Studying the Effect of Refactorings: a Complexity Metrics Perspective

Quinten David Soetens and Serge Demeyer
Department of Mathematics and Computer Sciences
University of Antwerp
{quinten.soetens, serge.demeyer}@ua.ac.be

Abstract—Refactoring is widely recognized as a way to improve the internal structure of a software system in order to ensure its long-term maintainability. Consequently, software projects which adopt refactoring practices should see reductions in the complexity of their code base. We evaluated this assumption on an open source system —namely PMD, a Java source code analyzer— and discovered that periods of refactorings did not affect the cyclomatic complexity. This paper investigates this counterintuitive phenomenon through a detailed analysis of the actual source code manipulations applied on the system under study.

I. INTRODUCTION

Already in the early 70’s, Manny Lehman and Les Belady perceived the chaotic nature inherent in the life-cycle of any real-life software system. Based on their work with IBM, they formulated the “laws of software evolution” [1]. The first two of these laws have become staples in many software engineering textbooks.

- Continuing Change: A program that is used in a real-world environment must change, or become progressively less useful in that environment.
- Increasing Complexity: As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure.

The law of “Continuing Change” provides a darwinistic view on the life-cycle of a software system, basically stating that a software system must adapt to its environment in order to survive. The law of “Increasing Complexity” on the other hand resembles the law of entropy in thermodynamics, expressing that a system strives towards a maximum state of disorder unless extra energy is added.

Refactoring is widely recognized as one of the principal techniques to temper the effects of “Increasing Complexity”. The key idea is to redistribute instance variables and methods across the class hierarchy in order to simplify the structure of a software system whilst preserving the behavior of the system and consequently prepare the software for future extensions [2]. If applied well, refactoring is said to improve the design of software, make software easier to understand, help to find bugs, and help to program faster [3]. As such, refactoring has received widespread attention within both academic and industrial circles, and is mentioned as a recommended practice in the software engineering body of knowledge (SWEBOK) [4].

Given the law of “Increasing Complexity”, and knowing that refactoring is meant to simplify the structure of a software system, one would assume that software projects which adopt refactoring practices should see reductions in the complexity of their code base. That is, after a period of refactoring one would expect drops in the total complexity of a software system. Indeed, Demeyer et. al. have used decreasing size measurements for identifying refactorings in a project’s history [5], hence it is likely that drops in complexity measurements are indications of refactoring activity as well. The inverse is not true however. Many refactorings —most notably the move refactorings— will not affect the complexity, but merely keep it at the same level. However as atomic refactorings are combined into chains of complex ones, we still expect a reduction in overall complexity.

This paper reports on a pilot case study to test the assumption that refactorings indeed reduce the complexity. To measure the complexity of a program, we opted for the well-known cyclomatic complexity from McCabe, which measures the complexity of the decision logic of a piece of code. Based on a mathematical analysis, we first deduce the expected effect of a given refactoring on the cyclomatic complexity metric; one of increase - decrease - equalize (Section II). Next, we describe the experimental set-up, where we analyze subsequent revisions of an open source system —namely PMD, a Java source code analyzer—to seek for revisions of the system where the complexity decreases. As a cross-validation, we compare these revisions against the ones marked in the commit-messages as being refactorings (Section III). We count those revisions where the measured effects of the refactorings matched the documented effect and where they did not. Moreover, we interpret the matches and mismatches by providing concrete examples of the refactorings as we observed them in this particular case (Section IV). Finally, we discuss the threats to validity (Section V), related work (Section VI), and future work (Section VII), before concluding the paper with a summary of our results (Section VIII).

II. MATHEMATICAL ANALYSIS

In this section we look at three refactorings (pull up method, extract method and inline method) and explain how they are

expected to affect the complexity of a system.

As a measure for complexity, we use McCabe’s well known cyclomatic complexity (CC) which was defined by Thomas McCabe in 1976 [6]. It measures the amount of decision logic in a single method. Each time there is a branch (if, for, while, do, case, catch or the ternary operator ‘ : ?’) or a boolean operator (&& or ||) the metric is increased by one. In other words it counts the number of flows through a method. This means that the complexity for a method is at least one.

As a measure for the complexity of the entire project we sum the CC for every method in the project. So we assume that when adding a method to the system the complexity of the system will increase by at least one.

Table I summarizes the refactorings and what the expected effect is on the complexity of the software system. This is explained in more detail in the rest of this section. We take \( C_0 \) as the complexity of the system before the refactoring, \( C_1 \) as the complexity of the system after the refactoring and \( x \) as the complexity of the method upon which the refactoring is performed.

### A. Pull up Method Refactoring

In a Pull Up Method refactoring we move a method from a class to a superclass and subsequently remove the duplicate implementations from the sibling classes. Suppose \( c \) is the number of times the method occurred (i.e., the number of duplicates plus the original) then we can write the complexity after the refactoring as:

\[
C_1 = C_0 + x - cx
\]

So the complexity is changed by adding a method in the superclass (+\( x \)) and subsequently removing the method from the subclasses that implemented it (−\( cx \)). We can now see that the complexity remains the same when only one (\( c = 1 \)) occurrence was removed from the subclasses (which is basically a move method operation). When more occurrences are removed (\( c > 1 \)) the complexity will be lower. The complexity can only increase if no occurrences are removed from the subclasses (\( c = 0 \), but then we would simply add a new method to the superclass and this would no longer be a pull up method operation.

### B. Extract Method Refactoring

The extract method refactoring creates a new method from a piece of code. In addition, occurrences of this code are replaced by calls to the newly created method. If we take \( c \) as the number of occurrences of the extracted piece of code that are replaced by a method call, we can write the changed complexity as:

\[
C_1 = C_0 + x - (x - 1)c
\]

The complexity changes by adding a new method (+\( x \)) and it decreases by removing \( c \) occurrences of the extracted piece of code (\( x - 1 \)). We can see that the complexity will increase when only one occurrence is changed by a method call (\( c = 1 \)) in this case the complexity is simply increased by one, for creating a new method. The decision logic contained in the method body is simply moved. Another way the complexity can increase is if the complexity of the extracted method is one (\( x = 1 \)), meaning that we simply extracted some statements without any decision logic.

The complexity will decrease when the extracted piece of code occurred more than one time and the complexity of the method is more than one. So if we extract something complex and remove several duplications of this, the complexity will be lower.

There is one exception, that is when the complexity of the new method is two and the piece of code occurred twice. In that case the complexity remains unchanged. We can illustrate this with the following example. In Listing 1 we see on the left a method \( \text{foo} \) that has complexity four, if we extract the for loop to a new method \( \text{bar} \), as is shown on the right then the complexity of the method \( \text{foo} \) is reduced to two and the complexity of the new method is also two. So the total complexity remains four.

### C. Inline Method Refactoring

The inline method refactoring is the inverse of the extract method refactoring. So if we take \( c \) as the number of calls that are inlined, we can write the change in complexity as:

\[
C_1 = C_0 - x + (x - 1)c
\]

Here the complexity decreases when the method was called only once or the method had complexity one, resulting in one less method after the inline operation, which reduces the complexity by one. If the method had a higher complexity and it was called more than once the complexity would have increased, as we would now distribute copies of the same decision logic.

As with the extract method refactoring there is an exception when we had a method of complexity two and we inline it twice, then the complexity remains unchanged as well.

### III. Experimental set-up

To validate our mathematical analysis, we did a pilot case study on one case. We calculated the complexity for a number of consecutive revisions and looked for revisions where the complexity has decreased. We then looked at the changes that were made to the code in these revisions as well as in the revisions that were marked in the commit messages as having
been refactored. This resulted in a number of observations about the evolution of the cyclomatic complexity.

We used Metrics\(^2\), an open source eclipse plugin that calculates several metrics of an eclipse project, to measure CC. From this plugin we exported the calculated metrics into an XML file, which made analyzing the metrics in an automated manner easier. Using this Metrics plugin as well as the SVNKit\(^3\), a subversion Library for Java, we wrote a small eclipse plugin to automate the following steps:

FOR \( i = 1 \) TO target revision DO
  Update project to revision \( i \) with SVNKit
  Calculate Metrics with Metrics plugin
  Export Metrics to \(*.xml\) file
END FOR

We then used a simple XML parser to get the metrics we are interested in out of the XML files.

We selected the open-source software system PMD\(^4\) as our case. It is written in Java and its development history is available on a subversion (svn) repository\(^5\). The SVN log contains commit messages that report on the refactoring activities that were performed. Messages like: “refactored report generation to consolidate”, “minor refactoring”, “refactoring report”, …, We can therefore easily identify those revisions where refactoring activities were done. This approach of identifying refactorings by mining commit notes was first introduced by Ratzinger et al. in [7].

When calculating the complexity we ignored the package “net.sourceforge.pmd.ast”. This package mostly contains generated code for a java parser, which blows up the complexity significantly. Since this is generated code there will be no refactorings performed here and this package has therefore no value in our experiment.

The unit of analysis corresponds to the revisions in the subversion repository. We analyzed revisions 17 to 810. These range over a period of two months of development. We only analyzed those revisions that were compilable (17 revisions contained build errors) since the metrics plugin requires a full build of the project in order to calculate the metrics. This resulted in a total of 776 revisions for which we could calculate the complexity.

\( i = 1 \) TO target revision \( DO \)
  Update project to revision \( i \) with SVNKit
  Calculate Metrics with Metrics plugin
  Export Metrics to \(*.xml\) file
END FOR

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Change in Complexity</th>
<th>Increases when</th>
<th>Decreases when</th>
<th>Equalizes when</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pull Up Method</td>
<td>( C_1 = C_0 + x - xc )</td>
<td>never</td>
<td>the method occurred more than once (i.e., duplicates removed from subclasses)</td>
<td>the method occurred once (i.e., same as move method)</td>
</tr>
<tr>
<td>Extract Method</td>
<td>( C_1 = C_0 + x - (x - 1)c )</td>
<td>the extracted piece of code occurred only once or the complexity of the new method is one</td>
<td>the extracted piece of code occurred more than one time and the complexity of the new method is more than one</td>
<td>the method occurred twice and the complexity of the new method is two</td>
</tr>
<tr>
<td>Inline Method</td>
<td>( C_1 = C_0 - x + (x - 1)c )</td>
<td>the complexity of the method is larger than one and the method is called more than once</td>
<td>the method is called once or the method’s complexity is one</td>
<td>the complexity of the method is two and the method is called twice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Decreased CC</th>
<th>Equalized CC</th>
<th>Increased CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documented</td>
<td>14</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>No Refactoring</td>
<td>27</td>
<td>580</td>
<td>136</td>
</tr>
<tr>
<td>Documented</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE I: Expected Effect of Refactoring on Cyclomatic Complexity

TABLE II: Nr of revisions in each category.

IV. OBSERVATIONS

Figure 1 shows an overview of the evolution of the overall cyclomatic complexity. We can see that overall the complexity keeps increasing, which confirms Lehman’s second law of software evolution.

There is a spike at revision 72, this is because some classes were moved to a separate package, however this move was done with a copy operation in revision 72 followed by a remove operation in the next revision. The copy operation increased the CC whereas the remove operation resulted in a decrease of the same amount. There is also a substantial drop in complexity at revisions 610 and a large increase at revision 631. This is caused by the removal of an entire package (net.sourceforge.pmd.cpd) in revision 610 and this package was brought back in revision 631. A similar observation can be made between revisions 554 and 557, where the file CPD.java (which implemented the copy-paste detector) was removed from the repository and was reintroduced three revisions later in the form of the package net.sourceforge.pmd.cpd.

Every now and then there is a revision that caused a slight drop in complexity. In what follows we, investigate whether these drops are due to the documented refactoring activities. We do this by studying the actual source code changes made in the revisions that contained documented refactorings and in the revisions that showed a decrease in complexity.

Table II shows an overview of the number of revisions in each category. Of the 776 revisions that were analyzed, 41 revisions showed a decrease in CC and 33 had a log message that contained the substring “refactor”. However these two sets overlapped in only 14 revisions. So 14 revisions had both documented refactorings and showed a decrease in CC; 27 revisions had no documented refactorings and showed a decrease in CC and 19 revisions had documented refactorings but with either no change or with an increase in CC.

\(^2\)http://metrics.sourceforge.net
\(^3\)http://svnkit.com/
\(^4\)http://pmd.sourceforge.net/
\(^5\)https://pmd.svn.sourceforge.net/svnroot/pmd/trunk/pmd/
A. Revisions with documented refactorings

a) Decreased CC: There are fourteen revisions with documented refactorings that showed a decrease in complexity. In six cases this is caused by the removal of duplicated code. This is done with the extract method refactoring, or by pulling up a method to a superclass. For instance, what they did in revision 458 was to create a new superclass for rules concerning braces (i.e., the classes WhileLoopsMustUseBracesRule, ForLoopsMustUseBracesRule, and IfElseStmtsMustUseBracesRule are now extended from the newly created abstract class BracesRule). They were now able to pull up the method hasBlockAsFirstChild, which was implemented in all three rules concerning braces. Thus the number of methods was reduced by two, which results in a decrease of two for the cyclomatic complexity.

In four cases out of fourteen the decrease in complexity is due to another commonly used refactoring, namely the inline method refactoring. This occurs for example in revisions 170 and 171, where a total of three methods are inlined causing the CC to drop by three.

In the four remaining cases the decrease in complexity is caused by the removal of code that is no longer needed, either because it is implemented elsewhere or because it is unreachable code.

b) Equalized CC: There were seven revisions that had documented refactorings and where the measurements indicated that the Cyclomatic Complexity remained the same. In two of these two revisions the refactoring consisted of introducing an explaining variable and one of these two revisions also contained a number of rename operations. The renaming of a method or a variable has absolutely no influence on the amount of branches in the source code, hence it does not affect the complexity of the code. This refactoring is merely meant to increase the understandability of the code. The same is true for the introduction of an explaining variable.

We found one occurrence (in revision 659) of the special case of the extract method refactoring (see the example in Listing 1 in Section II-B). In this revision two occurrences of a statement containing the ternary operator were extracted to a new method of complexity two.

c) Increased CC: The CC will be higher when a new method is added. This can be done by extracting a piece of code into a new method as was done in revision 578. In the preceding revision there was a method that started with iterating over a list to check whether any of its elements was already contained in another list. This iteration was extracted to a new method which increased the complexity by one.

However, the most common observation (in seven cases out of twelve) is that besides performing a refactoring there is also a new functionality added to the system, mostly in the form of new test cases. This confirms the conclusions made by Murphy-Hill et al. who state that programmers frequently apply the so called “floss refactoring” were refactoring is interspersed with other programming changes [8].

B. Revisions with no documented refactorings and Lower CC

In the 27 revisions that showed a decrease of CC but had no documented refactoring, it was mostly by removing unnecessary methods, sometimes even unnecessary functionality. For example in revision 148 the functionality to render reports in plain text was removed resulting in a CC drop of eight.

V. Threats to Validity

In this section we identify factors that may jeopardize the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [9], [10]) we organize them into four categories.

Construct validity – do we measure what was intended:

For this pilot case study we use McCabe’s cyclomatic complexity metric to measure the overall complexity of a revision of a program. However, the cyclomatic complexity represents only a single perspective on the complexity of a program and many more are possible [11]. Therefore, as a follow-up to this pilot case study, we will also investigate
alternative complexity metrics. Also, we point out that the cyclomatic complexity is only defined for subroutines, and that we define the complexity of a whole program as the sum of the complexity of its methods. Consequently, we make some small but crucial assumptions during the mathematical analysis of the effect of a refactoring and researchers that want to replicate our results should be aware of them (Section II).

During cross-validation, we relied on the versioning system’s log messages to identify revisions corresponding to refactorings. As no strict conventions are in place for what should be specified in such messages, there may be significant differences in the content and quality of log messages across tasks and developers. Consequently, we might miss certain revisions which do correspond to refactorings.

**Internal validity – are there unknown factors which might affect the causal relationship:** During this research, we investigate whether there is a causal relationship between refactorings as applied on a software system, and (reductions of) the cyclomatic complexity of that system. Due to the particular set-up of the experiment (i.e., analysis of revisions committed in a version control system) it is likely that other changes to the code affect the cyclomatic complexity. In fact, detailed analysis of particular changes revealed that this was frequently the case, as is reported in Section IV. To counter this threat, we cross-validated against those revisions explicitly marked as being refactorings.

**External validity – to what extent is it possible to generalize the findings:** For this pilot case study, we selected PMD because the developers consciously documented when they applied refactorings. We only analyzed the first 800 revisions which mainly corresponded to implementing anticipated requirements, that is the “initial development” phase in the staged process model for software evolution [12]. So far, we cannot claim any results outside this context. Nevertheless, PMD is a representative for a class of software systems produced in smaller teams (five to ten core developers) where team members take up varying roles (analyst-designer, programmer, tester, debugger) and were quality assurance is mostly integrated into the daily activities of the team members. Therefore, we will investigate other systems of that class as a follow-up study.

**Reliability – is the result dependent on the researchers and tools:** For collecting measurements, we mainly relied on external tools (i.e., “Metrics”, an open source eclipse plug-in) accessed through the standard and reliable API of SVN and glued together in an eclipse plugin. During cross-validation we relied on grep to find all commit-logs referring to all variants of “refactor”, which is highly reliable but might miss misspellings.

The manual analysis of the revisions under study (i.e., the ones with reduced cyclomatic complexity and the ones explicitly marked as being refactorings) is highly dependent on the person doing the research, so it is likely that other researchers will make other observations. One of the goals of this pilot study was precisely to learn how to conduct such a manual analysis, and for the follow-up study we will write out a systematic procedure in order to reduce this risk.

**VI. Related Work**

There has already been some research on the effects refactoring has on the complexity of a software system as well as on other quality attributes. A general observation is that refactoring has a positive influence on some quality attributes, a negative impact on others and some studies even reported no changes at all.

Alshayeb did an empirical investigation into the effect refactoring has on several quality attributes: Adaptability, Maintainability, Understandability, Reusability and Testability. He concludes that the effect refactoring has on the quality attributes vary greatly without any consistent trends and he was thus unable to validate that refactoring improves software quality [13]. Stroggylos and Spinellis examined the effect documented refactorings have on several source code metrics. Their conclusion was that either refactoring does not always improve the quality of a software system or developers do not use refactoring effectively [14]. Du Bois et al. did several studies concerning the effect refactoring has on the quality of a system [15] [16]. They did a controlled experiment with final-year computer science students to study the effect refactoring has on the understandability of the code. This resulted in the first empirical support for the claim that refactoring can improve program comprehension [16]. They also developed guidelines for applying refactorings in order to reduce coupling and boost cohesion and they validated these guidelines on an open-source software system [15]. Kataoka et al. also used coupling to evaluate the effect refactoring has on the maintainability of a system. Their conclusions were that coupling is useful in evaluating the effect of refactoring, however the number of refactorings for which this is applicable is limited [17]. Geppert et al. investigated the impact of refactoring on changeability, taking into account customer reported defect rates as well as the effort and the scope of changes. Their results showed that customer reported defects and the effort needed to make changes were significantly decreased [18]. Moser et al. concluded from their research that refactoring increases the productivity, improves certain quality measures and reduces complexity and cohesion [19]. Their other result ended in the conclusion that refactoring has a positive effect on reusability [20].

**VII. Future Work**

As a pilot case study this paper provides a solid foundation for a follow-up study. However subsequent research can go in many different directions. The most obvious future directions are to investigate alternative complexity measures and to investigate other cases both open-source and industrial.

Murphy-Hill et al. have shown that developers often perform refactoring as an unconscious activity and therefore they do not indicate all refactoring activities in the commit logs [8]. Therefore instead of relying on the log messages of the version control system, we can use refactoring detection techniques (such as the ones proposed by Danny Dig [21] or Peter
Weißgerber [22]) to identify revisions where refactoring activities took place. We observed several revisions where the refactorings were not the only changes applied to the code. To counter this, we could do a more fine-grained analysis, where we study the complexity changes at class level instead of on the entire system.

VIII. Conclusion

In this paper, we report the results of a pilot case study to verify the assumption that refactorings reduce the complexity of a software system. Using the well-known McCabe cyclomatic complexity metric, we analyze subsequent revisions of an open source system—namely PMD, a Java source code analyzer—to seek for revisions of the system where the complexity decreases. We compare these revisions against the ones marked in the commit-messages as being refactorings and count those revisions where the measured effects of the refactorings matched the documented effect and where they did not. We interpret the matches and mismatches by providing concrete examples of the refactorings as we observed them in the system under study.

Counterintuitively, we discovered that refactoring practices rarely affect the cyclomatic complexity of a program. Our mathematical analysis showed that while some refactorings—most notably move operations—have no effect on the cyclomatic complexity, most refactorings do indeed reduce the cyclomatic complexity especially when they involve duplicated code. However, if we analyze the commit messages in the version control system, we see that many revisions explicitly marked as performing refactorings, did not reduce the complexity. We also discovered several revisions where the complexity was reduced, however that was mainly because developers removed unnecessary methods, sometimes even unnecessary functionality.

We see three possible explanations for these—at first sight—counterintuitive results. First of all, we noticed that in this particular case, few refactorings did involve removal of duplicated code which is by far the most visible reduction of complexity. Secondly, we have observed that in many revisions, developers mix refactorings with the addition of extra functionality, nullifying the reduction in complexity. This suggests that the refactoring practices applied during the PMD development fall in the category of so-called “floss refactoring” [8]. Thirdly, it seems that the refactorings we have discovered in the project’s history are mainly about cleaning up the code (moving methods and attributes to reduce coupling and improve cohesion) instead of discovering classes that reflect the intrinsic concepts of the problem domain (i.e., split class, replace conditional by polymorphism, ...). This suggests that refactoring as used during the development of PMD so far, mainly tackles the accidental complexity imposed by the programming language, instead of the inherent complexity induced by the problem domain [23].

ACKNOWLEDGMENTS

This work has been carried out in the context of the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy, project MoVES.

REFERENCES