International Journal of Business, Human and Social Sciences
ISSN: 2517-9411
Vol:4, No:3, 2010

Evaluating Refactoring with a Quality Index

Crt Gerlec, Marjan Hericko

***Abstract*—**The aim of every software product is to achieve an appropriate level of software quality. Developers and designers are trying to produce readable, reliable, maintainable, reusable and testable code. To help achieve these goals, several approaches have been utilized. In this paper, refactoring technique was used to evaluate software quality with a quality index. It is composed of different metric sets which describes various quality aspects.

***Keywords*—**Refactoring, Software Metrics, Software Quality, Quality Index, Agile methodologies

I. INTRODUCTION

REFACTORING is a technique used for reconstructing existing source code and changing its internal structure without changing its external behavior[1]. The process essentially deals with the cleaning of source code to improve software design, raise the readability level, ease the maintaining process, find bugs and help to understand program faster. Refactoring is an integral part of the software development cycle in agile methodologies [12] (e.g. extreme programming). Developers first write tests, then write the source code to make the tests pass, and finally refactor the code to improve its internal consistency and clarity.

It has been proven that software metrics reflect software quality [2][2]. They have been widely used in software quality measurements. The results of these evaluations indicate which pieces of software need to be reengineered. Furthermore, developers and designers strive to achieve higher software quality after making source code changes (refactoring). However, several quality metrics have been proposed that describe different software quality aspects. In this research, a quality index metric has been applied to evaluate the refactoring impact on software quality.

II. RELATED WORK

Researchers have often studied refactoring and described its correlation to software systems. Mohammad Alshayeb[3] has assessed the effect of refactoring in different external quality attributes (adaptability, maintainability, understandability, reusability and testability) in order to decide whether the cost and time put into refactoring are worthwhile [3]. The effect has been measured by applying software metrics. Wilking et al. [4] researched the effects of refactoring on maintainability and modifiability [4]. Maintainability was tested by inserting bugs into the code and measured the developers' time to find and solve them. Modifiability was tested by adding new requirements and measuring the time needed to fulfill them. Bois et al.[5] investigated program comprehension using refactoring to understand and traditional reading to understand patterns [5]. They concluded that refactoring can be used to improve the understanding of a software system. Fowler also defined a catalog containing information on how and when to do refactoring[1]. Wake even identified smells within classes and smells between classes [6]. Furthermore, he also described how to recognize important smells and how to apply refactoring techniques to remove these smells[6].

C. Gerlec is with the Institute of Informatics, Faculty of Electrical Engineering and Computer Science at the University of Maribor, Smetanova ul. 17, 2000 Maribor, Slovenia (e-mail: crt.gerlec@uni-mb.si).

M. Hericko is with the Institute of Informatics, Faculty of Electrical Engineering and Computer Science at the University of Maribor, Smetanova ul. 17, 2000 Maribor, Slovenia (e-mail: marjan.hericko@uni-mb.si).

III. PRODUCT METRICS

Measurement and software data collecting is an essential source of information in computer science. A correct interpretation allows engineers to understand their software's behavior and recognize common patterns.

Fenton defined product metrics as measured facts or the documented results of the software development process [8][8]. He believes that measurement provides important information about code quality, processes and changes in a software product.

Commonly used product metrics are:

- Cyclomatic Complexity (CC) – this metric represents the complexity of a method and complexity of a class. The metric value should be as low as possible. Higher values (more than 20) indicate that the software is hard to maintain, understand and that the degree of readability is low.
- Maintainability index (MI) – This measures software maintainability. The metric consists of several elementary metrics. Two versions are frequently in use. The first version uses 3 elementary metrics and the second uses 4 (the additional metric comprises class comments).
- Coupling between objects (CBO) – A class is coupled with another if its methods use the attributes of the other class. If the class is coupled with several different classes, its reusability and understandability is low. Normally the classes should have low coupling in order to achieve

modularity.

- Number of children (NOC) – indicates the number of sub-classes in the hierarchy. The greater the number, the better the reuse. However, a large number of children could also indicate improper abstraction.
- Depth of Inheritance Tree (DIT) – This measures the depth of the inheritance of the class. Inheritance increases the class efficiency by reducing the redundancy. However, deep hierarchy could also lead to lower understandability and predictability.
- Weighted Methods per Class (WMC) – This is the sum of the complexities of the methods of a class. The metric indicates how much time and effort is needed to develop and maintain a class. A large number of methods increase overall complexity.
- Response for a Class (RFC) – This is the sum of the class methods and the total number of other methods that are invoked from origin class methods. A higher value raises the maintenance level.
- Lack of Cohesion in Methods (LCOM) – This is the number of disjoint sets (intersection of the set of the attributes) of local methods reduced by the number of methods using at least one shared attribute. A high LCOM value indicates that the class should be split into two or more sub classes.
- Size related metrics – Size metrics play an important role in software measurement. They can be divided into two groups:
    - The first group contains class level metrics: the number of methods, properties, constructors, nested classes, data fields, events, attributes and the number of all instructions.
    - The second group (method level) consists of the number of parameters, local variables, exception blocks, maximum stack size, number of instructions, number of all operators in the method, number of distinct operators and number of operands.

IV. REFACTORING AND QUALITY INDEX

A. Refactoring

Developers use refactoring to improve the internal quality of software systems. Before each refactoring phase, they have to identify bad code design (called bad smells) [1]. This step is usually done by experienced developers. When an inappropriate code is identified, the correct refactoring method is used to clean bad or dangerous code. Indicators for such source code are:

- redundant source code,
- long classes,
- long methods,
- switch cases,
- long parameters lists,
- unreadable and unclear source code…

While the refactoring methods are:

Extract method: This method extracts selected source code from the code block of an existing member. The new method contains the selected source code and the source code in the existing member is replaced with a call (new method). Extracting the source code into the methods raise the readability and reusability level.

Encapsulate field: Other objects can access public fields and change them freely. In such a scenario, the owning object does not have control over its fields. By encapsulating them through the use of properties, direct access is disabled.

Extract interface: The refactorinq technique creates a new interface with members that originate from an existing class, structure or even interface.

Rename: Provides an easy way to rename identifiers for code tokens (symbols) such as fields, properties, method names, local variables, namespaces etc. Refactoring changes the names in comments, declarations and identifier calls.

Promote local variable to parameter: Operations move a local variable from the method's body to a method, indexer or constructor parameter and all method calls should be updated.

Add parameter: The operation adds a parameter to methods, indexers or delegates. All declarations are changed at any location where the member is called.

Remove parameter: The operation removes a parameter from methods, indexers or delegates. All declarations are changed at any location where the member is called.

Reorder parameter: The operation changes the order of the parameters for methods, indexers or delegates. All declarations are changed at any location where the member is called.

Move method: The operation creates a new method with a similar body in the class it uses most. Old method could be turned into a simple delegation.

Move fields: The operation creates a new field in the target class and changes all its users. Previous field is deleted from the originate class.

B. Quality Index

Developers, software designers and project managers strive to develop quality software with little to no bugs. Furthermore, the source code should be maintainable and adaptable. In order to achieve and evaluate these goals, several approaches can be used. Hericko et al. proposed the quality index metric [9][10][13], which expresses source code quality, and is defined as:

$$QI = \frac{\sum_{i=1}^{n} PMQR_i}{n}$$
$$PMQR \in \{1..5\}$$
$$PMQR_i = f_i(mv_i),$$

where PMQR is the product metric quality rating, n is the number of code metrics used in the calculation, mv is the code metric value and f is the function that transforms the metric to the product metric quality rating.

The proposed metric consists of n software product metrics. Depending on project and environment characteristics, these metrics should be defined. Each product metric has its threshold values which are calibrated according to the project needs. The project attributes are:

- development team (experience level),
- development environment,
- domain (insurance, banking, etc.),
- customer and
- development type (research projects, new development…).

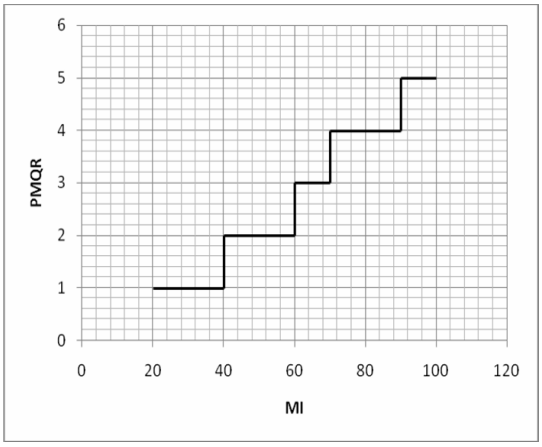

Fig 1 An example of the function f for the MI metric

Product metrics used in the calculation should be non-complementary (e.g. a complementary pair consists of a depth of inheritance tree (DIT) and the number of children (NOC)) and non-correlated (e.g. correlated metrics are weighted methods per class (WMC), coupling between objects (CBO), response for a class (RFC)). When the final metric set is chosen, n transformation functions f need to be defined (Fig 1) that transform metric values to the metric quality ratings (PMQR). Their values are within one and five and metric values are metrics specific.

TABLE I
QUALITY INDEX METRIC SETS

Quality metric number	METRIC SETS USED FOR CALCULATING QUALITY INDEX
1	DIT, CBO, LCOM, MI
2	WMC, DIT, CBO, LCOM
3	NOC, CBO, LCOM, WMC
4	DIT, NOC, LCOM, WMC
5	NOC, RFC, LCOM, MI
6	CBO, NOOM, NOC, WMC
7	DAC, NOC, WMC, NOOM
8	DAC, DIT, MPC, LCOM
9	RFC, DIT, LCOM, MPC
10	NOC, LCOM, CBO, MPC
11	NOC, LCOM, NOOM, MPC

NOOM (number of overridden methods), DAC (data abstraction coupling), MPC (message passing coupling).

V. MEASUREING QUALITY INDEX

In order to evaluate the quality index before and after the refactoring, its metric sets have to be defined. 11 different metric sets (Table 1) have been used in the research. These sets consist of non-correlated and non-complementary metrics[7].

The next step is to define the quality rating (PMQR functions) for all product metrics. Ratings are project specific and should be defined based on project needs and characteristics. In our research, PMQR functions described in [7][7] have been used and they are shown in Table 2.

Each metric included in the quality index has its own specific quality intervals that have been recommended by metric authors or researchers[7]. For instance, for the maintainability index, it is known that a higher value indicates better software quality and better software design. According to this definition, quality intervals have been proposed [11]. Values higher than 90 indicate good software design, so that adding new functionalities or changing the existing source code is easier. On the other hand values lower than 40 mean a low quality of software design and software renewal is required.

The third step is to calculate the quality index (for all variants) on the origin source code (with no refactoring operations). Four different applications written in C# have been used to evaluate quality. All applications are relatively small. The first two applications are the largest while the second two are smaller.

TABLE II
METRICS AND TRANSFORMATION FUNCTIONS (PMQR)

PMQR	MPC value	DAC value
5	30 - 35	1,2 - 1,4
4	0 - 30	0 - 1,2
3	35 - 50	1,4 - 1,7
2	60 - 70	1,7 - 2
1	>70	> 2

PMQR	NOOM value	RFC value
5	0 - 3,1	45 - 60
4	3,1- 4,5	30 - 45
3	4,5 - 5,5	60 - 80
2	5,5 - 6	20 - 30
1	> 6	0 - 20 & > 80

PMQR	NOC value	WMC value
5	2 - 2,5	7 - 9
4	1,75 - 2	5 - 7
3	1,25 - 1,75	9 - 12
2	1 - 1,25	3,5 - 5
1	0 - 1	0 - 3,5 & > 12

PMQR	MI value	LCOM value
5	> 90	0 - 10
4	70 - 90	10 - 25
3	60 - 70	25 - 40
2	40 - 60	40 - 60
1	20 - 40	> 60

PMQR	CBO value	DIT value
5	0 - 2,5	2,5 - 3,6
4	2,5 - 5	1,6 - 2,5
3	5 - 7,5	3,6 - 7,2
2	7,5 - 10	0 - 1,6
1	> 10	> 7,2

The next step is the “refactoring phase”. All four applications have been modified with refactoring techniques. In 3 cases, the lines of code have been increased and in one case the number of lines has been decreased. First, the application has had 21% more lines after refactoring, the second 139% and the third 31%. The fourth application ended with 25% less source code after refactoring.

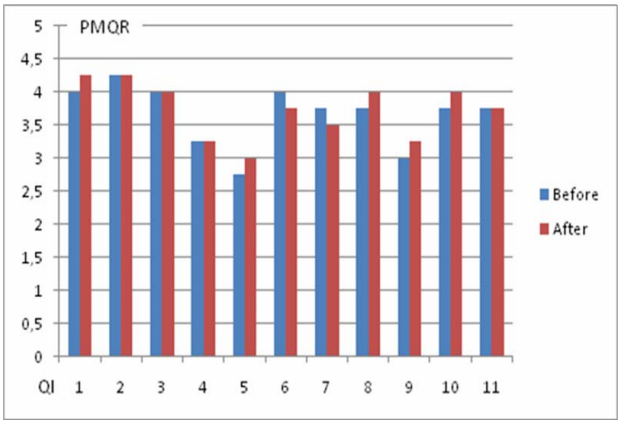

Fig. 2 The quality index of the first application

Fig. 2 (first application) shows that the quality index increased in 5 cases, quality index 6 and 7 were higher before the refactoring and in 4 cases the values stay the same. However, the metric sets for the quality index 6 and 7 are composed by similar metrics. Three metrics are the same (NOOM, NOC, WMC) and only one metric is different (CBO and DAC). The impact of these tree metrics is bigger than the rest metric and it is obvious that the final quality index will be similar in both cases.

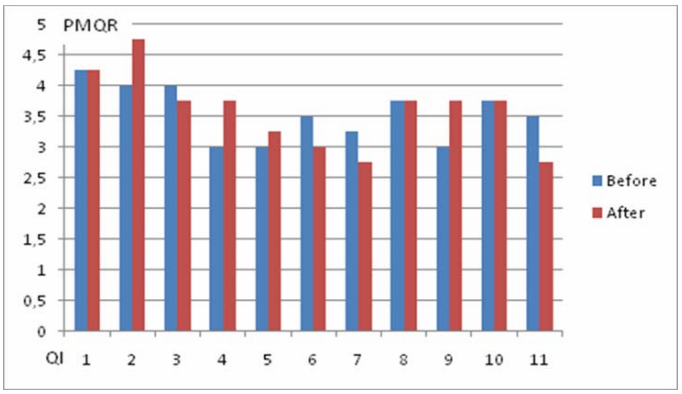

Fig. 3 Quality index of the second application

The quality index of the second application (Fig. 3) is higher and lower after the refactoring in 4 cases. In 3 cases the quality stays the same. There are four bigger jumps in quality. In cases 2, 4 and 9 the quality was raised and in case 11 the quality dropped. Different metrics are used in these cases and there is no recognizable pattern within the metrics sets.

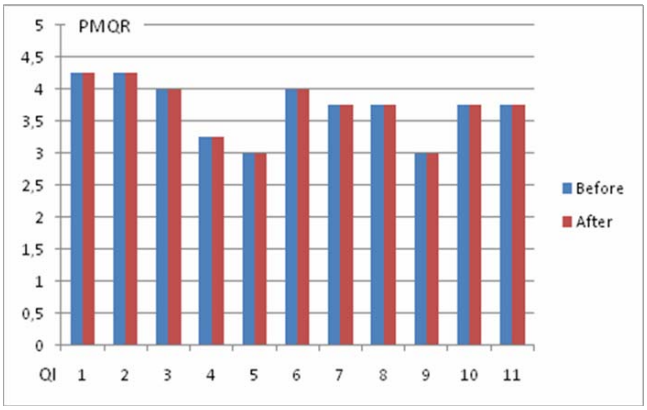

Fig. 4 Quality index of the third and fourth application

The quality of the third and fourth application (Fig. 4) is the same before and after the refactoring for all cases. One of the causes could be their length because the last two applications have the lowest LOC (lines of code) value among all researched programs.

Future work

In this research, only the refactored applications have been observed without knowing the refactoring techniques that have been applied. The next task is to analyze refactored source code in order to detect which refactoring approaches have been applied. By knowing the approaches, their impact on the quality index could be explored. Furthermore, the impact of different metric sets (within the quality index) and correlated ranking functions could also be observed.

The next idea is to compose the quality index that contains metrics (and proper PMQR functions) that expresses the readability or reusability degree of software. Intuitively, it is expected that the quality index should be higher after the refactoring phase.

Conclusion

Our research has shown that refactoring techniques do not always improve software quality. Some metrics have a negative score after refactoring and if such metrics compose the quality index, its value will be lower after the source code transformation. Intuitively, it is expected that software quality will be higher and it also was in most cases.

The aim of the research was also to check the difference between the quality index sets and to get some useful information about their correlation. However, this final idea is beyond the scope of this paper.

References

[1] M. Fowler, J. Brant, W. Opdyke, D. Roberts, “Refactoring: Improving the Design of Existing Code”, Addison Wesley, 1999.
[2] B. W. Boehm, J. R. Brown, M. Lipow, “Quantitative Evaluation of Software Quality”, Proceedings of the 2nd international conference on Software engineering, pp. 592-605, 1976.
[3] M. Alshayeb , “Empirical investigation of refactoring effect on software quality”, Information and Software Technology, vol. 9, no. 9, pp. 1319-1326, 2009.
[4] D. Wilking, U.Kahn, S. Kowalewski , “An Empirical Evaluation of Refactoring”, e-Informatica Software Engineering Journal, vol. 1, no. 1, 2007.
[5] B. Bois, S. Demeyer, J. Verelst, “Does the "Refactor to Understand" reverse engineering pattern improve program comprehension?”, Proceedings of the Ninth European Conference on Software Maintenance and Reengineering, pp. 334-343, 2005.
[6] W.C. Wake, Refactoring Workbook, Addison Wesley, 2003.
[7] A. Zivkovic, U. Goljat, M. Hericko, “Improving the usability of the source code quality index with interchangeable metrics sets”, Information Processing Letters, vol. 110, no. 6, 2010.
[8] N. Fenton, S. L. Pfleeger, “Software Metrics: A Rigorous and Practical Approach”, Thomson Computer Press. 1994.
[9] C. Gerlec, A. Zivkovic, “Software Metrics Repository Architecture”, Collaboration, software and services in information society, 2009.
[10] M. Hericko, A. Živkovic, P. Porkoláb, “A method for calculating acknowledged project effort using a quality index”, Informatica. Vol. 31. No. 4. 2007.
[11] M. Andersson, P. Vestergren, “Object-Oriented Design Quality Metrics”, Computing Science Department Uppsala University.
[12] R.C. Martin, “Agile Software Development, Principles, Patterns, and Practices”, Prentice Hall, 2002.
[13] A. Krajnc, M. Hericko, U. Goljat, “Measuring the Advantages of the Software Factories Approach”, 5th Central and Eastern European Software Engineering Conference in Russia, 2009.