1–10, 2004.

[9] British. Standard for software component testing (draft 3.4), April 2001.

[10] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[11] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[12] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[13] Marc Alexis Côté, Witold Suryn, and Elli Georgiadou. Software quality model requirements for software quality engineering. pages 1–16, 2006.

[14] Fernando Brito e Abreu, Rogério Carapuça, and O Brito E Abreu (inesc/iseg. Object-oriented software engineering: Measuring and controlling the development process, 1994.

[15] Karin Erni and Claus Lewerentz. Applying design-metrics to object-oriented frameworks. In *Proc. of the Third International Software Metrics Symposium*, pages 25–26. Society Press, 1996.

[16] Norman E. Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[17] Ronan Fitzpatrick. Software quality:definitions and strategic issues. pages 1–34, 1996.

[18] David A. Garvin. What does product quality really mean. pages 1–19, 1984.

[19] Olivier Gaudin and Freddy Mallet. Sonar. *Methods & Tools*, pages 40–46, Spring 2010.

[20] Paul Hamill. *Unit Test Frameworks, Tools for High-Quality Software Development*. O'Reilly Media, 2004.

[21] R. Harrison, S. Counsell, and R. Nithi. An overview of object-oriented design metrics. In *STEP '97: Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97)*, page 230, Washington, DC, USA, 1997. IEEE Computer Society.

[22] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. *Quality of Information and Communications Technology, International Conference on the*, 0:30–39, 2007.

[23] Danny Holten, Roel Vliegen, and Jarke J. Van Wijk. Visual realism for the visualization of software metrics. In *In VISSOFT'05: Proceedings of 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (2005), IEEE CS*, pages 27–32. Press, 2005.

[24] IEEE. Ieee std 1061-1998: Ieee standard for a software quality metrics methodology, 1998.

[25] ISO/IEC. ISO/IEC 9126: Software product evaluation - quality characteristics and guidelines for their use. *International Organization for Standardization*, 1991.

[26] ISO/IEC. ISO/IEC 14598: Information technology - software product evaluation - part 5: Process for evaluators. *International Organization for Standardization*, 1998.

[27] ISO/IEC. ISO/IEC 14598: Information technology - software product evaluation - part 1: General overview. *International Organization for Standardization*, 1999.

[28] ISO/IEC. ISO/IEC 14598: Software engineering - product evaluation - part 4: Process for acquirers. *International Organization for Standardization*, 1999.

[29] ISO/IEC. ISO/IEC 14598: Software engineering - product evaluation - part 2: Planning and management. *International Organization for Standardization*, 2000.

[30] ISO/IEC. ISO/IEC 14598: Software engineering - product evaluation - part 3: Process for developers. *International Organization for Standardization*, 2000.

[31] ISO/IEC. ISO/IEC 14598: Software engineering - product evaluation - part 6: Documentation of evaluation modules. *International Organization for Standardization*, 2001.

[32] ISO/IEC. Iso/iec 9126-1: Software engineering - product quality - part 1: Quality model. Technical report, Institute of Electrical and Electronics Engineers, 2001.

[33] ISO/IEC. Iso/iec tr 9126-2: Software engineering -product quality -part 2: External metrics. Technical report, Institute of Electrical and Electronics Engineers, 2003.

[34] ISO/IEC. Iso/iec tr 9126-3: Software engineering product quality - part 3: Internal metrics. Technical report, Institute of Electrical and Electronics Engineers, 2003.

[35] ISO/IEC. Iso/iec tr 9126-4: Software engineering product quality - part 4: Quality in use metrics. Technical report, Institute of Electrical and Electronics Engineers, 2004.

[36] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994. Foreword By-Thomas, Brian.

[37] Cem Kaner, Senior Member, and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In *In METRICS 2004. IEEE CS*. Press, 2004.

[38] Karthik.S and Jayakumar.H.G. Static analysis: C code error checking for reliable and secure programming, 2005.

[39] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance*, 15(2):87–109, 2003.

[40] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003.

[41] Rudiger Lincke and Welf Lowe. Compendium of software quality standards and metrics. `http://www.arisa.se/compendium`, 2007.

[42] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects, 2000.

[43] Yashwant K. Malaiya, Michael Naixin Li, James M. Bieman, Senior Member, Senior Member, and Rick Karcich. Software reliability growth with test coverage. *IEEE Transactions on Reliability*, 51:420–426, 2002.

[44] Robert Cecil Martin. Object oriented design quality metrics: An analysis of dependencies. `http://www.objectmentor.com/resources/articles/oodmetrc.pdf`, 1994.

[45] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[46] Inc McCabe Software. Using code quality metrics in management of outsourced development and maintenance, 2009.

[47] Jim A. McCall, Paul K. Richards, and Gene F. Walters. Factors in software quality. volume i. Concepts and definitions of software quality. Technical report, General Electric CO Sunnyvale CA, 1977.

[48] Tim Menzies, Justin S. Di Stefano, Mike Chapman, and Ken McGill. Metrics that matter. In *SEW '02: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, page 51, Washington, DC, USA, 2002. IEEE Computer Society.

[49] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, 1963.

[50] Everald E. Mills, Everald E. Mills, and Karl H. Shingler. Software metrics - sei curriculum module sei-cm-12-1.1, 1988.

[51] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874, 1988.

[52] Roy Osherove. *The Art of Unit Testing.* Manning Publications Co., 2009.

[53] Linda H. Rosenberg and Lawrence E. Hyatt. Software quality metrics for object-oriented environments. 1997.

[54] Jeffrey S. Foster, Michael W. Hicks, and William Pugh. Improving software quality with static analysis. *7thWorkshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 83–84, 2007.

[55] Raed Shatnawi, Wei Li, James Swain, and Tim Newman. Finding software metrics threshold values using roc curves. *J. Softw. Maint. Evol.*, 22(1):1–16, 2010.

[56] Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in Action, Second Edition.* Manning Publications Co., 2010.

[57] TickIT. Getting the measure of tickit. `http://www.tickit.org/measures.pdf`, 2002.

[58] W. T. Tsai, Xiaoying Bai, Ray Paul, Weiguang Shao, and Vishal Agarwal. End-to-end integration testing design. In *Proc. of IEEE COMPSAC*, pages 166–171. IEEE, 2001.

[59] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *In Proc. of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. Society Press, 2007.