Effect of Object Oriented Refactorings on Testability, Error Proneness and other Maintainability Attributes

György Hegedűs
Department of Software Engineering, University of Szeged
H-6720 Szeged, Árpád tér 2, Hungary
hgy@inf.u-szeged.hu

György Hrabovszki
Department of Software Engineering, University of Szeged
H-6720 Szeged, Árpád tér 2., Hungary
Hrabovszki.Gyorgy.Janos@stud.u-szeged.hu

Dániel Hegedűs
Department of Software Engineering, University of Szeged
H-6720 Szeged, Árpád tér 2., Hungary
Hegedus.Daniel.Ferenc@stud.u-szeged.hu

István Siket
Department of Software Engineering, University of Szeged
H-6720 Szeged, Árpád tér 2., Hungary
siket@inf.u-szeged.hu

ABSTRACT

Refactoring (object-oriented) code aims to improve some of the quality attributes of the software system under maintenance. However, as any other changes to a system, refactoring actions may have side-effects too, which need to be taken carefully into account during their implementation. Consequences of refactoring are only moderately discussed in literature. In this work, we emphasize the importance of documenting such consequences, by reviewing relevant literature and giving our views on the topic. We do this in three steps: first, we investigate how high level quality attributes (including testability and other maintainability aspects based on the ISO 9126 standard and fault proneness) can be estimated based on measurable metrics in the code, like complexity, size, and coupling. In the second step, we examine what effect of individual refactoring techniques can have on such metrics. Finally, we combine these findings to get a view on how refactoring can influence high level quality characteristics. With this work, we want to foster discussion in this important topic, rather than giving a solution to the problem, as it requires a vast amount of further research by the testing and software quality communities.

Keywords

refactoring, testability, error proneness, ISO 9126 quality model, code metrics, static software measurement

ECOOP '2010 Maribor, Slovenia EU

1. INTRODUCTION

Although source code refactoring encompasses many widely used techniques to improve the structure of existing code, the advocates of those techniques do not even mention or just rarely mention the side effects of these changes. This means that we do not know much about the results of refactoring. For example, many quality characteristics (e.g. *testability* or *error proneness*) might be changed in the wrong direction. Therefore the quality of the software does not increase unambiguously. The fact is that software quality characteristics are contradictory. A good example is object-orientation itself which improves the flexibility of the code at the cost of performance. It is a proven fact that object-oriented systems require more resources than procedural ones. Improving the reliability of code by including more exception handling also decreases the readability of the code. There is little one can do with code to improve one quality without having a negative effect on some other quality. Achieving high quality software is always a tradeoff between conflicting quality goals [22].

Testing has two goals, one is to install confidence in the product and the other is to find as many bugs as possible before the product is released. Therefore, we should concentrate our testing effort on the most fault prone source code elements. The problem lies in selecting them in advance. Many researchers have made investigations on the estimation of the number of bugs before testing. For example, Basili *et al.* [2] analyzed six metrics on a medium sized system defined by Chidamber and Kemerer [5]. They found that both *structural complexity* and *coupling* were useful for fault prediction but coupling seemed to be better. Although *coupling* is essentially a kind of a high level complexity measure, we use this terminology as it is widely used in object oriented metrics community.

The quality characteristics of a software system can be determined from source code metrics like coupling and complexity. A typical way to measure the quality of existing source code is to compare it against the ISO/IEC standards and common product metrics for complexity and size. Heitlager *et al.* [12] formulated a maintainability model, which links high level maintainability characteristics to code level measures. They found that the *complexity* of source code units had a negative influence on the *changeability* and *testability* of the system.

Since testing effort is costly and limited, the testing process should be optimized in some way to decrease its cost and to improve its efficiency [3, 10, 14, 18]. One possible way is to decrease the number of test cases without decreasing the quality of the test. Refactoring usually increases the number of test cases and their instances, so one has to be cautious about the effect of refactoring on testability. For example, the *Extract Method* refactoring technique creates additional methods which have to be tested. This aspect of refactoring has been neglected in the literature on refactoring.

Our goal is to investigate the direct effects of refactoring on unit testability and other source code quality characteristics. In this way we can analyze how refactoring affects maintainability. We give an initial step of a complex methodology based on the previous aspects to optimize the usage of refactoring. The purpose of this study is to summarize the results related to this topic and to open up a debate on the effect of refactoring on testability. We believe this is to be an important topic even though only a few studies have dealt with it [11, 16, 20]. In spite of the fact that our results are not yet supported by empirical evidence, the outcome of our study reflects the risk of such code transformations.

Our paper is structured as follows. In Section 2 we describe the relationship between low level system properties and high level quality characteristics. In Section 3 the refactoring techniques and their effect on low level system properties are demonstrated. In Section 4 we summarize the effects of refactorings on high level system characteristics. In the final section, we draw some conclusions from the study.

2. ANALYZING THE HIGH LEVEL QUALITY CHARACTERISTICS

First, we discuss in detail how *structural complexity*, *coupling* and *size* metrics can be used for fault prediction. Next, we extend the set of metrics with *code duplications* and *unit test effort*, and later we refer to this group of measures as *low level system properties*. We investigate their effects on *fault proneness* and *maintainability sub-characteristics*. For the sake of simplicity, fault proneness and maintainability sub-characteristics are referred to *high level quality characteristics*.

2.1 Fault prediction capabilities of metrics

Now we review the relationship between the metrics and the fault prone properties of the classes by summarizing a few experiments. Although a lot of different metrics have been defined, only *structural complexity*, *coupling*, and *size* ones are examined in this work.

Basili *et al.* [2] analyzed the Chidamber and Kemerer complexity metric suite [5] on a medium sized system. They found that both structural complexity and coupling are useful for fault prediction but coupling seemed to be better. On the other hand, size metric was not considered in their study. Yu *et al.* [25] chose a client side of a large network service management system and examined the fault prediction capabilities of ten metrics. They found that the structural complexity and size metrics were the best indicators of fault probability, but the examined coupling metric also correlated to the fault density. Subramanyan and Krishnan [24] chose a relatively large B2C e-commerce application suite developed in C++ and Java. Among others metrics, size, structural complexity, and coupling metrics were examined to see how well they suit for fault-prediction. Size was found to be a good predictor in the case of C++ and Java code, but coupling and structural complexity were useful only for C++ code. In the case of Java, structural complexity and coupling were not useful for fault-prediction. Gyimóthy *et al.* [9] investigated the fault-proneness capabilities of the Chidamber and Kemerer metric suite as well as code size in lines of code on the open source Web and e-mail suite called Mozilla [15]. The coupling metric was the best indicator, followed by the size metric. The structural Chidamber and Kemerer metrics ranked on third place. Olague *et al.* [17] analyzed how the Chidamber and Kemerer metrics [5], the Abreu's metrics for object-oriented design [6], and Bansiya and Davis' quality metrics for object-oriented design [1] could be used for fault-prediction. In their study six different versions of Rhino [19] were analyzed. Both structural complexity and coupling were found to be good predictors, but complexity ranked higher than coupling. Finally, we take into account the results of a survey, where the predictability of structural complexity, coupling, and size were compared [21]. The experiment was carried out at the Software Engineering Department of the University of Szeged. 50 participants were asked to rank these categories according to their importance from the view point of fault-prediction capabilities. According to that experiment, structural complexity is the most important predictor, while the other two measures are more or less equally important, but both significantly less important than structural complexity.

Study	Complexity	Coupling	Size
Basili [2]	2	1	n.a.
Yu [25]	1	3	2
Subramanyan [24]	2-3	2-3	1
Gyimóthy [9]	3	1	2
Olague [17]	1	2	n.a.
Siket [21]	1	2-3	2-3

Table 1: The prediction strength of the properties in the different studies (the numbers mark the usefulness of the metrics where 1 means the best one)

Table 2.1 summarizes the result of the above mentioned experiments. In two cases the size metric was not examined (n.a.), but in all other cases, some correlations were established between the code metrics and the faults found. Number 1 in the table means that this property was the best one in the given study and the other numbers refer to further ranks.

	Fault proneness	Analysability	Changeability	Stability	Testability
Complexity	+	n.a.	-	-	-
Coupling	+	-	-	-	-
Size	+	-	-	-	-
Clones	n.a.	-	-	n.a.	n.a.
Unit test effort	-	+	n.a.	+	+

Table 2: Correlations between low level system properties and high level quality characteristics *(+ positive correlation, - negative correlation, n.a. no information available)*

2.2 Connections between low level properties and high level quality characteristics

The overall quality characteristics of a system can be derived from source code metrics like coupling and complexity. According to the quality model of ISO/IEC 9126-1 [13] there are three different views of the product's quality: *internal quality*, *external quality*, and *quality in use*. Internal quality means the measurable properties of the system through static code analysis. External quality is determined by the runtime behavior of the software as observed through dynamic analysis. The quality in use measures the quality of a system relative to the users' goals in a particular environment.

The ISO/IEC 9126-1 quality model distinguishes six software characteristics under external and internal quality (*functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, and *portability*). The purpose of this section is to investigate the relationship between sub-characteristics of *maintainability* and source code metrics. The definitions of the sub-characteristics are the following:

- **Analisability** is the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
- **Changeability** is a measure of how easy to modify the system. It can be measured in terms of the number of statements impacted by a change.
- **Stability** attributes refer to the unexpected side effects of the system modification.
- **Testability** is the ease with witch a modified software product can be to revalidated. It is measurable in terms of the number of test cases required to execute all modified statements with all relevant states relative to the number of modified statements [23].

Heitlager *et al.* [12] formulated a maintainability model that links high level maintainability characteristics to code level measures. They found the following influence between the examined source code properties and the maintainability characteristics. The *complexity* of source code has a negative correlation to the changeability and testability of the system. The greater the complexity of the source code is, the less the changeability and the testability are. The *size* of code units influences their analysability and testability. The greater the unit size is, the less the analysability and testability are. As might be expected the increased size metrics causes a deterioration of maintainability. The degree of *source code duplication* (also called *code cloning*) influences the analysability and the changeability. The degree of unit testing influences the analysability, stability, and testability of the system. The greater the *quality of unit testing* is, the better the analysability, stability, and testability of the system are. We use this model as the base of our examination, but the authors mention only the most influential causative links, so we had to add some other results from related works as it is illustrated in Table 2.1.

Elish *et al.* [7] investigated object-oriented design metrics proposed by Chidamber and Kemerer [5], and found correlations between these metrics and the logical *stability* of classes. They concluded from their experimental results that WMC, DIT, CBO, RFC, and LCOM metrics negatively correlate with the logical stability of classes. Nevertheless, since the Chidamber and Kemerer metrics are so widespread, we decided to use them to measure *code complexity* and *coupling*.

Chaumun *et al.* [4] observed the object-oriented metrics on three industrial sized test systems to find some influences on *changeability*. They showed that one type of CBO (CBO-IUB or *Fan In* is the number of foreign classes that reference a target class) is a good indicator of changeability, but not all of them were good because of the dismissed factors.

The linkage between system level characteristics and the impact of refactoring on a system can be found by these drafted contexts. In the next section we will explain how these factors are influenced by refactoring techniques.

3. EFFECT OF REFACTORINGS ON LOW LEVEL SYSTEM PROPERTIES

Having introduced selected refactoring techniques in detail, we now study their effect on the structure of the source code to see how the low level system properties are changed by them.

3.1 Refactoring techniques

Refactoring is a process of improving a software system's design [8]. It is a discipline that helps change the internal structure of the code without changing the external behaviour. These changes improve non-functional characteristics of the code like *analysability*, *changeability*, *stability*, and *testability*. Moreover refactoring helps to find bugs, because if you apply a technique in a difficult-to-understand code, you are better able to comprehend it. In this section we examine 14 refactoring techniques among the most popular ones.

Category	Refactoring	Complexity	Coupling	Size	Clones	Unit Test Effort
C1	Extract Method	-	+	-	0	+
	Replace Parameter With Explicit Method	-	+	-	0	+
	Replace Method with Method Object	-	+	-	0	+
	Replace Data Value with Object	-	+	-	0	+
	Extract Class	-	+	-	0	+
	Replace Type Code With Class	-	+	-	0	+
C2	Inline Method	+	-	+	0	-
	Inline Class	+	-	+	0	-
C3	Extract Superclass	-	0	-	-	-
	Pull Up Method	-	0	-	-	-
C4	Encapsulate Field	+	0	+	0	+
C5	Collapse Hierarchy	+	-	+	0	0
C6	Move Method	0	-	0	0	0
	Move Field	0	-	0	0	0

Table 3: The effects of refactorings on low level system properties

Extract Method extracts selected source code from the code block of an existing member. The new extracted method contains the replaced code from the original method. In the original method the code is replaced by a call to the new method.

Replace Method with Method Object extracts a part of a long method and makes the code more transparent. The number of local variables is also reduced for each class.

Replace Parameter with Explicit Method creates new methods from a method that runs different code depending on the values of an enumerated parameter. The new methods come from values of the enumerated parameters.

Replace Data Value with Object creates a new object which represents the data item from the original class. If you have two or more data items and functions, duplications may appear, the best thing you can do is to turn the data value into an object.

Extract Class creates a new class and moves the relevant attributes and methods from the old class into the new class. For new class, you feel it unnecessary to separate this class, but as soon as the responsibility grows, the code becomes too complicated.

Replace Type Code with Class replaces numeric type code with a new class. If you have a class with a numeric type code it does not affect its behaviour. Replace the number with a new class.

Inline Method puts a method's body into the caller's body and removes the method, if the method's body is just as clear as its name. Use short methods named to show their intention, because these methods lead to clearer and easier reading of the code.

Pull Up Method moves method which have the same body from the child class to the parent. This method helps to eliminate duplicate behaviour. Although two duplicate methods can work fine but sometimes in the future it may lead to bugs.

Extract Superclass creates a new class from two classes that do similar things in the same way or similar things in different ways. This method reduces code duplication.

Encapsulate Field enables you to create quickly a property from an existing field, and then update your code with references to the new property.

Inline Class abolishes a class which is no longer pulling its weight. When a class is not doing very much, move all its features into another class and delete it. Inline Class is the reverse of Extract Class.

Collapse Hierarchy merges similar classes in the inheritance hierarchy. The role of the collapsed class is taken over by the superclass and subclasses.

Move Field moves a field from one class to another. The previous field is deleted from the original class.

Move Method creates a new method with the same body in the mostly used class. The old method could be turned to a simple delegation.

3.2 Determining the connections between the refactoring techniques and low level system properties

We examine here the impact of the selected refactorings for each source code metric. After an overview of the existing literature, we objectively evaluate the effect of refactoring on metrics. We found that the effects of several refactoring techniques were the same on low level system properties, therefore these transformations were classified into six different categories. We examine these categories instead of the individual transformations by studying how the maximum values have changed at the given level. We consider the possible effects of each refactoring. The results are presented in Table 3.

Since the empirical studies investigates the maximum values of a given metrics instead of the total values, we have to take this into account. So if the maximum values of *structural*

complexity, *coupling*, and *size* metrics decrease during the transformation, we treat this change as a decrease of these values. The increase is defined similarly. We will put a + or − sign into the appropriate field of the table. If there is no effect between the refactoring and the given low level system property, a 0 is shown in the table.

It would be too long to explain all connections in detail, therefore we show only two examples. First, we can present obvious impact on clones only with *Pull Up Method* and *Extract Superclass* (Category 3). In these examples the main objective is to eliminate duplicate behaviour, therefore it decreases the number of clones, so a − sign was written into these places.

Harman *et al.* [11] chose *Move Method* refactoring and used a hill climbing technique to decrease the system level CBO value of the XOM system. They demonstrated in a practical example that applying *Move Method* might decrease the coupling which supports the − connection between them (see Table 3).

Finally, we examine the impact of the selected refactoring techniques on unit test effort. When we have to create new test cases as a result of a technique, the test effort will be increased. In this case, a + sign was put into the unit test effort column. In some cases unit test effort might be decreased (for example, a method is removed), so a − sign is used.

4. REFACTORINGS EFFECTS ON MAINTAINABILITY AND FAULT PRONENESS

In this section we describe the possible effects of refactoring on *maintainability* and *fault proneness*. Table 4 shows the relationships among our *refactoring categories* and system based maintainability and system based fault proneness (see Table 2.1 and Table 3). These tables show which category of refactoring can imply a shift in the spectrum of impact on maintainability. Rather than describing the potential impact of the associated characteristics, the cells indicate an improvement (+), a deterioration (-) or a neutral impact (0). If the relationship between a low level system property and a high level quality characteristic is unknown, we cannot determine the effect of the examined refactoring on the characteristic (a . is used for this case).

In Section 2 we overviewed the effect of the metrics, code duplications, and unit test effort on fault proneness and high level quality characteristics (see Table 2.1). Next we investigated the possible impacts of refactoring on the metrics (e.g. structural complexity), code duplications, and unit test effort (see Table 3). Now, we would like to merge these results to see the possible effects of refactorings on maintainability and fault proneness. Before introducing the process in general, we present a simple example for better understanding of the essence of this idea. One of the simplest refactoring is the *Extract Method* (EM) from the first category (C1). The effect of EM on fault proneness is based on many factors. The first one is that EM decreases structural complexity, which correlates positively with fault proneness, therefore EM decreases the fault proneness caused by the large complexity. Since coupling is increased by EM, and coupling correlates positively with fault proneness, the fault proneness based on coupling is increased. Similarly to complexity, the size based fault proneness is also decreased by EM. Since there is no connection between the *code duplications* and the fault proneness, this factor is not taken into account at all now. Finally, after carrying out EM refactoring more test effort is necessary (see Table 3) and the unit test effort has a negative effect on faults (see Table 2.1). Since the effort is usually fixed, the fault proneness is increased due to the growth of the test effort. These results (− + − . +) are summarized in the first cell of Table 4. Unfortunately, this result has two positive and two negative changes, so we cannot decide whether it is a positive or negative transformation overall. Hence we propose a weighting approach based on particular situations in the next section.

4.1 Discussion about the possible effects of refactoring techniques

Six categories were set up in Section 3 based on the effects of the examined refactoring actions (see Table 3). In the following text, each category will be discussed and the main advantages and disadvantages will be emphasized. Since we did not set up any order among the examined properties, they are treated equally according to their importance and we can take sides only in those cases, where the positive or negative effect has definite majority. In all other cases, we simply present the different advantages and disadvantages of each refactoring technique, leaving it to the reader to decide whether it is worth applying it or not.

A possible scenario for this could be as follows:

1. In the first step, the user selects the category of the desired refactoring.

2. Next, the weights for low level system properties are determined based on the particular situation.

3. Finally, based on the weights and the details from Table 4 the consequences on the high level quality characteristics can be estimated.

The **first category** (C1) is the reverse of the **second category** (C2). For example, *Extract Method or Class* can be found in C1, and their reverses, *Inline Method or Class*, are in C2. The transformation of C1 can take the source code more *changeable* (see Table 4) while C2 can transforms the code less *changeable*. All the other sections in these two categories contain two positive and two negative changes so its effect cannot be judged unambiguously, and the circumstances determine whether or not we should apply the given refactoring.

Refactoring techniques of the **third category** (C3) have unambiguous results. They decrease the probability of faults and increase all other examined aspects, so as to increase the *maintainability*. This means that any refactoring of this type is recommended for usage from all different points of view. *PullUp Method* is a good example of this category. Moving methods to the superclass can improve the maintainability of the system because it can decrease the *structural complexity* and the *size* of the code, and at the same time it eliminates the *clones*. Furthermore, it decreases the required *unit test effort*.

Category	Fault					Analysability					Changeability					Stability					Testability				
	C	CP	S	D	U	C	CP	S	D	U	C	CP	S	D	U	C	CP	S	D	U	C	CP	S	D	U
Category 1	−	+	−	.	+	.	−	+	+	−	+	−	+	+	.	+	−	+	.	−	+	−	+	.	−
Category 2	+	−	+	.	−	.	+	−	−	+	−	+	−	−	.	−	+	−	.	+	−	+	−	.	+
Category 3	−	0	−	.	−	.	0	+	+	+	+	0	+	+	.	+	0	+	.	+	+	0	+	.	+
Category 4	+	0	+	.	+	.	0	−	0	−	−	0	−	0	.	−	0	−	.	−	−	0	−	.	−
Category 5	+	−	+	.	0	.	+	−	0	0	−	+	−	0	.	−	+	−	.	0	−	+	−	.	0
Category 6	0	−	0	.	0	.	+	0	0	0	0	+	0	0	.	0	+	0	.	0	0	+	0	.	0

Table 4: Refactorings effects on high level quality characteristics (C: Complexity, CP: Coupling, S: Size, D: Code Duplications, U: Unit Test Quality)

The *Encapsulate Field*, which is the only element of **category four** (C4), is an often recommended technique. Hiding the data and adding accessors transformation can improve the object-oriented quality of the system but the new methods require new test cases and increase the *structural complexity* and the *size* of the class. Due to this change, the *maintainability* of the system is decreased and the *fault proneness* of the elements is increased. These results were surprising because according to our investigation this category does not have any positive effect on either maintainability or fault proneness.

The **fifth category** (C5) contains only the *Collapse Hierarchy*. This refactoring merge two classes into a new one, so *structural complexity* and *size* increase whereas *coupling* decreases. The number of the *clones* and the *unit test effort* do not change. The effect of this modification increases the *fault proneness* slightly, and it slightly decreases the *changeability*, the *stability*, and the *testability*.

According to Martin Fowler moving methods is the bread and butter of refactoring [8]. The *Move Method* refactoring is the general example of the **sixth category** (C6). The *general degree of coupling* can be decreased and the *maintainability* can be increased by using this technique. It is possible that this technique also decreases the *fault proneness* of the system, but there is unfortunately no proof of that.

All these outcomes are presented in Table 4.

5. CONCLUSIONS AND FUTURE WORK

Applying code refactoring is undoubtedly a plausible activity in maintenance, as it improves one or more quality attributes of the software system, if it is applied correctly at an appropriate time. However, the emphasis is on the latter condition; namely, each refactoring step (as any other change to a system) could have unwanted side-effects, which may lead to degradation of some other quality attributes. This fact is, unfortunately, often neglected by the literature and software professionals dealing with refactoring, since a detailed and widely accepted list of consequences applying the individual refactoring is still missing.

Software quality is difficult to define. The ISO/IEC 9216 standard establishes several categories, of which maintainability is the one we are interested in in this work, and it has a sub-characteristic testability which is of prime importance for us. Concrete measurable entities, metrics, are usually used to express high level quality characteristics, and similarly there is an accepted view on the relationship between different metrics and quality attributes. For instance, high coupling and structural complexity are treated as predictors of low testability and high error proneness. On the other hand, some other metrics are difficult to associate with high or low testability, like the amount of code clones present in the system.

If we combined these two mappings we could obtain a picture on the effect of different refactorings on higher level quality attributes, but as our research suggests, there is still a lot of research to be done in the field. One of the concrete goals would be to establish a list of objective consequences when applying the different refactoring techniques in terms of higher level quality attributes. A long term objective is that, on the one hand, 1) the quality attributes – like those proposed by the ISO/IEC 9126 standard – can be objectively estimated using metrics and other statically measurable data, and on the other hand, 2) the targeted quality attributes could be attained by applying software evolution activities (like refactoring) in a methodological way. Our proposal about refactoring and their consequences presented in this work is a contribution to this goal.

We plan to continue this research by performing more concrete case studies to verify the findings about the relationship of refactoring and quality attributes. Also, a more thorough analysis of the presented properties is required in order to be applicable as part of a real methodology. We plan to refine our connections by changing + and − into a more detailed scale for example an ordinal scale with numeric rating.

Acknowledgements

This research was supported by the Hungarian national grants OTKA K-73688, TECH 08-A2/2-2008-0089, and GOP-1.1.2-07/1-2008-0007.

6. REFERENCES

[1] J. Bansiya and C. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. In *IEEE Transactions on Software Engineering*, volume 28, pages 4–17. IEEE Computer Society, Jan. 2002.

[2] V. R. Basili, L. C. Briand, and W. L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. In *IEEE Transactions on Software Engineering*, volume 22, pages 751–761, Oct. 1996.

[3] M. Bruntink and A. van Deursen. An empirical study into class testability. *The Journal of Systems & Software*, 79(9):1219–1232, 2006.

[4] M. A. Chaumun, H. Kabaili, R. K. Keller, F. Lustman, and G. Saint-Denis. Design properties and object-oriented software changeability. *Software Maintenance and Reengineering, European Conference*

on, page 45, 2000.

[5] S. Chidamber and C. Kemerer. A Metrics Suite for Object-Oriented Design. In *IEEE Transactions on Software Engineering 20,6(1994)*, pages 476–493, 1994.

[6] F. B. e Abreu and W. Melo. Evaluating the Impact of Object-Oriented Design on Software Quality. In *Proceedings of the Third International Software Metrics Symposium*, pages 90–99. IEEE Computer Society, Mar. 1996.

[7] M. Elish and D. Rine. Investigation of metrics for object oriented design logical stability. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 26–28. IEEE Computer Society, 2003.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Pub Co, 1999.

[9] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. In *IEEE Transactions on Software Engineering*, volume 31, pages 897–910. IEEE Computer Society, Oct. 2005.

[10] M. Harman, A. Baresel, D. Binkley, R. Hierons, L. Hu, B. Korel, P. McMinn, and M. Roper. *Formal Methods and Testing, Testability Transformation–Program Transformation to Improve Testability*. Springer Berlin / Heidelberg, 2008.

[11] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1106–1113, New York, NY, USA, 2007. ACM.

[12] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *QUATIC*, pages 30–39, 2007.

[13] International Standards Organization. *Software engineering - product quality - part 1: Quality model*, ISO/IEC 9126-1 edition, 2001.

[14] S. Jungmayr. Reviewing software artifacts for testability. *EuroSTAR*, 99:8–12, 1999.

[15] The Mozilla Homepage. `http://www.mozilla.org`.

[16] M. O'Keeffe and M. Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(5):345–364, 2008.

[17] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. In *IEEE Transactions on Software Engineering*, volume 33, pages 402–419, June 2007.

[18] J. Pipka. Refactoring in a "Test First"-World. In *Proceedings of the Third International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, 2002.

[19] Rhino Home Page. `http://www.mozilla.org/rhino`.

[20] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916, New York, NY, USA, 2006. ACM.

[21] I. Siket. Evaluating the Effectiveness of Object-Oriented Metrics for Bug Prediction. *Periodica Polytechnica*, Budapest, Accepted for publication.

[22] H. Sneed. Reengineering for testability. In *Proceedings des Workshops zum Software Engineering, Seite 31ff, Bad Honnef*. Citeseer, 2006.

[23] H. M. Sneed and S. Jungmayr. Produkt- und prozessmetriken für den softwaretest. *Informatik Spektrum*, 29(1):23–38, 2006.

[24] R. Subramanyan and M. S. Krishnan. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. In *IEEE Transactions on Software Engineering*, volume 29, pages 297–310, Apr. 2003.

[25] P. Yu, T. Systä, and H. Müller. Predicting Fault-Proneness using OO Metrics: An Industrial Case Study. In *Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 99–107, Mar. 2002.