The Impact of Refactoring on Maintability of Java Code: A Preliminary Review

MITJA GRADIŠNIK, SAŠO KARAKATIČ, TINA BERANIČ and MARJAN HERIČKO, University of Maribor
GORAN MAUŠA, University of Rijeka
TIHANA GALINAC GRBAC, Juraj Dobrila University of Pula

The preservation of a proper level of software systems quality is one in the cornerstones of making software evolution easier and sustainable in the long run. A good design allows complex systems to evolve with little effort and in an economically efficient way. When design deviations are detected, refactoring techniques are applied to eliminate or at least reduce the identified flaws. A number of studies show that not all refactoring techniques contribute to improving the quality of different software systems equally. Therefore, effective approaches to measuring the impact of refactoring on software quality are required. In this study, we examine approaches to estimate the effect of applied refactoring techniques on the maintainability of Java based software systems. Since refactoring primarily affects the system's internal structure, maintainability was put in the focus of the study. We conducted a brief literature review, limiting our study on quantitative metrics. The results show that researchers use different approaches for evaluating the impact of refactoring on the observed Java based software systems. In some studies, researchers measured the effect of refactoring on the internal structure attributes measured by software metrics, e.g. C&K metric suite but the scope of our research was limited to the effects of refactoring on maintainability. In other studies, the effects of refactoring are estimated through external quality attributes, e.g. maintainability, readability, and understandability. Additionally, some researchers observed the impact of refactoring indirectly, e.g. through the defect proneness of classes of observed systems.

1. INTRODUCTION

Software quality is one of the most important issues in software engineering, drawing attention from both practitioners and researchers [Alkharabsheh et al. 2018]. Insufficient quality leads to an unacceptable product. Likewise, as the level of quality increases, the product becomes more useful, and the highest levels of quality can give the product a competitive advantage. Although, higher product quality decreases software evolution and maintenance costs, beyond a certain point the level of quality becomes excessive [Barney et al. 2012]. Thus, increasing the quality level of a software system even further does not bring any additional competitive advantage. This is why, software quality should maintain a balance between development velocity, efficiency and resources required to build a useful software products that satisfies the expectations of stakeholders. As software quality is a complex and multifaceted concept, user satisfaction is not its only aspect [Barney et al. 2012]. Software quality also

The authors acknowledge the project (An empirical comparison of machine learning based approaches for code smell detection, BI-HR/18-19-036) which was supported financially by the Slovenian Research Agency and by Croatian Ministry of Science and Education.
Author's address: M. Gradišnik, S. Karakatič, T. Beranič, M. Heričko, Koroška cesta 46, 2000 Maribor, Slovenia; email: {mitja.gradisnik, saso.karakatic, tina.beranic, marjan.hericko}@um.si; G. Mauša, Vukovarska 58, 51000 Rijeka, Croatia; email: goran.mausa@riteh.hr; T. Galinac Grbac, Zagrebačka 30, 52100 Pula, Croatia; email: tihana.galinac.grbac@unipu.hr.

In: Z. Budimac and B. Koteska (eds.): Proceedings of the SQAMIA 2019: 8th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Ohrid, North Macedonia, 22–25. September 2019. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

includes non-functional attributes, such as reliability and maintainability [Gorla and Lin 2010]. Improving maintainability is one of the cornerstones of making software evolution easier [Alkharabsheh et al. 2018].

The design of a software system is one of the most influential factors in its quality, and a good design allows the system to evolve with little effort and less money [Stroggylos and Spinellis 2007]. Code smells have become an established concept for patterns or aspects of software design that may cause problems to further development and maintenance of the system [Yamashita and Leon 2013]. The term code smells was first coined by [Riel 1996] and [Brown et al. 1998] to refer to any symptom for poorly designed parts of code that can cause serious problems while maintaining the software [Lafi et al. 2019]. [Fowler 1999] provided subsequently a set of informal descriptions for twenty-two code smells. Each code smell in the catalog is associated with a corresponding refactoring technique that can be applied to remedy it. The main motivation for using code smells for system maintainability assessment is that they constitute software features that are potentially easier to interpret than traditional object-oriented software measures [Yamashita and Counsell 2013]. Refactoring is a valuable tool that can be used to improve the design of software and consequently reduce thenegative effects of software quality degradation. Our definition of refactoring comprises only those changes in the software system's program code that change its structure but do not add any additional functionalities. Although there is a general belief among software developers that refactoring always leads to improved software quality, research shows all refactoring techniques do not necessarily improve software quality nor do they improve all aspects of software quality equally. Often, the improvement of one aspect of software quality leads to deterioration in another aspect of software quality. Since we do not have a clear and quantitative instrument for measuring software quality it is often hard to evaluate the benefits of such quality improvement activities.

Therefore, the general research field looking at the impact of refactoring due to code smells on software quality is not yet entirely clear. The goal of this paper is to study which approaches to the quantitative measurement of changes in software quality have been used in recent studies by researchers who are studing the effect of refactoring techniques on software quality attributes. Since refactoring primarily affects the maintainability, the study was focused on maintainability and its related external attributes, such as reusability, analyzability, and modifiability. To answer our research goal, we performed a brief literature review. We only considered studies published within the last five years.

The main objective of this paper is to review measurement approaches used to evaluate improvements in software design achieved by refactoring techniques. Hence, we conducted a brief literature review of the last five years and extracted approaches for evaluating the change in maintainability of two observed software versions.

This paper is structured as follows. The introduction establishes the scope and purpose of the paper and gives the necessary background information relevant to the research. Following the introduction, a second section explains the procedure of the literature review. In sections 3 and 4, refactoring and maintainability are defined using definitions provided by the authors of the analyzed studies. Section 5 outlines the key approaches in measuring the effects of refactoring on maintainability. Finally, we summarize our research and give concluding remarks in the last section.

2. METHOD

The main research goal of this paper is to explore *the measures used to quantify the refactoring impact*. In addition, we looked at *how refactoring is defined* in the literature and searched for the *reason that guided the refactoring activities*. Since we focused our study on maintainability, our aim was to detect *different measures related to maintainability* and to summarize all the *tools used for measuring these measures*. To answer these questions and gather available research contributions, a preliminary review

Table I. Papers on measuring the impact of refactoring on software maintainability from 2015 to 2019.

Research Digital Library	URL	No. of Results	No. of Selected Studies (based on abstract and title)	No. of Relevant Studies
IEEE Xplore Digital Library	ieeexplore.ieee.org	38	12	9
ScienceDirect	sciencedirect.com	22	8	3
ACM Digital Library	dl.acm.org	28	12	8
arXiv (all repositories)	arxiv.org	20	4	2
Google Scholar [A,B]	scholar.google.com	50	3	1

[A] "*-Elsevier -IEEE*" was added to the search query, to prevent the duplication of results.
[B] Only 50 of the most relevant papers were taken into account.

was done using five digital libraries and research search engines: IEEE Xplore, ScienceDirect, ACM Digital Library, arXiv, and Google Scholar. The selection of the relevant papers is based on the title and the abstract relevance. We only considered papers published between 2015 and 2019. The search was done using the following search query: *refactoring AND maintainability AND software*. The results obtained from digital libraries are shown in Table I. In our research, we only included studies available in the selected digital libraries and written in the English language. The search was conducted for both journal and conference papers and all of the studies used in our paper were published in the software engineering domain. In total, more than 158 (and many more from Google Scholar, which were not relevant) results were obtained from five research digital libraries. Based on the title and the abstract 39 papers were read and the final set of 27 papers was formed which served as the basis for this research. There is a rising trend in the number of published relevant papers, whereas the conferences still constitute the most popular medium for publishing papers on the topic.

3. REFACTORING

The definition of refactoring as an activity of software quality assurance is unanimous among researchers, mainly taken from [Fowler et al. 1999]. They all agree that refactoring is basically the restructuring of source code used to improve existing code [Fontana et al. 2015], i.e. the internal structure of software systems [Kádár et al. 2016; Rathee and Chhabra 2017], thereby improving its understandability and readability [Tarwani and Chug 2016] while preserving their external behavior [Fontana et al. 2015; Malhotra et al. 2015; Ouni et al. 2016; Vidal et al. 2018; Szőke et al. 2017; Kannangara and Wijayanayake 2013; Hegedűs et al. 2018; Gatrell and Counsell 2015; Bashir et al. 2017], or more generally, to reduce technical debt [Kouros et al. 2019]. Another view on refactoring, from the developers' perspective, was given by [Szőke et al. 2017]. They rely on findings by [Kim et al. 2012], supported by their own research [Szőke et al. 2014], that developers' use refactoring primarily to fix coding issues and not for the refactoring of code smells or antipaterns. A different definition was given by [Kaur and Singh 2017] where the authors described refactoring as an approach that decreases the complexity of software by fixing errors or appending new features. For [Mens and Tourwé 2004] the aim of refactoring, adopted by [Mehta et al. 2018], is to redistribute classes, variables, and methods across a class hierarchy in order to facilitate future adaptations and extensions. Refactoring activities may include operations like [Ouni et al. 2017; Szőke et al. 2017; Steidl and Deissenboeck 2015]: (1) class - level: move, inline, rename, extract class, subclass, superclass or interface, (2) method - level: move, pull up, push down, extract, rename, parameter change and (3) field - level: move, pull up, push down.

On the other hand, a study done by [Shatnawi and Li 2011] identified a different subset of ten refactoring techniques with the highest impact: Introduce Local Extension, Duplicate Observed Data, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Conditional with Polymorphism, Introduce Null Object, Extract Subclass, Extract Interface, Form Template Method, and Push Down Method.

Code Smells are thought to be the primary technique for identifying refactoring opportunities [Fontana et al. 2015; Kádár et al. 2016; Rathee and Chhabra 2017] and refactoring depends solely on our ability to identify them [Rathee and Chhabra 2017], which may be done by tools based on predefined metrics [Steidl and Deissenboeck 2015]. Code smells are defined as symptoms of problems at the code or design level [Fontana et al. 2015], specific types of design flaws [Kouros et al. 2019] or certain structures which violate the design principles [Malhotra et al. 2015] that originated from poor design choices applied by programmers during the development of a software project [Palomba et al. 2018]. Although code smells do not always represent direct problems to a software system (faults or defects) [Fontana et al. 2015], they are known to degrade quality, and impact the legibility, maintainability, and evolution of the system [Vidal et al. 2018], thus motivateing developers to remove them through refactoring.

The identification of refactoring may be done by self-reported refactorings by programmers [Nayebi et al. 2018] or by looking into manual refactoring, but the support of a static source code analyzer tool like SourceMeter is found to be helpful for developers [Szőke et al. 2017]. However, the dominant approach is to use different tools to detect the refactoring. [Hegedűs et al. 2018] searched for refactorings done in seven open-source Java systems with RefFinder [Kim et al. 2010], a tool for refactoring extraction. The same tool was also used by [Kaur and Singh 2017] to extract refactoring tasks from the source code of four versions of the open-source Junit project. On the other hand [Gatrell and Counsell 2015] used the automated tool Bespoke to detect the occurrence of 15 types of refactoring in C# programming language.

4. MEASUREMENT OF MAINTAINABILITY

Within the performed review, we focused on software maintainability since it represents an important aspect within source code refactoring. Maintainability can be understood differently. For example, maintainability is one of the characteristics defined in the ISO 25010 [ISO/IEC 25010 2011] that presents a software product quality model, ”the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in the environment, and in the requirements”. This is not the only definition used in the literature. The definition of maintainability can also be found within ColumbusQM quality model [Bakota et al. 2011] and others individual definitions are frequently found. Additionally, in the software product quality model, maintainability is composed of more subcharacteristics defined and measured in a prescribed way. To unify the understanding of maintainability, the definition of the investigated attribute and possible subattributes within each work is crucial.

[Tarwani and Chug 2016] define software maintainability as the ease, with which software can be modified, corrected or updated. To measure software maintainability, the authors relied on the relation between metrics and software maintainability by using the definition proposed by [Dubey and Rana 2011], which states that the relation between metrics and software maintainability is always inversed. [Malhotra and Chug 2016] adopt the term maintainability by [Aggarwal et al. 2006], who define it as the ease with which software can be modified. It is measured in terms of Line of Code (LOC) added, deleted and modified in the maintenance phase of SDLC. [Bashir et al. 2017] define software maintainability as the level of ease in extension, fixing bugs and preforming certain maintenance-related activities. [Szőke et al. 2017] and [Hegedűs et al. 2018] adopt the definition of maintainability provided in the ColumbusQM probabilistic quality model [Bakota et al. 2011], which is based on ISO/IEC 25010 quality characteristics and reduces software quality to one single value.

Still, the majority of researchers [Steidl and Deissenboeck 2015; Ouni et al. 2017; Nayebi et al. 2018; Kannangara and Wijayanayake 2013; Kádár et al. 2016; Ouni et al. 2016; Vidal et al. 2018], did not provide the definition of the maintainability or targeted subcharacteristic. A special case was the paper by [Gatrell and Counsell 2015], since the main focus of their paper was not in measuring the maintainability but following the change and fault proneness of classes. In the paper by [Kaur and Singh 2017] the explicit definition of software maintainability is not provided, but they refer to software metrics as a measure of software maintainability of observed software projects.

Only a small number of papers look into the subcharacteristics of maintainability. For example, [Kannangara and Wijayanayake 2013] measured the analysability, changeability, resource utilization and time behavior for each participant in their experiment doing the refactoring. They also measured the maintainability index, cyclomatic complexity, depth of inheritance, class coupling, and lines of code for refactored code and code without refactoring. [Nayebi et al. 2018] used two metrics to measure software quality, especially the maintainability. First, the decoupling level by Mo et al. [Mo et al. 2016], which says how well a software system is decoupled into independent modules, using Baldwin and Clark's design rule theory as the underlying theoretical foundation: the more active, independent, and small modules there are, the more option values can be produced. They also used the propagation cost proposed by [MacCormack et al. 2006] to measure how tightly coupled a system is, based on the dependencies among files. They also identified the following design flaws and architecture smells in accordance with [Mo et al. 2015]: package cycles, improper inheritance, modularity violations, crossings, and unstable interfaces. They combined this with the DRSpace tool [Xiao et al. 2014] which identifies most error-prone files.

5. EVALUATING THE CHANGE IN MAINTAINABILITY

Researchers agree that once refactoring is applied, the developer should assess its effects in terms of software complexity, understandability, and maintainability or in terms of productivity, cost, and effort for the undertaken process [Ouni et al. 2016; Vidal et al. 2018]. This could be done, for example, by looking at different software metrics to measure the attributes of coupling, complexity, cohesion, or other aspects of the system, before and after refactoring [Vidal et al. 2018]. Some of the established software metrics used later in the paper with abbreviations are summed up in Table II.

5.1 Assessment of the internal design

In some studies, the assessment of the change in maintainability relies on the software's internal design. For instance, to assess a change in maintainability of software products, researchers [Chug and Tarwani 2017] measured nine software metrics, namely CBO, LCOM, RFC, WMC, NOC, DIT, Ocavg, AHF, and MHF. According to the authors' understanding of maintainability adopted by [Dubey and Rana 2011], maintainability is in inverse relation to the values of the measured software metrics, i.e. an increase in value of the metrics results in a decrease in the maintainability value. An identical set of metrics was also used by researchers [Tarwani and Chug 2016], who acquired the metric measurements using the IntelliJ IDEA metrics plug-in. Furthermore, a similar approach was taken by the researchers [Malhotra and Chug 2016]. The researchers refer to the work of [Singh and Malhotra 2012] which states that the metrics of the C&K suite are negatively correlated with maintainability. In the study, the researchers empirically evaluated the repercussions of refactoring on maintainability with the help of their measurable effects on the internal quality attributes as well as the external quality attributes. While internal attributes were assessed using the C&K metric suite, the external attributes of observed software projects were assessed through expert opinion. The researchers [Kaur and Singh 2017] estimated the internal structure of software products by measuring its complexity. According to researchers, complexity can be efficiently estimated via an assessment of size (TLLOC, and TNOS), coupling (RFC, and NOI), clone (CI), complexity (WMC) and comment (TCLOC) of software products. To measure the required code metrics for the complexity estimation, the authors used the Halstead Metrics tool. Furthermore, [Ouni et al. 2017] measured the number of code smells, number of design patterns and the hierarchical model of evaluating software with 11 low level metrics by

[Bansiya and Davis 2002]: design size in classes, number of hierarchies, average number of ancestors, data access metric, direct class coupling, cohesion among methods of class, measure of aggregation, measure of functional abstraction, number of polymorphic methods, class interface size, and number of methods. The authors of the study did not provide the information on which tool was used to measure the metrics.

Table II. List of metrics and their descriptions.

Metric name	Metric description	Metric name	Metric description
AHF	Attribute Hiding Factor	NC	Number of Classes
CBO	Coupling Between Objects	NGen	Number of Generalizations
CI	Clone Index	NGenH	Number of Generalization Hierarchies
Connectivity	Connectivity	NOC	Number of Children
DIT	Depth of Inheritance Tree	NOI	Number of Incoming Invocations
EPM	Entity Placement metric	OCavg	Cyclomatic Complexity of a Class
LCOM	Lack of cohesion of methods	RFC	Response For Class
maxDIT	Maximum DIT	TCLOC	Total Comment Lines of Code
MHF	Method Hiding Factor	TLLOC	Total Logical Lines of Code
MI	Maintainability Index	TNOS	Total Number of Statements
MPC	Message Passing Coupling	WMC	Weighted Methods for Class
NAggH	Number of Aggregation Hierarchies		

[Kannangara and Wijayanayake 2013] also focused on the internal attributes of studied software products while estimating a change in software maintainability after refactoring the project's code. The authors of the study did an experiment with students of computer science, where they split the students into two groups: the control group with C# without the refactoring and the experimental group of students which got the refactored C# code. They measured the students' performance on fixing bugs and answering questions about the code. An analysis of the internal metrics showed that there was improvement in the maintainability index, while other metrics (cyclomatic complexity, Dept of Inheritance, Class Coupling and Lines of Code) stayed the same.

In their study, researchers [Steidl and Deissenboeck 2015] used a simplified model to estimate the change in the software quality of studied software by measuring the change in code size. The researchers did use a change in lines of code of observed method as a metric (measured by their own tools), besides their own ıgrowth quotient, which showed how methods got bigger. According to the quality model used in the research, longer methods mean less maintainable program code.

5.2 Assessment of external software attributes

In contrast to the approaches that estimate a change in software maintainability through a change of values of internal attributes, in some studies the estimation of external attributes of software products is applied. [Kouros et al. 2019] argued that metrics are not reliable indicators of software quality when comparing different products or even different versions of the same system and that quality can be objectively assessed only by measures that evaluate a design against the optimum design that could be achieved for a particular context. Hence, they used the Entity Placement metric from [Tsantalis and Chatzigeorgiou 2009] which encompases interclass coupling in the numerator and intra-class cohesion in the denominator, making it a suitable fitness function for their search-based approach to propose activities like refactoring to achieve an optimal architecture and thus reduce technical debt. In the literature, it is common to assess the maintainability of a software system by calculating its maintainability index (MI). One of the works that focuses on the maintainability index, is a study performed by [Mehta et al. 2018]. The authors of the study proposed an approach to improving software quality by removing relevant code smells from the source code of observed software systems. The effectiveness of the proposed approach is demonstrated by measuring the Maintainability Index and Relative logical complexity, first measured by the JHawk tool and subsequently measured by the Eclipse Metrics plug-in. To reduce maintainability measures to a single assessment value the authors introduced the Maintainability Complexity Index (MCI), calculated according to the formula $MCI = MI * RLC$. According to the authors, the combination of MI and RLC does better at estimating the maintainability of a software system than the maintainability index itself.

[Bashir et al. 2017] adopted the MOMOOD quality model, proposed by [W. A. Rizvi and Khan 2010]. The model defines maintainability through the following formula: $Maintainability = -0.126 + 0.645 * understandability + 0.502 * Modifiability$, where understandability is calculated via the formula: $Understandability = 1.166 + 0.256 * NC – 0.394 * NGenH$, and modifiability is calculated via the formula: $Modifiability = 0.629 + 0.471 * NC – 0.173 * NGen - 0.616 * NaggH – 0.696 * NGenH + 0.396 – MaxDIT$. The authors do not state how the metrics required for maintainability assessment were measured. Furthermore, [Szőke et al. 2017] measured the effect of refactoring on the software projects by the Columbus QM probabilistic software maintainability model [Bakota et al. 2011] which is based on the quality characteristics defined by the ISO/IEC 25010 standard. The maintainability of the software project was measured by SourceMeter, a tool developed by the authors of the study. The same tool, QualityGate SourceAudit, was used by [Hegedűs et al. 2018], within which the Relative Maintainability Index (RMI) is measured. RMI expresses the maintainability of a code element and is by calculated using dynamic thresholds from a benchmark database istead of fixed formulas [Hegedűs et al. 2018]. In a similar study, [Szőke et al. 2017] identified refactoring commits based on the tickets and analyzed the maintainability of the revision before and after the commit. The measurement was done with QualityGate SourceAudit [1]. The effect of refactoring is measured as: $MaintainabilityChange = Maintainability(t(i)) – Maintainability(t(i) - 1)$. [Hegedűs et al. 2018] did not deal with maintainability change itself, but researched the differences in relative maintainability index between refactored and non-refactored elements. [Kádár et al. 2016] used a metric named the Relative Maintainability Indices of source code elements, calculated by the QualityGate, an implementation of the ColumbusQM quality model. Like the well-known maintainability index, the Relative Maintainability Indices reflects the maintainability of a software module, but is calculated using dynamic thresholds from a benchmark database and not via a fixed formula. Thus, it expresses the relative maintainability of a software module compared to the maintainability of other elements in the benchmark.

Similarly, [Han and Cha 2018] propose a two-phase assessment approach for refactoring identification based on the calculation of the delta value in maintainability. The authors took into consideration several aspects that can affect maintainability. To assess maintainability as accurately as possible the metric values of the Entity Placement Metric (EPM), Connectivity and message Passing Coupling (MPC) were calculated. There are also novel measures used for this purpose. For example, [Malhotra et al. 2015] defined the measure Quality Depreciation Index Rule (QDIR) that is calculated by considering both bad smells and the C&K Suite of Metric, while [Rathee and Chhabra 2017] focused on improving the cohesion of different classes of object-oriented software using a newly proposed similarity metric based on frequent usage patterns. Ouni et al. [Ouni et al. 2017] measured the gain in different QMOOD quality factors (reusability, flexibility, understandability, effectiveness, functionality and extendibility) ratio changed as defined by [Bansiya and Davis 2002]. Also, they used Ph.D. students to evaluate the refactoring and compare the results of their approach to professional recommendations.

While most researchers evaluated maintainability via metric measurements, the researchers [Malhotra and Chug 2016] estimated understandability, level of abstraction, modifiability, extensibility, and reusability through expert opinions.

[1] https://www.quality-gate.com/

5.3 Indirect assessment of external quality attributes

Last but not least, in some studies researchers focused on the reliability of a software system in order to assess a change in the software quality of an observed software system after refactoring had been applied. A change in reliability can be detected by the increased frequency of defects that are rooted in poor software' design. However, the measure of quality is often expressed in terms of a number of defects, before and after applying refactoring [Fontana et al. 2015; Ouni et al. 2016]. For example, maintainability was not measured by [Gatrell and Counsell 2015], however, the researchers looked into the change and fault proneness of classes, which is a commonly used predictor of a system's reliability.

6. CONCLUSION

To maintain a proper level of software quality in the long run, the detection of deviations in a system's design should be responded by improvement actions, usually performed by refactoring the affected part of the software system. Despite the liveliness of refactoring research field in the past, there is no complete consensus about the definition of refactoring. The majority of researchers in the analyzed studies understand refactoring as a restructuring of the code used to improve existing code, i.e. the internal structure of software systems, its understandability and readability, while preserving their internal behavior. Hence, refactoring improves the quality of the internal structure of the software, without adding any extra functionalities. In general, refactoring can be understood as a set of corrective activities that contribute to their longevity by improving the internal structure of software systems.

In existing studies, maintainability represents a software quality aspect of the software that is mostly affected by source code refactoring. In general, maintainability can be best described as the ease, with which software can be modified, corrected and updated. Researchers do not agree completely on when the maintenance phase start. For most of them, maintenance is a phase that starts once the software is delivered to customers. Regardless of the interpretation of maintenance activity, maintainability represents an important aspect of the quality of the software overall. Consequently, it is also an important aspect of software quality models, i.e. ISO 25010 standard and ColumbusQM quality model. Despite the fact that maintainability is defined as a compound quality attribute in software quality models, only a small number of papers look into the sub-characteristics of maintainability, e.g. understandability, readability, and changeability.

The results of the study show that researchers use different approaches to evaluate the effects of refactoring on observed software systems. In some studies, researchers are focused on measuring the effect of refactoring on attributes of a system's internal structure. According to the studies, a change in the quality of the system's internal structure can be detected by a set of software metrics or metric suites. Often, the effect of refactoring on maintainability can also be estimated through the estimation of some external quality attributes, e.g. maintainability, readability, and understandability. Last but not least, some researchers observe the impact of refactoring indirectly, e.g. through the defect proneness of the classes in the observed software systems.

A study of the effects of the defected anomalies in software' design on software quality attributes has remained a lively field of research over the last decade. One of the main objectives of the research field is to objectively assess how improvements in software design achieved by refactoring techniques contribute to higher software quality. The goal of this study was to review the literature of the last five years and extract approaches for evaluating the changes in maintainability of two observed software versions. In the literature, maintainability was most commonly associated with software's internal design. Therefore, this quality attribute was the focus of our study.

REFERENCES

K Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. 2006. Empirical Study of Object-Oriented Metrics. *Journal* *of Object Technology* 5 (01 2006), 149–173. DOI:http://dx.doi.org/10.5381/jot.2006.5.8.a5

Khalid Alkharabsheh, Yania Crespo, Esperanza Manso, and José A. Taboada. 2018. Software Design Smell Detection: a systematic mapping study. *Software Quality Journal* (2018).

T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, and T. Gyimóthy. 2011. A probabilistic software quality model. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 243–252.

Jagdish Bansiya and Carl G. Davis. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on software engineering* 28, 1 (2002), 4–17.

Sebastian Barney, Kai Petersen, Mikael Svahnberg, Aybüke Aurum, and Hamish Barney. 2012. Software quality trade-offs: A systematic map. *Information and Software Technology* 54, 7 (2012), 651–662.

R. S. Bashir, S. P. Lee, C. C. Yung, K. A. Alam, and R. W. Ahmad. 2017. A Methodology for Impact Evaluation of Refactoring on External Quality Attributes of a Software Design. In *2017 International Conference on Frontiers of Information Technology (FIT)*. 183–188. DOI:http://dx.doi.org/10.1109/FIT.2017.00040

William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (1st ed.). John Wiley & Sons, Inc., New York, NY, USA.

A. Chug and S. Tarwani. 2017. Determination of optimum refactoring sequence using A* algorithm after prioritization of classes. In *2017 International Conference on Advances in Computing, Communications and Informatics*. 1624–1630. DOI:http://dx.doi.org/10.1109/ICACCI.2017.8126075

Sanjay Kumar Dubey and Ajay Rana. 2011. Assessment of Maintainability Metrics for Object-oriented Software System. *SIGSOFT Softw. Eng. Notes* 36, 5 (2011), 1–7. DOI:http://dx.doi.org/10.1145/2020976.2020983

Francesca Arcelli Fontana, Marco Mangiacavalli, Domenico Pochiero, and Marco Zanoni. 2015. On Experimenting Refactoring Tools to Remove Code Smells. In *Scientific Workshop Proceedings of the XP2015 (XP '15 workshops)*. ACM, Article 7, 8 pages. DOI:http://dx.doi.org/10.1145/2764979.2764986

Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

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

M. Gatrell and S. Counsell. 2015. The effect of refactoring on change and fault-proneness in commercial C# software. *Science of Computer Programming* 102 (2015), 44 – 56.

Narasimhaiah Gorla and Shang-Che Lin. 2010. Determinants of software quality: A survey of information systems project managers. *Information and Software Technology* 52, 6 (2010), 602–610.

A. Han and S. Cha. 2018. Two-Phase Assessment Approach to Improve the Efficiency of Refactoring Identification. *IEEE Transactions on Software Engineering* 44, 10 (2018), 1001–1023. DOI:http://dx.doi.org/10.1109/TSE.2017.2731853

Péter Hegedűs, István Kádár, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology* 95 (2018), 313 – 327.

ISO/IEC 25010. 2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. (2011).

István Kádár, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. 2016. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2016)*. ACM, New York, NY, USA, Article 10, 4 pages. DOI:http://dx.doi.org/10.1145/2972958.2972962

SH Kannangara and WMJI Wijayanayake. 2013. Measuring the Impact of Refactoring on Code Quality Improvement Using Internal Measures. In *Proc. of the International Conference on Business & Information*.

G. Kaur and B. Singh. 2017. Improving the quality of software by refactoring. In *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*. 185–191. DOI:http://dx.doi.org/10.1109/ICCONS.2017.8250707

Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *in Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, ser. FSE '10*. 371–372.

Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 50:1–50:11.

Panagiotis Kouros, Theodore Chaikalis, Elvira-Maria Arvanitou, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, and Theodoros Amanatidis. 2019. JCaliper: Search-based Technical Debt Management. In *Proceedings of the 34th Symposium on Applied Computing*. ACM, 1721–1730. DOI:http://dx.doi.org/10.1145/3297280.3297448

M. Lafi, J. W. Botros, H. Kafaween, A. B. Al-Dasoqi, and A. Al-Tamimi. 2019. Code Smells Analysis Mechanisms, Detection Issues, and Effect on Software Maintainability. In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. 663–666. DOI:http://dx.doi.org/10.1109/JEEIT.2019.8717457

Alan MacCormack, John Rusnak, and Carliss Y Baldwin. 2006. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science* 52, 7 (2006), 1015–1030.

R. Malhotra and A. Chug. 2016. An empirical study to assess the effects of refactoring on software maintainability. In *2016 International Conference on Advances in Computing, Communications and Informatics*. 110–117. DOI:http://dx.doi.org/10.1109/ICACCI.2016.7732033

Ruchika Malhotra, Anuradha Chug, and Priyanka Khosla. 2015. Prioritization of Classes for Refactoring: A Step Towards Improvement in Software Quality. In *Proceedings of the Third International Symposium on Women in Computing and Informatics*. ACM, 228–234. DOI:http://dx.doi.org/10.1145/2791405.2791463

Y. Mehta, P. Singh, and A. Sureka. 2018. Analyzing Code Smell Removal Sequences for Enhanced Software Maintainability. In *2018 Conference on Information and Communication Technology (CICT)*. 1–6. DOI:http://dx.doi.org/10.1109/INFOCOMTECH.2018.8722418

Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.* 30, 2 (2004), 126–139. DOI:http://dx.doi.org/10.1109/TSE.2004.1265817

Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 51–60.

Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. 2016. Decoupling level: a new metric for architectural maintenance complexity. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 499–510.

Maleknaz Nayebi, Yuanfang Cai, Rick Kazman, Guenther Ruhe, Qiong Feng, Chris Carlson, and Francis Chew. 2018. A Longitudinal Study of Identifying and Paying Down Architectural Debt. *arXiv preprint arXiv:1811.12904* (2018).

Ali Ouni, Marouane Kessentini, Mel Ó Cinnéide, Houari Sahraoui, Kalyanmoy Deb, and Katsuro Inoue. 2017. MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software* 29, 5 (2017), 18–43.

Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study. *ACM Trans. Softw. Eng. Methodol.* 25, 3, Article 23 (June 2016), 53 pages. DOI:http://dx.doi.org/10.1145/2932631

Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation. In *40th Proceedings of ICSE*. ACM, 482–482. DOI:http://dx.doi.org/10.1145/3180155.3182532

Amit Rathee and Jitender Kumar Chhabra. 2017. Restructuring of Object-Oriented Software Through Cohesion Improvement Using Frequent Usage Patterns. *SIGSOFT Softw. Eng. Notes* 42, 3 (Sept. 2017), 1–8. DOI:http://dx.doi.org/10.1145/3127360.3127370

Arthur J. Riel. 1996. *Object-Oriented Design Heuristics*. Addison-Wesley Professional.

Raed Shatnawi and Wei Li. 2011. An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering and Its Applications* 5, 4 (2011), 127–149.

Yogesh Singh and Ruchika Malhotra. 2012. *Object oriented software engineering*. PHI Learning Private Limited.

Daniela Steidl and Florian Deissenboeck. 2015. How do Java methods grow?. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 151–160.

K. Stroggylos and D. Spinellis. 2007. Refactoring–Does It Improve Software Quality?. In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*.

Gábor Szőke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2017. Empirical study on refactoring large-scale industrial systems and its effects on maintainability. *Journal of Systems and Software* 129 (2017), 107–126.

Gábor Szőke, Csaba Nagy, Rudolf Ferenc, and Tibor Gyimóthy. 2014. A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve Source Code Quality. In *Computational Science and Its Applications – ICCSA 2014*. 524–540.

S. Tarwani and A. Chug. 2016. Sequencing of refactoring techniques by Greedy algorithm for maximizing maintainability. In *2016 International Conference on Advances in Computing, Communications and Informatics*. 1397–1403. DOI:http://dx.doi.org/10.1109/ICACCI.2016.7732243

Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Trans. Softw. Eng.* 35, 3 (May 2009), 347–367. DOI:http://dx.doi.org/10.1109/TSE.2009.1

Santiago Vidal, Iñaki berra, Santiago Zulliani, Claudia Marcos, and J. Andrés Díaz Pace. 2018. Assessing the Refactoring of Brain Methods. *ACM Trans. Softw. Eng. Methodol.* 27, 1, Article 2 (April 2018), 43 pages. DOI:http://dx.doi.org/10.1145/3191314

S W. A. Rizvi and Prof. Raees Khan. 2010. Maintainability Estimation Model for Object-Oriented Software in Design Phase (MEMOOD). *Journal of Computing* 2, 4 (2010), 26–32.

Lu Xiao, Yuanfang Cai, and Rick Kazman. 2014. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 967–977.

Aiko Yamashita and Steve Counsell. 2013. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software* 86, 10 (2013), 2639–2653.

Aiko Yamashita and Moonen. Leon. 2013. To what extent can maintenance problems be predicted by code smell detection? – An empirical study. *Information and Software Technology* 55, 12 (2013), 2223–2242.