Journal of Computer Science and Information Technology
June 2015, Vol. 3, No. 1, pp. 01-13
ISSN: 2334-2366 (Print), 2334-2374 (Online)
Copyright © The Author(s). All Rights Reserved.
Published by American Research Institute for Policy Development
DOI: 10.15640/jcsit.v3n1a1
URL: http://dx.doi.org/10.15640/jcsit.v3n1a1

The Downsides of Software Refactoring

Jason R. Frier[1] & Robert F. Roggio[2]

Abstract

Software quality is often measured in terms of both external usually indirectly measureable quality attributes and internal often directly measurable software metrics. While both professionals and academics alike are attracted to numbers that can be analyzed and compared and inferences made, these are not often available or, at best, are difficult to acquire. Sometimes external quality factors such as maintainability and reusability can be assessed without mapping to internal quality metrics and often direct software measures are taken without regard to indirectly measurable quality attributes. Still other times, we like to simply look internally and count the number of methods in a class or carefully look at the size of a class, or perhaps the number of parameters passed to a method and consider such metrics measures of software quality. These interests have given rise to a considerable amount of research on refactoring, but, unfortunately, research continues to indicate that some refactoring efforts can lead to poor software quality. This paper exams some of the literature on refactoring in order to encapsulate both positive and negative impacts of refactoring: where refactoring might be useful and where it might be avoided.

Keywords: refactoring, code quality, internal refactoring, external quality attributes

I. Introduction

In today's intense world of software engineering, one of the most pressing matters is how to make software easier to maintain while keeping costs down. The reality is that as a software system ages, the likelihood is that maintaining the software may well become a costly endeavor. This is a result of code "decay."

[1] School of Computing, University of North Florida, Jacksonville, FL USA 32224.
Email: northflorida904@gmail.com
[2]School of Computing, University of North Florida, Jacksonville, FL USA 32224.
Email: broggio@unf.edu

Code smells are symptoms that are detected within the software that can signal the need to address specific code decay. Refactoring is the preferred method to deal with code decay, both in fixing and preventing it. According to Kessentini, there are two steps to refactoring code. These two steps are: 1.) determining when a system needs to be refactored, and 2.) determining how to best refactor the problem by determining which refactoring actions need to be performed. The commonly used term, “Code Smells” refers to structures within code that suggest a refactoring technique might be used to improve the code structure. (Kessentini et al. 2013). According to Buschmann, refactoring isn’t just “the process of changing a software system in such a way that it does not alter behavior of the code yet improves its internal structure.” (Buschmann 2011). Refactoring must meet the following conditions: it can only improve developmental qualities of the code, like maintainability; published contracts cannot be changed; and refactoring cannot be a substitute for actual bug fixing. Thus, adding new functionality would not qualify as a refactoring activity, as it violates the first rule, namely that only developmental qualities such as maintainability are allowed under the refactoring umbrella. Buschmannalso states that it’s usually only the design of the code that “smells” bad, since refactoring almost certainly always occurs on working code, that is, code that currently provides functionality. However, Buschmanalso admits that refactoring is an indispensable tool to the developer, allowing him/her to maintain the health of the system while saving money that would be incurred through future maintenance rework.

One of the first to actually publish a work on the topic of refactoring (specifically, refactoring object-oriented systems) was William Opdyke in his doctoral thesis entitled, “Refactoring Object-Oriented Frameworks”. In his thesis, Opdyke talked about the need to be able to design object-oriented systems in such a way that each subsequent design iteration could accommodate the accompanying changes made to the system, thus making overall design of the system easier, since it could accommodate frequent change. He proposed automated “restructuring”, or refactoring, of the system as an approach to ensure that the system was able to remain flexible to change.

However, Opdyke's thesis references the fact that practitioners in the software development field felt that object-oriented systems were easier to change than procedural systems, and this seems to lead to the conclusion that software practitioners had been practicing refactoring on not only object-oriented systems, but procedural systems as well, for some time prior to the publishing of his doctoral thesis in 1992. Opdyke's paper is nonetheless often credited as one of the theoretical foundations upon which modern refactoring techniques are based on.

II. Empirical Studies in Refactoring

A.Empirical Results of Refactoring – the Shrivastava Study (2009)

A study published by Vishal and Suprika Shrivastava addresses a number of practical complexities of refactoring. The application that formed the basis of their study was known as Inventory Delux 1.03 and contained only three classes with thirty-eight total methods. The system also contained three additional files. The authors of the study used the following metrics to assess where “code smells”might lie in the system: Number of Attributes (NOA), Number of Classes (NOC), Number of Methods (NOM), Depth of Inheritance Tree (DIT), Cyclomatic Complexity (CC)(McCabe, 1976), and Total Lines of Code (TLOC). (Shrivastava) To begin with, the authors detected a smell in a GUI class, which was a large class with many responsibilities including displaying components of each tabbed pane of the GUI along with each function performed in each pane. Two methods within the class were also very long, making the class difficult to understand. The class had a Cyclomatic Complexity of 8.875 before refactoring and one of the two methods had a CC of 56. Using the refactoring method *Extract Class*, a total of six classes were extracted. The responsibility of displaying and working inventory-related functionalities was carefully assigned to these six new classes, the result of which was a significant improvement in understandability. The complexity of the original class was reduced to 5.125, and the complexity of the large method having a CC of fifty-six was reduced to twenty-six. The length of this one method was reduced by 179 lines of code. The Shrivastava's continued their refactoring using *Extract Method*. One class contained a constructor with duplicate code found in one of the class's methods. *Extract Method* was performed, and as a result one new method was created out of the constructor and the other method.

This new method was reused when necessary. Shrivastava cites that this refactoring reduced the total lines of code for the class as a result of the elimination of the duplication. The GUI for the application was developed using Java Swing. Specifically JPanel class was implemented. Refactoring was applied to not only the JPanel class but also to J Label and J Text Field. *Extract Subclass* was applied with a result of creating three additional classes. Rather than using the aforementioned classes, these extended classes were used. Because of this refactoring, while considerable duplication was eliminated, the DIT increased. Shrivastava reported less understandability and increased complexity.

In sum, the overall impact on software quality metrics for the Inventory Delux 1.03 application is as follows: The NOA changed from an average of 37 to 11.3 over the course of fourrefactorings (resulting from more classes and redistribution of the attributes). The NOC increased as more classes were extracted, resulting in an increase in complexity and a corresponding increase in DIT. NOC increased from three to twelve over the course of fourrefactorings, as more classes were developed. NOM decreased from an average of 12.67 to 5.5 over four versions, which makes sense as there are fewer classes. Cyclomatic Complexity decreased on average from 3.39 to 2.43 with a decrease in the maximum Cyclomatic Complexity from 56 to 13 over four revisions. The TLOC decreased from 1103 to 508 over the course of these efforts as significant duplication was expunged. As stated, duplication was drastically reduced, but the DIT increased on average from 2.67 to 4.5 over the course of four refactoring exercises. Complexity increased as new classes and subclasses were extracted. The findings of the Shrivastava study suggest that overall code structure improved over the course of four refactorings of the application. The number of lines of code decreased significantly, making the application smaller, more maintainable, and ultimately easier to understand. However, the *Extract Class* refactoring led to an increase in the number of classes, and an increase in the DIT, with an attendant overall increase in complexity. According to Shrivastava, while the quality of the application improved, this same application became more complex rendering it more difficult to maintain.

B. More Empirical Results of Refactoring – the Dandashi Study (Dandashi, 2002)

In a study by Dandashi, the possibility to develop correlations between directly measurable software metrics and indirectly measurable quality attributes was undertaken.

According to Dandashi, directly measurable software metrics are code metrics based on the syntax and behavior of that code, while indirect measurable quality attributes are those that typically cannot be ascertained from code syntax and behavior. In his research, Dandashi set out to automate the gathering of software metrics for C++ components. The metrics chosen for his study included both class level and method level metrics. Method level metrics included McCabe's Cyclomatic Number (MCC), Halstead's Volume (Halstead, 1977), and Physical Source Statements (PSS), while class level metrics included Weighted Methods per Class (WMC), Depth of Inheritance tree (DIT), Number of Children (NOC), Response For a Class (RFC)(Number of Distinct Methods and Constructors invoked by a Class), Coupling Between Objects (CBO), and Lack of Cohesion in Methods (LCOM).

These metrics were gathered via an automated tool. Then, groups of software practitioners and graduate students were presented with a survey asking them to assess indirect quality attributes of the C++ components that were run through the automated metric gathering tool. Indirect quality attributes included adaptability, flexibility, maintainability, understandability, portability, reliability, expandability, completeness, correctness, and modularity. The survey participants were asked to evaluate the code components to assess levels of these quality attributes. Once the surveys were collected, correlation coefficients were calculated using SPSS to determine the relationship between the directly measurable metrics and the indirectly measurable quality attributes. The results of this study provide some very interesting data. As DIT and NOC increased (directly measurable software metrics), adaptability, completeness, maintainability, and understandability decreased (indirectly measurable software quality attributes). As RFC and CBO decreased (more directly measurable software metrics), the indirect measurable quality attributes increase. However, as coupling decreased (class level directly measurable software metric) as a result of larger, more self-contained classes, a decrease in adaptability, maintainability, and understandability (indirectly measurable quality measures) was discovered. Further research by Dandashi revealed that complexity and LOC were found to be *directly* proportional to indirect software quality attributes, while NOC, DIT, RFC, and CBO are *inversely* proportional to indirect software quality attributes. (Dandashi, 2002)

C. More Empirical Results of Refactoring – the Elish and Alshayeb Study (Elish, 2011)

Elish's study is very significant as it expanded the work by Dandashi by taking code components, measuring their internal directly measurable metrics (based upon Dandashi's metrics) and then performing nine different refactoring activities on that code. After refactoring activities, he then re-analyzed a number of software metrics to note any changes due to the refactoring. Changes in metrics due to refactoring were then mapped to external, indirect quality attributes based on the findings from Dandashi's study. Refactoring activities chosen by Elis hand Alshayeb, as cited in their study (Elish 2011) included *Consolidate Conditional Expression, Encapsulate Field, Extract Class, Extract Method, Hide Method, Inline Class, Inline Method, Inline Temp, and Remove Setting Method.* The results of their refactoring exercises using directly measurable quality metrics were:

- Consolidate Conditional Expression: rendered the class less cohesive; increased NOM and decreased LOC; decreased adaptability and maintainability
- Encapsulate Field: Made class less cohesive, increased NOM and LOC; increased adaptability and maintainability
- Extract Class: Made class more cohesive, decreased NOM and LOC, increased CBO, increased number of classes; decreased adaptability and maintainability
- Extract Method: Made class less cohesive, increased NOM and LOC; increased adaptability and maintainability
- Inline Method: Made class more cohesive, reduced NOM and LOC; decreased adaptability and maintainability

As discussed in their study, correlations were made between certain refactoring activities and internally directly measurable metrics and external quality attributes. In short, the refactoring activities *Consolidate Conditional Expression, Extract Class,* and *Inline Method* decreased adaptability and maintainability, while *Encapsulate Field* and *Extract Method* increased adaptability and maintainability. It seems safe to assert that they have shown that while some refactoring activities improve overall software quality, other activities may be a clear detriment to software quality.

D. More Empirical Results of Refactoring – the Shatnawi and Li Study

RaedShatnawi and Wei Li (Shatnawi and Li, 2011) undertook a similar study in which they proposed a set of refactoring heuristics that were used to correlate internal software metrics with external quality attributes, and in doing so set out to validate the extent to which refactoring has a positive or negative impact on software quality. They proposed a quality model known as the QMOOD (Quality Model for Object Oriented Design). The QMOOD is a hierarchical model used to assess external quality factors using internal design metrics that can be used at both the system and the component levels. Thus this research took a different approach than their predecessor's work. The Shatnawi and Li Study looked at four external quality attributes. First, they considered *reusability*. Reusability was defined as the degree to which a software module could be used in other programs or systems. Secondly, they considered *flexibility*, which they defined the degree to which a system or component can be used within environments for which it wasn't designed. Thirdly, *extendibility* was chosen to measure the ease with which the system or component can be modified to increase its functional capabilities and lastly, *effectiveness* was chosen as a measure to see if a component achieves desired functionality. These four external quality attributes were chosen inthis study to assist in measuring the degree to which refactoring improved software quality. The QMOOD model linked the following internal metrics (ahead) with the previously-mentioned four quality attributes. First, *reusability* was linked to coupling, cohesion, messaging, and design size. *Flexibility* was linked to encapsulation, coupling, composition, and polymorphism. *Extendibility* was linked to abstraction, coupling, inheritance, and polymorphism, and *effectiveness* was linked to abstraction, encapsulation, composition, inheritance, and polymorphism.

Shatnawi and Li explained how the QMOOD model was used in *Extract Class* refactoring. They found that when *Extract Class*was undertaken, internal measurable metrics such as design size increases, abstraction stays the same, encapsulation stays the same, coupling increases, cohesion increases, composition increases, inheritance stays the same, polymorphism stays the same, and messaging stays the same,while reusability, flexibility, and effectiveness all improve, and extendibility deteriorates. So simply using this refactoring by itself had varying impacts on software quality. Interestingly, Shatnawi and Lialso showed there were what they termed safe refactorings and unsafe refactorings. Safe refactors left the quality of the software in a desirable state.

In contrast, unsafe refactors had a detrimental impact on software quality. An example of a safe refactoring was the *Encapsulate Field* action. The encapsulate field action improved reusability, flexibility, and effectiveness, and had no impact on extendibility. The software was left in an overall improved state. An example of an unsafe refactoring is the *Pullup Method* action. This action was shown to have a negative effect on all the external quality attributes, and therefore left the software in a deteriorated state in terms of overall quality. Lastly, Shatnawiand Li suggested allocating refactoring actions into categories. The *Composing Methods Category* had a high impact over 50% of the time on the internal metrics of messaging and coupling. The *Moving Features Between Objects Category* of refactors had a high impact over 50% of the time on the internal metrics of coupling and cohesion. The *Organizing Data Category* of refactors had a high impact over 50% of the time on the internal metrics of messaging, coupling, design size, and composition. The *Simplifying Conditional Expressions Category* of refactors had a high impact over 50% of the time on the messaging internal metric. The same was true for the *Making Method Calls* refactoring category.

E. More Empirical Results of Refactoring – the Kannangaraand Wijayanayake Study (2013) A Different Approach

S.H. Kannangaraand Wijayanayakeperformed a study on the impact of refactoring on code quality. Refactoring techniques used in the study were *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*. Their study evaluated external quality factors <u>without</u> mapping to internal quality metrics, as was the case in other studies. The external quality factors evaluated in this study were maintainability, which they decomposed into *analyzability* and *changeability*; and *efficiency*, specifically *resource utilization* and *time behavior* (These are discussed ahead) The study was conducted on a system that implemented C# using Visual Studio as the development environment. Their study was conducted in two steps. In Step 1, participants were divided randomly into two groups: the refactored code group and the non-refactored code group. During Step 1, sixty students were chosen based on previous programming experience and were randomly place into one of the two aforementioned groups. They were first given questions to answer, and then were given code containing bugs and were asked to fix the bugs.

This was the case for both the experimental (refactored code) and control (non-refactored code) groups. Kannangaraand Wijayanayakeclaim that the bug fixes addressed the issue of *changeability*. Step 2 addressed the issues of *resource utilization,* namely *memory consumption* and *time behavior* for the control and experimental groups of code. Step 2 was carried out in a software testing environment. Before the study was conducted, the four hypothesis categories were established. The first hypothesis group predicted whether the *analyzability* of the refactored code would be higher or lower than the non-refactored code. The second hypothesis group predicted whether the *changeability* of the refactored code would be more difficult or easier than the non-refactored code. The third hypothesis category dealt with whether the *response time of the refactored code* would be longer or shorter than the non-refactored code. The fourth hypothesis category dealt with whether the *efficient utilization of computer resources* was lower or higher for the refactored code as opposed to the non-refactored code.

The results of the Kannangara and Wijayanayake study are as follows in regards to hypothesis predictions: The analyzability of the refactored code was lower than the non-refactored code, except for the refactoring technique Replace Conditional with Polymorphism. The changeability was more difficult for refactored code than for non-refactored code. The response time of the refactored code was longer than for the non-refactored code for a majority of the techniques employed. For resource utilization, the techniques Duplicate Observed Data and Extract Subclass resulted in better utilization scores. For the rest of the refactoring techniques, there was insufficient evidence to prove that there was an improvement of resource utilization for the refactored code. All refactoring techniques, except for Replace Conditional with Polymorphism, showed a propensity to deteriorate the quality of the code more than improving it, with the majority of the techniques showing a 75% deterioration rate. Replace Conditional with Polymorphism was the only refactoring technique used that showed a propensity to improve the quality of the code. From the ten refactoring techniques used in the study, analyzability, changeability, and time behavior showed an overwhelming deterioration in quality. Resource utilization remained largely unchanged, showing only a small improvement.

III. Application, Conclusions, and Future Work

A. Refactoring Mitigates Change

It is important to note that refactoring is a perfective maintenance technique that may be used to improve the internal structure of existing code while ensuring its external behavior unchanged. This principle is essential. In theory, changes to a system result may result in more complex systems that are difficult to maintain and understand. In addition, changes introduced to a system can make that system difficult to reuse. Refactoring is one of the primary ways in which developers mitigate change. (Wake, 2004)

B. Refactoring as a Set of Safe Transformations

Refactoring should take place as safe transformations (Wake, 2004). Refactoring, in theory, always leaves the code in a better state in which it was discovered. Research reveals that this may not necessarily always be the case. This is always the goal, however, of refactoring. Wake says that refactoring does not mean making just any change to the code; rather, if new functionality is added to the system, then this is not refactoring and refactoring did not take place. Refactoring also does not involve generating new functionality from “scratch”. One goal of refactoring is to preserve the knowledge contained in the existing code. The transformations of code absolutely need to be “safe” and code definitely should not be left in a non-working state, as this would violate the agile manifesto. The results of research presented indicates that refactoring does not always benefit the software system in question. Sometimes, according to the literature, refactoring will improve the*structure of the code* while lessening the *overall quality* of the software system, and conversely.

C. Refactoring and Internal Software Metrics and External Quality Attributes

A number of studies and their results assessing refactoring as these techniques apply to internal program metrics and external quality attributes have been presented. The literature is filled with additional case studies. In reducing scope, this paper has looked at studies addressing internal directly measurable software metrics and indirect quality attributes and their relationships, if any, in accordance with several studies. One of the key aspects involved in the studies pertaining to refactoring and its impact on software quality is the concept of metrics.

There are usually two main types of metrics employed: internal, directly observable metrics such as Lines of Code and Lack of Cohesion Measure and external quality attributes that are often more difficult to assess quantitatively. These include quality factors such as maintainability, changeability, and reusability. Many researchers attempt to map the direct internal metrics to external attributes. Some researchers, such as Elish and Alshayeb, used prior research to map directly measurable quality metrics to external quality factors. Others, such as Kannangara and Wijayanayake, measured external quality factors directly, without mapping to internal quality metrics. In the latter's case, this was done primarily through experimental groups answering questions and performing software fixes on bugs, as well as factors such as resource utilization being measured in a controlled lab environment. It becomes clear after looking at only a few research studies that there are clear tradeoffs that should be considered when undertaking refactoring. Certain refactoring actions like *Extract Class* tend to create more classes, which increases the *Depth of Inheritance Tree*, which increases overall complexity. Other refactoring actions, such as *Extract Method*, can be used to reduce overall *Total Lines of Co*de and reduce duplication in the code, which decreases the size of the application and increases maintainability, thereby seemingly increasing overall software quality for the application.

D. Future Work

What remains to be seen is this: How effective are mappings between directly measurable internal metrics and external quality attributes? Both are so extremely important to overall software quality. To what extent does increasing maintainability and size of application warrant increasing application complexity? Is complexity something that can always be sacrificed when it comes to decreasing the TLOC and reducing duplication? Is reducing duplication more important than increasing complexity? There seems to be a complicated interconnectedness when it comes to refactoring. Shatnawiand Li's study suggests that grouping refactoring actions into categories on which have a high impact on the internal metrics and which ones do not ultimately gives the development team more knowledge of their environment, and they will be better suited to choose and pick which refactoring actions to perform. Overall, by looking at how individual refactoring actions improve or deteriorate the quality of the code, a development team will be more educated. To say that knowledge is power in this situation is a bit of an understatement.

Understanding which refactoring actions will affect the code in certain ways makes the task of creating maintainable, efficient software that much easier for the developers. Refactoring seems to both improve and hinder software quality at the same time. This may not seem to be such an important thing when dealing with small applications that have fewer than twenty classes. But what about large, complex applications? Performing the *Extract Class* refactoring on such an application will likely create even more classes. Is this something that is worthwhile and beneficial for developers? It is clear from the research that much more research is indeed warranted. Refactoring is a mixed blessing.

References

Buschmann, Frank. "Gardening your Architecture, Part 1: Refactoring." IEEE Software. 2011; Downloaded from Web: 20 December 2014,http://eds.a.ebscohost.com.dax.lib.unf.edu/eds/detail/detail?vid=6&sid=490a5e07-cfd6-4543-863b-f6613b336922%40sessionmgr4001&hid=4210&bdata=JnNpdGU9ZWRzLWxpdmU%3d#db=iih&AN=62026654

Dandashi, Fatma. "A Method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements." ACM 2002.Downloaded September 2014: http://dl.acm.org.dax.lib.unf.edu/citation.cfm?id=508791.508985&coll=DL&dl=ACM&CFID=464392331&CFTOKEN=63296891

Elish, Karim, and Mohammad Alshayeb. "A Classification Of Refactoring Methods Based On Software Quality Attributes." Arabian Journal For Science & Engineering (Springer Science & Business Media B.V.) 36.7 (2011): 1253-1267. Academic Search Complete. Downloaded from Web. 12 Sept. 2014.http://eds.b.ebscohost.com.dax.lib.unf.edu/eds/detail/detail?vid=2&sid=142e453b-42d7-4099-845e-d3752604b0fc@sessionmgr111&hid=112&bdata=JnNpdGU9ZWRzLWxpdmU=#db=a9h&AN=67186817

Halstead, Maurice H.. Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc., 1977, ISBN 0-444-00205-7.

Kannangara, S.H. and W.M.J.I. Wijayanayake. "Impact of Refactoring on External Code Quality Improvement: An Empirical Evaluation." International Conference on Advances in ICT for Emerging Regions. IEEE 2013. Downloaded from Web. 25 Nov 2014.http://ieeexplore.ieee.org.dax.lib.unf.edu/xpl/articleDetails.jsp?tp=&arnumber=6761156&queryText%3DImpact+of+Refactoring+on+External+Code+Quality+Improvement%3A+An+Empirical+Evaluation

Kessentini, Marouane, Rim Mahaouachi, Khaled Ghedira. “What you like in design use to correct bad smells.” Software Quality Journal. 2013. Downloaded from Web: 19 December 2014, http://eds.a.ebscohost.com.dax.lib.unf.edu/eds/detail/detail?vid=4&sid=490a5e07-cfd6-4543-863b-f6613b336922@sessionmgr4001&hid=4113&bdata=JnNpdGU9ZWRzLWxpdmU=#db=iih&AN=90015765

McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, December 1976, p.308–320.

Opdyke, William F. “Refactoring Object-Oriented Frameworks.” Doctoral thesis, 1992, Downloaded from Web: 19 December 2014, http://www.ai.univ-paris8.fr/~lysop/opdyke-thesis.pdf

Shatnawi, Raed and Wei Li. “An Empirical Assessment of Refactoring Impact on Software Quality using a Hierarchical Quality Model.” International Journal of Software Engineering and its Applications. Oct. 2011. Downloaded from Web. 12 Sept. 2014http://eds.b.ebscohost.com.dax.lib.unf.edu/eds/detail/detail?vid=5&sid=142e453b-42d7-4099-845e-d3752604b0fc%40sessionmgr111&hid=108&bdata=JnNpdGU9ZWRzLWxpdmU%3d#db=iih&AN=67545955

Shrivastava, Suprika, and Vishal Shrivastava. “Impact of Metrics based Refactoring on the Software Quality: A Case Study.”IEEE 2009. Downloaded from Web. 24 Sept. 2014.http://ieeexplore.ieee.org.dax.lib.unf.edu/xpl/articleDetails.jsp?tp=&arnumber=4766459&queryText%3DImpact+of+Metrics+based+Refactoring+on+the+Software+Quality%3A+A+Case+Study

Wake, William C. Refactoring Workbook. Boston. Pearson Education 2004, Addison-Wesley Professional, ISBN-10-0321109295