TreeCycle: a Sonar plugin for design quality assessment of Java programs

João Miguel Veiga and Maria João Frade

jmrqveiga@gmail.com, mjf@di.uminho.pt

Techn. Report CROSS-10.07-1
2010, July

CROSS
An Infrastructure for Certification and Re-engineering of Open Source Software
(Project PTDC/EIA-CCO/108995/2008)

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

Centro de Ciências e Tecnologias de Computação (CCTC)
Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

COMPETE
QREN QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL
UNIÃO EUROPEIA Fundo Europeu de Desenvolvimento Regional

CROSS-10.07-1
TreeCycle: a Sonar plugin for design quality assessment of Java programs by João Miguel Veiga and Maria João Frade

Abstract

Software quality assessment is a crucial process in software development. To evaluate source code quality it is essential to use tools that help to analyse the code in its different aspects. Sonar is an open source tool used to analyse and manage source code quality.

In this paper we present a Sonar plugin (the TreeCycle) for design quality assessment of Java programs. The TreeCycle plugin represents 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. This plugin provides an overall picture of the design quality of Java projects.

We make a concise presentation of the Sonar platform. We also briefly describe the ISO/IEC 9126 model for software product quality and the main suites of object-oriented design metrics that are use in the plugin.

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 [6,18]. Therefore it all depends very much on the point of view of each person.

Quality is a complex and multifaceted concept. In [10] 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 on 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).

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.

One of the first predecessors of modern quality models was proposed by Jim McCall [26,9] in 1977 for the US military. This model intended to bridge the gap between users and developers by concentrating on a number of software characteristics mapping the users's view with the developer's view. These quality factors are organized in three major perspectives: product revision (ability to change), product transition (adaptability to new environments), and product operations (basic operational characteristics). McCall's quality model has 11 quality factors each linked to several quality criteria (23 total). The quality metrics capture these quality criteria, giving a way of performing the quality assessment of the software.

Another predecessor of modern quality models was proposed in 1978 by Barry W. Boehm [4,3] and uses the same hierarchical approach. At the highest level of Boehm's quality model there are three primary uses representing high-level requirements, which are: as-is utility (ease of use, reliability and efficiency), maintainability (ease to understand, modify and retest) and portability (ease to adapt to new environments). These primary uses breakdown in 7 quality factors representing the qualities expected from a software system, namely: portability, reliability, efficiency, usability, testability, understandability and flexibility. These quality factors are further subdivided in primitive constructs that can be measured and that provide the foundation for defining quality metrics.

There are many other quality models. However McCall's and Boehm's models were the basis for the ISO/IEC 9126 [13,14], 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. We will talk about this standard in the next section.

To evaluate source code quality it is essential to use tools that help to analyse the code in its different aspects. 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 provides as core functionality code analysers, defects hunting tools, reporting tools and a time machine. It enables to manage multiple quality profiles and

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

also has a plugin mechanism giving the opportunity to extend the functionality to the community. Sonar is a very recent tool (it appeared in 2009), but it has already more than forty plugins available. However only four plugins are devoted to visualisation and report of results.

In this paper we present a Sonar plugin (the TreeCycle[2]) for design quality assessment of Java programs. Java [2] is an object-oriented language that is currently one of the most popular programming languages with a large community support. 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.

The rest of the paper is organised as follows. Section 2 gives an overview of ISO/IEC 9126 standard for software product quality. Section 3 is devoted to software metrics with special emphasis on object-oriented design metrics. In Section 4 we briefly describe the Sonar platform. In Section 5 we focus on the TreeCycle plugin, describing how it works and giving an example of its use. In Section 6 we comment on some related work and in Section 7 we conclude and map some directions for further exploration.

2 ISO/IEC 9126: Software Product Quality Standard

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* [13]. 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 [14], external metrics [15], internal metrics [16] and quality in use metrics [17].

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.

[2] http://wiki.di.uminho.pt/twiki/bin/view/Research/CROSS/Tools

– *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 1 illustrates this model. The model is divided in 6 characteristics (quality factors): functionality, reliability, usability, efficiency, maintainability, and portability; which are further subdivided into 27 sub-characteristics (also called attributes or quality sub-factors). The standard also provides more than a hundred metrics that can be used to measure these characteristics. However those metrics are not exhaustive, and other metrics can also be used.

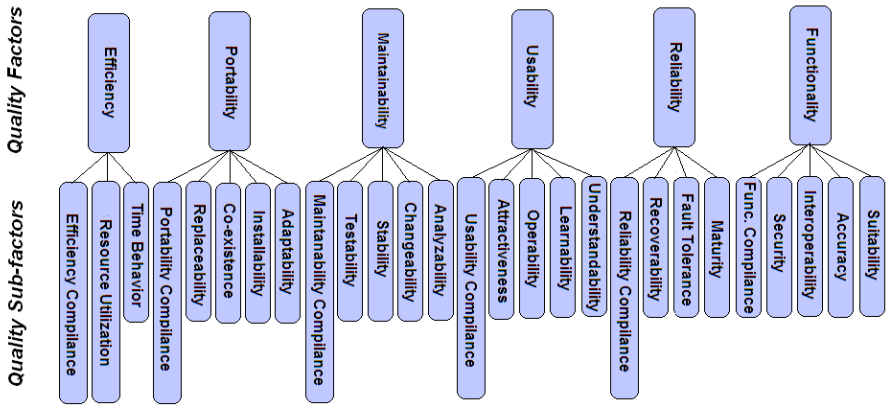

Fig. 1. ISO/IEC 9216 internal and external quality model

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.

Sonar follows the ISO/IEC 9126 to assess the quality of the projects under evaluation. Concretely, the internal metrics norm in which concerns to reliability (the probability of failure), usability (effort to understand, learn the software and also its attractiveness), efficiency (efficient use of computer resources), maintainability (effort necessary to correct, improve or adapt the software to changes), and portability (effort of transferring the software from an environment to another).

The plugin we developed for Sonar is devoted to the visualisation of metrics related to the maintainability, portability, and re-usability. So we will focus on these characteristics. The re-usability is defined by McCall et al. [26] 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 [22].

3 Software Metrics

Metrics are defined as "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" [8]. In software engineering metrics can be used to determine the cost and effort of a software project, staff productivity, and the quality of a software product [8,28].

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) [27,28] is one of the most well-known metrics, and 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 "lines of code" can include comments, non-executable statements and even blank lines. This metric is considered one of the best all-around error predictors [27].

Cyclomatic complexity (CC) metric was developed by Thomas McCabe [25,27] 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.

3.2 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 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 [22]. Next we present two of the most used sets of object-oriented design metrics and their relation with the quality factors described in Section 2. Almost all metrics from these sets of metrics are used by the TreeCycle plugin.

C&K Metrics Suite In 1994 Shyam R. Chidamber and Chris F. Kemerer proposed a metrics suite for object-oriented design [5,29]. 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 of 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.

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 [24,23]. 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 and CE+CA. 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 Metrics Thresholds

When working with software metrics one has to know how to evaluate 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* [21].

Over time many authors proposed software metric thresholds based on their experience. 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.

Erni et al. [7] 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. [30] propose a methodology based on the use of Receiver-Operating Characteristic (ROC) curves to determine software metrics thresholds capable of predicting the existence of different categories of errors. This methodology 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 a 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.

4 Sonar – a Tool for Software Quality Management

Nowadays, we can easily find tools capable of measuring all the software metrics mentioned previously. These tools range from simple command line tools that only output numerical results, to more complete tools with graphical user interfaces that display the results using graphs, in order to facilitate the visualisation of results. 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 tens of software metrics.

4.1 The Sonar platform

Sonar [11] 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 Cobertura, PMD, FindBugs, CheckStyle and Clover, wich 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 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 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.

Sonar uses the ISO/IEC 9126 quality model to divide the coding rules in 5 quality factors: maintainability, usability, efficiency, portability and reliability. It is also possible to define and manage multiple quality profiles adapted to different projects. These profiles consists of:

- activate/deactivate and weight coding rules;
- define metrics thresholds, enabling automatic alert;
- define projects associated to each profile.

Sonar is a web-based application. Quality profiles settings can be configured on-line. It provides a dashboard, as seen in Figure 2, that gives us an idea of a project overall quality. It is also possible to drill down to visualize the software project's code quality at different levels (packages, classes, source code).

Sonar can be used for audits, however all its potential is reached when used as a shared central repository for quality management enabling to improve code quality 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.

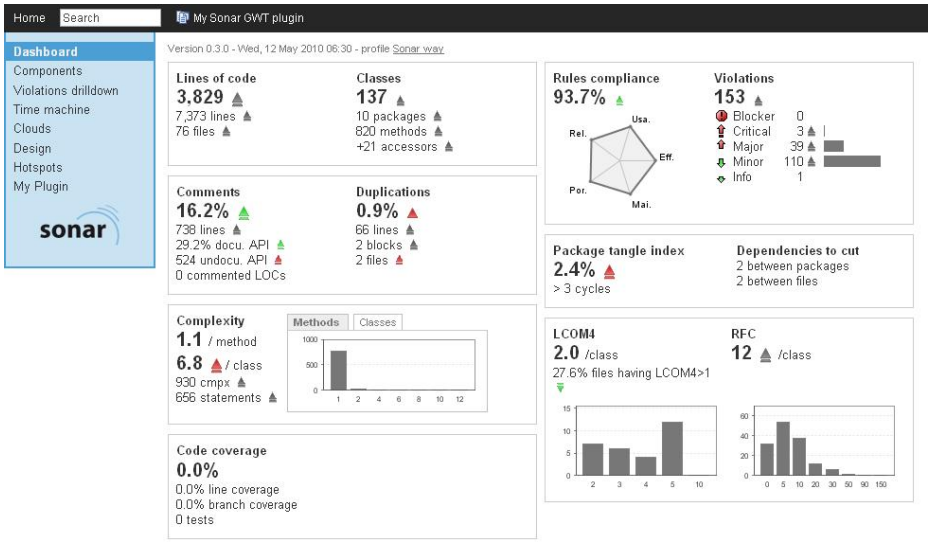

Fig. 2. Example of a dashboard of a project in Sonar

5 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" [19]. Studies [19] 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. There are several projects that attempt to combine software visualisation and software metrics [12,20,31].

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 report of results. We think that Sonar could evolve further in this area and we have built a plugin for visualisation of information that concerns to the design quality of Java projects.

Our plugin 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 Sonar DSM.

5.1 How it Works

A Sonar plugin allows us to define new extensions like new metrics to be calculated and collected, new Ruby on Rails widgets to display the new metrics results in the dashboard, and sensors and decorators to gather and process all the new defined metrics.

A sensor collects and analyses data but has no access to other plugins collected data. A decorator, in addition to collect and analyse data, is allowed to cross reference data collected from other plugins. It is also possible to define Google Web Toolkit (GWT) web pages that allows to add more complex features to Sonar, as is the case of the TreeCycle plugin that displays a full page tree graph with all the dependencies between packages of a project.

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? We chose to display 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 to be very large and sometimes do not even fit 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. We use the source code of the TreeCycle plugin as an example. The tree graphs generated for our source code consists of a main tree containing 35 nodes, and 2 smaller trees, containing 1 and 15 nodes. In Figure 3 it can be seen the main tree that has node 0 (*Veigh.TreeCycle.page.client*) as its root. 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.

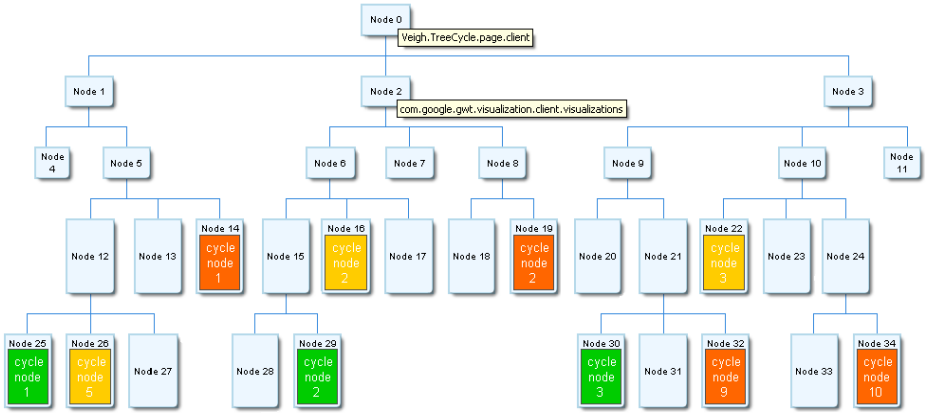

Fig. 3. TreeCycle: package dependencies tree graph

Choosing the tree root. Initially, all packages are candidates to be the roots of the tree graphs that will be generated. What we do 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 we begin 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 [24], because these tend to generate spaghetti code. An important feature of our plugin is the highlighting of dependency cycles between packages in the tree graphs. In Figure 3 it can be seen some leafs highlighted with different colors. 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 depencency cycle can be reflected in a tree several times. To isolate cycles we identify each cycle with a different color (for instance, the tree in Figure 3 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 4.

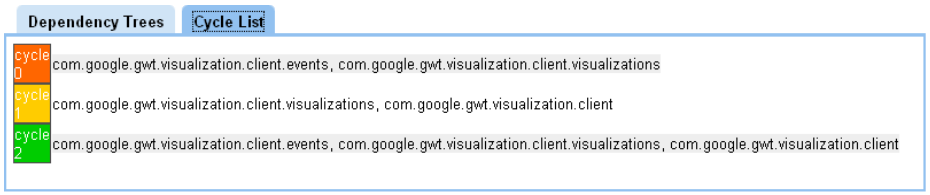

Fig. 4. TreeCycle: list of dependency cycles

Isolating components. Besides giving an 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 package *com.google.gwt.visualization.client* (node 2) we have to include all the nodes standing bellow it (nodes 6, 7,

8, 15, 16, 17, 18, 19, 28 and 29). Actually this sub-tree represents the Google Chart Tools 1.1 library used by the TreeCycle plugin.

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 by the Google Chart Tools 1.1 library. For example, in Figure 5 it can be seen the metrics results for all the classes from package *Veigh.TreeCycle.client.page*, represented in Figure 3 as node 0. 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. The *orgChart* class is the one with the higher values for WMC, RFC and CBO 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.

6 Related Work

In this section we comment on some works related with software visualisation whose ideas can be used or serve as inspiration for new features that can be implemented in future versions of the TreeCycle plugin.

Holten et al. [12] 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, colors, textures and bump mapping.

Lanza et al. [20] 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. [31] 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

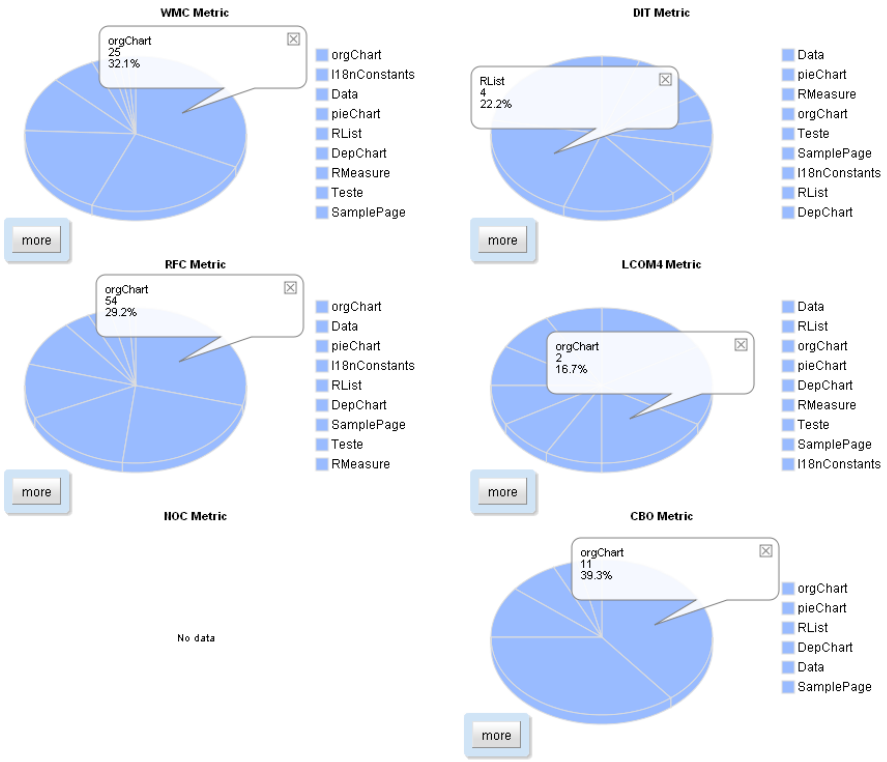

Fig. 5. TreeCycle: C&K metrics

approach represents metrics by the size (width and height) and color of both buildings and districts.

7 Conclusions and Future Work

We have presented a Sonar plugin that provides an overall picture of the design quality of Java projects. 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 (using pie charts) the results of a suite of metrics for object-oriented design (C&K metrics).

Our plugin adds to Sonar a different way of viewing results of projects analysis through the use of software visualisation techniques. However, despite all its features, the TreeCycle plugin is still under development. In the near future we count on launching new versions of the plugin with new features.

One of these features will be the option to define thresholds for each metric in the TreeCycle plugin. Whenever a metric result exceeds a threshold an alert will be issued in the dependency tree graph and/or in the pie charts. These alerts can be represented by different colors (for example green, yellow or red) depending on the degree of risk. Another interesting feature would be to calculate all the missing R.C. Martin metrics (instability, abstractness and main sequence) for each package and present its results in the dependency tree graph. Although these features are obvious, we do not have implemented it yet due to technical issues that we count to solve in a very near future.

The features of the Sonar tool make it a good framework to implement new ideas. For instance, an interesting idea for a different plugin for Sonar would be to use the methodology proposed in [1] and using the Sonar database for the benchmark.

Acknowledgments We thank Joost Visser for showing us the Sonar platform. This work was supported by the Portuguese Foundation for Science and Technology (FCT), in the context of the CROSS project, under contract PTDC/EIA-CCO/108995/2008.

References

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. 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.
4. Barry Boehm, J.R. Brown, H. Kaspar, M. Lipow, G. McLeod, and M. Merritt. *Characteristics of Software Quality*. North Holland, 1978.
5. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.
6. Marc Alexis Côté, Witold Suryn, and Elli Georgiadou. Software quality model requirements for software quality engineering. In *Software Quality Management - International Conference*, pages 31–50, 2006.
7. 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.
8. Norman E. Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.
9. Ronan Fitzpatrick. Software quality: Definitions and strategic issues. Technical report, Staffordshire University, School of Computing Report, 1996.

10. David A. Garvin. What does product quality really mean. *MIT Sloan Management Review*, 26(1), fall 1984.
11. Olivier Gaudin and Freddy Mallet. Sonar. *Methods & Tools*, pages 40–46, Spring 2010.
12. 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.
13. ISO/IEC. ISO/IEC 9126: Software product evaluation - quality characteristics and guidelines for their use. *International Organization for Standardization*, 1991.
14. ISO/IEC. ISO/IEC TR 9126-1: Software engineering - product quality - part 1: Quality model. *International Organization for Standardization*, 2001.
15. ISO/IEC. ISO/IEC TR 9126-2: Software engineering - product quality - part 2: External metrics. *International Organization for Standardization*, 2003.
16. ISO/IEC. ISO/IEC TR 9126-3: Software engineering - product quality - part 3: Internal metrics. *International Organization for Standardization*, 2003.
17. ISO/IEC. ISO/IEC TR 9126-4: Software engineering - product quality - part 4: Quality in use metrics. *International Organization for Standardization*, 2004.
18. Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994. Foreword By-Thomas, Brian.
19. Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance*, 15(2):87–109, 2003.
20. Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003.
21. Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
22. Rudiger Lincke and Welf Lowe. Compendium of software quality standards and metrics. `http://www.arisa.se/compendium`, 2007.
23. Robert Cecil Martin. Object oriented design quality metrics: An analysis of dependencies. `http://www.objectmentor.com/resources/articles/oodmetrc.pdf`, 1994.
24. Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
25. Inc McCabe Software. Using code quality metrics in management of outsourced development and maintenance, 2009.
26. 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.
27. 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.
28. Everald E. Mills, Everald E. Mills, and Karl H. Shingler. Software metrics - sei curriculum module sei-cm-12-1.1, 1988.
29. Linda H. Rosenberg and Lawrence E. Hyatt. Software Quality Metrics for Object-Oriented Environments. In *Proceedings of National Conference on Challenges & Opportunities in Information Technology (COIT-2007)*, March 2007.

30. 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.
31. 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.