Old habits die hard

Why refactoring for understandability does not give immediate benefits

Ammerlaan, Erik; Veninga, Wim; Zaidman, Andy

DOI
10.1109/SANER.2015.7081865

Publication date
2015

Document Version
Submitted manuscript

Published in
2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings

Citation (APA)
Ammerlaan, E., Veninga, W., & Zaidman, A. (2015). Old habits die hard: Why refactoring for understandability does not give immediate benefits. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings* (pp. 504-507). [7081865] Institute of Electrical and Electronics Engineers (IEEE). https://doi.org/10.1109/SANER.2015.7081865

Old Habits Die Hard: Why Refactoring for Understandability Does Not Give Immediate Benefits

Erik Ammerlaan
Exact International Development
The Netherlands
Email: erik.ammerlaan@exact.com

Wim Veninga
Exact International Development
The Netherlands
Email: wim.veninga@exact.com

Andy Zaidman
Delft University of Technology
The Netherlands
Email: a.e.zaidman@tudelft.nl

***Abstract*—Depending on the context, the benefits of clean code with respect to understandability might be less obvious in the short term than is often claimed. In this study we evaluate whether a software system with legacy code in an industrial environment benefits from a "clean code" refactoring in terms of developer productivity. We observed both increases as well as decreases in understandability, showing that immediate increases in understandability are not always obvious. Our study suggests that refactoring code could result in a productivity penalty in the short term if the coding style becomes different from the style developers have grown attached to.**

I. Introduction

Large, long-lived software systems often cope with the problem of *technical debt* caused by shortcuts that were taken to increase development speed that result in code that is difficult to maintain [1]. A technical debt can be justifiable, e.g., for quickly getting a new product on the market. However, just like actual debts, if the debt is not repaid, one has to pay interest, which takes the form of additional time that developers need to understand and change complex code. Cunningham states that "entire organizations can be brought to a stand-still under the debt load of an unconsolidated implementation" [2].

Refactoring is a technique to restructure source code that can be used to pay off technical debt. Refactoring is claimed to increase developer productivity, to improve the design of software, to make software easier to understand and to help developers find bugs [3]. The argument that the structure of code has an impact on development productivity has been put forward by Martin, who urges developers to take care of their code and start writing *clean code* [4]. However, it has been recognized that few studies have quantitatively assessed the claimed refactoring benefits, especially with regard to development productivity.

There is a gap of knowledge if we consider the claimed refactoring benefits on the one hand and the little empirical evidence on the other hand. Some studies have measured that refactoring can lead to an increase in productivity over multiple weeks, when new features are added to the system (e.g., [5]), or that developers that actually perform refactorings themselves understand the code better [6]. However, we could not find any work that studied whether refactored code increases *understandability* in general, i.e., developers in the team needing less time to understand a piece of code.

This study was performed at Exact[1], an organization that internationally serves entrepreneurs with business software solutions. The continuous development of their multi-tenant cloud platform has led to a large organizational growth with development teams distributed over multiple continents. The rapid growth of the system as well as the number of people working on it, has led to technical debt. More precisely, Exact has been confronted with a decrease in code quality, which has had an effect on the time required to maintain and extend the code. At Exact, it can happen that a software engineer, when fixing a bug or making a small change, needs multiple hours to understand a piece of code, while it turns out that the required change is but a few lines of code. The complexity of the code and functions with side-effects that do more than their names suggest, are what makes program comprehension time-consuming. From this perspective, we are wondering to what extent refactoring can reduce that time? This brings us to our main research question:

RQ Is the productivity of developers influenced by code that is refactored for understandability when performing small coding tasks?

We explicitly set up our study to focus on small coding tasks, as with these tasks the largest part of the development time required to finish them will be taken up by *understanding* the code, and not performing the actual changes in code. Our study was set up as follows:

- 30 developers from 11 different teams participated.
- We performed 5 experiments, each of them associated with a specific class.
- Experimental runs were performed in the Netherlands (10 developers) and in Malaysia (20 developers). At each location, the exact same experiments were performed.

The remainder of this paper is structured as follows: in the next section, we first present the set-up of the 5 experiments. Next, we present the results. After that, we discuss the results and

[1]http://www.exact.com

TABLE I: Classification of refactorings.

Name	Classes added	Description	Unit tests
Small	0	Rename, Extract Method	
Medium	1	Extract Class, Adapter	Yes
Large	> 1	Dividing responsibilities	Yes

put them into perspective. Finally, we draw our conclusions and do some suggestions for future work that would call for a joint effort by academia and the software industry.

II. Experiments

For each experiment, we chose a *code target* and a small *coding task* that developers would have to perform on the target. In each experiment, we provided the experimental group with a refactored version of the component, whereas the control group received the original version. Both groups had to complete the same coding task, which took the form of fixing a small bug that we had inserted in the code, or making a small change in functionality. When a participant thought he had completed the task, we verified the correctness of the changes, and registered the participant's finishing time. The targets were classes from different packages/projects. Considering the size of the system (2.7 million LOC), we assumed that participants would not be very familiar with the chosen targets, which turned out to be mostly correct. Of course, participants were still familiar with the style of the unrefactored code.

For each experiment we formulated this hypothesis:

Hypothesis. If given the refactored code, developers finish the coding task earlier than if the original code were given.

We acknowledged that the term 'refactoring' is not very specific. A refactoring can be very different from another in terms of scale and complexity. For simple refactoring (e.g., rename method), we are not expecting a drastic change in productivity. However, if a large class with too many responsibilities is split up in multiple components that work together, the resulting code becomes very different. Therefore, we categorized our refactorings based on the number of classes that were added while refactoring a component (cf. Table I).

We created small refactorings of three components, where we applied refactorings Rename, Extract Method and Introduce Explaining Variable [3] to 'clean' the code. We created one medium refactoring, where we added one extra class to the system by extraction. In a large refactoring, we took a monolithic class of around 350 lines of code and split it up into multiple classes according to the Single Responsibility Principle [7]. The goal was also to move towards more object-orientation, as many components, though written in an object-oriented language (VB.NET), were written in a 'procedural object-oriented programming' style [8].

Usually, refactoring on a larger scale goes hand in hand with unit testing to mitigate the risk of breaking functionality [9]. Therefore, we added unit tests to our medium and large refactorings and we provided a subset of those unit tests to the experimental group, to determine if those tests would improve their understanding of the code and thus increase their productivity [10]. In those experiments where participants were assigned to fix a bug, we made sure that the unit tests did not include a test that triggered the bug, which would have made the task trivial.

III. Results

In this section we present the results for each experiment. Each experiment targeted the refactoring of a single component. We stopped each small experiment after 30 minutes; 60 minutes were reserved for each medium and large refactoring. The diagrams represent the data from the experiments in both the Netherlands and Malaysia.

A. Small: RejectionNotifier

For the first experiment, the original code contained a method with a deep nesting level of 8 (nested `if` and `for` control structures), from which we extracted several helper methods so that the method names could act as helpful descriptions of the steps in the process. In both the original and the refactored code, participants had to add one line of code to finish the task. Fig. 1.a shows the results as box plots.

We observed quite some variance in the data; apparently the solution to the coding task was not straightforward for everyone (both in the control group as well as the experimental group). Even so, there was a statistically significant difference in completion times between the groups with original and refactored code (Mann-Whitney-Wilcoxon test, $W = 28, p = 0.0179 < 0.05$). Contrary to our expectations, most developers required more time when given the refactored code. Therefore, we could not accept our hypothesis.

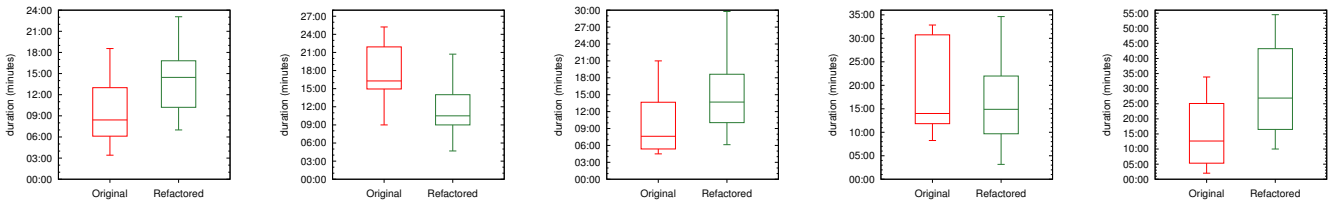
B. Small: PurchaseOrderTools

The second small experiment was very similar to the first one in terms of applied refactorings, though Fig. 1.b shows that the outcome was different. About 75% of the developers with refactored code finished the task before 25% of the developers with the original code did. A possible explanation for this difference in experiments might be that this coding task was experienced as much easier by participants. Therefore any difficulty of the coding task would have caused less noise in the result, so that the difference in time more precisely measures the difference between understanding the original and refactored code.

There was a statistically significant difference in completion times between the groups with original and refactored code (Mann-Whitney-Wilcoxon test, $W = 176.5$, $p = 0.0012 < 0.05$). Therefore, we accept our hypothesis for the second experiment.

C. Small: ContractProlongation

The results for the refactoring surrounding the ContractProlongation class, are similar to the results for RejectionNotifier (see Fig. 1.c). Both experiments showed roughly the same absolute difference between the original and the refactored code. There was a statistically significant difference in completion times between the two groups (Mann-Whitney-Wilcoxon test, $W = 34$, $p = 0.0273 < 0.05$), but the finishing times were



(a) RejectionNotifier (S) (b) PurchaseOrderTools (S) (c) ContractProlongation (S) (d) FinancialEntryTools (M) (e) ERDomain (L)

Fig. 1: Finishing times for small and medium refactoring

higher rather than lower for the developers with refactored code. Again we did not find any evidence for our hypothesis that the coding task would be finished earlier with refactored code than with the original code, so we could not accept it.

A possible explanation for this result is that we could only extract pieces of code to a separate method by requiring that variables from the calling methods were passed to the extracted methods. The flow of method arguments and return values might have been more difficult to understand than a linear flow in a large method.

D. Medium: FinancialEntryTools

Fig. 1.d shows the results for our medium-sized refactoring on the FinancialEntryTools class. Participants in the experimental group were provided with unit tests, which led to an interesting result. The participants were instructed to fix a bug and two participants who were quite experienced in unit testing, found the bug by extending the test suite with a new test case that followed the bug description. By debugging this new failing test case, they found the root cause, fixed the bug, and validated their change by re-running the unit tests.

From a pre-test questionnaire, we know that most participants had little unit testing experience, whereas both participants who used the tests to their advantage in this experiment, indicated that they execute unit tests on a daily basis and that they wrote tests 'multiple times a week'. Indeed, we found weak statistical differences between the results of those with refactored code who wrote tests multiple times a week and those with refactored code who wrote tests less often (Mann-Whitney-Wilcoxon test, $W = 18$, $0.05 < p = 0.1212 < 0.15$). To eliminate the fact that this difference had another cause (e.g., those that often write unit tests might also be more experienced software engineers or have better problem-solving skills), we applied the same statistical test to the control group and found no significant differences in the distribution (Mann-Whitney-Wilcoxon test, $W = 14$, $p = 0.7758 > 0.15$). So, if all participants would have had more experience in unit testing, then likely the difference in time between the two groups would have been bigger.

Looking only at completion times, the Mann-Whitney-Wilcoxon test ($W = 72$, $p = 0.7399 > 0.05$) indicated that it was unlikely that the experimental and control group had different distributions, which can be explained by the fact that there is no major shift in the data between the two groups. Fig. 1.d shows that the medians for both groups are roughly equal, but also that the interquartile range is much smaller for the refactored group, which means that the task was completed earlier by most people when they had refactored code. Also, the test only considers the ranking of data and therefore does not add any significance to the fact that two persons finished much earlier than most others due to their use of unit tests. Therefore, we argue that we produced enough supplementary evidence, to cautiously accept our hypothesis that the task is finished earlier with refactored code (including unit tests).

E. Large: ERDomain

The results in Fig. 1.e show a drastic difference in time between original and refactored code in favour of the original code (Mann-Whitney-Wilcoxon test, $W = 36$, $p = 0.0374 < 0.05$). The results do not support our hypothesis that the coding task can be finished faster with refactored code. It might take more time to understand the relations between classes that emerge from a large refactoring (we implemented the Strategy pattern), especially if a developer is used to working with large classes.

However, looking beyond the mere data, we noticed an important difference in solution quality. Most developers with original code made a 'quick fix' to fix the bug, whereas those with refactored code fixed the root cause. Ergo, the refactored code led to better solutions.

IV. Discussion

The results indicate that the productivity of developers is indeed influenced by code that has been refactored for understandability, though not necessarily in the way one would expect. In this section we make some additional observations.

A. Habits

When the original and refactored code were shown side-by-side after each experiment, most developers appreciated the refactored code and preferred it over the original code. They recognized that having 'clean code' would make the codebase more understandable. This seems in contrast with our results.

We argue that the fact that not all experimental results support the intuitive feeling that tasks would be finished faster with refactored code, might be explained by the habits of developers. The participants were used to working with long, procedural methods and thus were also trained by experience to skim through long methods to understand its structure. When working with small methods, on the other hand, one might have to jump around between different methods to understand a certain feature. Although by extracting details to helper methods one can make the calling method read more like a story, the developers at Exact were not used to

this concept and hence may not have taken full advantage of such aids. Our experiments showed that having refactored code in general does not lead to instant benefits with respect to understandability and developers need time to adjust.

As such, one observation that we make is that when actively working to increase the code quality of a large codebase, it is not only the time investment in the actual refactoring activities that needs to be taken into account, but also the readjustment period for developers to catch up with the refactoring. Since code is read more than it is written [4], the ones who read the new code might not be the ones who performed the refactoring.

B. Other motives for refactoring

Our study focused on refactoring for understandability, striving for 'clean code' that is self-explanatory. However, there are more motives for refactoring, such as to increase maintainability, extensibility, reusability and testability. In general, if a refactoring leads to a (possibly temporarily) decrease in understandability, there could still be increases in maintainability and testability that make the refactoring worthwhile. The changes that were required in the code during the experiments were very small, so that we mostly measured the required effort to understand the code. Note that, if we had designed larger coding tasks, then we would have measured maintainability and the results could have been different.

Observations during our fifth experiment already suggested improvements in maintainability in the long term. We noticed an important difference in the quality of the solutions implemented in original and refactored code, serving as a first indicator that good quality code prevents developers from writing sloppy code. This is in line with the *broken windows theory*, a criminological theory that can also be applied to refactoring [11]: a dirty code base makes developers feel that they can get away with 'quick and dirty' code changes, while they might be less inclined to do so in a clean code base.

In related work Moser et al. [12] found that when time was spent on refactoring during one development iteration, the productivity was significantly higher during the next iteration compared to the average productivity. Furthermore, evidence suggests that the number of defects that are found in a component decreases after it has been refactored, e.g., [13].

C. Threats to Validity

1) Internal Validity: For each experiment, the participants were not told whether they were part of the control or the experimental group. While participants could guess if code was refactored or not, it also happened that people guessed incorrectly or were unsure. Because targets came from different projects and because the refactorings ranged in size, learning to recognize which experiments were refactored and which not, was made difficult. The presence of unit tests in some experiments could have triggered the novelty effect, so that participants spent more time with the unit tests than they would have done otherwise. However, we established that those who took the most advantage of unit tests, were those who have experience with unit testing; others did not spend much time looking at the tests. Finally, some participants could not finish all assignments. This number was roughly equal in both the experimental group and the control group. So it was not related to the dependent variable (refactored/original code), but to the difficulty of the coding tasks.

2) External Validity: Our sample population was representative of the total population of software engineers at Exact. First of all, there were roughly equal numbers of senior and non-senior software engineers in the experimental and control groups. Secondly, since Exact applies global software development, our sample population was also international. The majority of participants were abroad, which is also true for the total population of software engineers. Thirdly, the 30 participants came from 11 different development teams, taking into account that different development teams work on different parts of the code base.

V. Conclusion

In our experiment at Exact, we observed that the productivity of developers can be influenced both positively and negatively by code that is refactored for understandability when performing small coding tasks. So-called 'clean code' does not immediately improve one's understanding of code, if one is used to working with the old structures present in source code. As such, we can say that old programming (understanding) habits do die hard.

We also see our initial observations as a call to arms with regard to investigations that try to shed light on how developers grow attached to a certain coding style, however inefficient, and how difficult it is for developers to change their attitude.

References

[1] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.

[2] W. Cunningham, "The wycash portfolio management system," in *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2. ACM, 1992, pp. 29–30.

[3] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[4] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[5] T. Ng, S. Cheung, W.-K. Chan, and Y.-T. Yu, "Work experience versus refactoring to design patterns: a controlled experiment," in *Proc. Int'l Symp. Foundations of Software Engineering*. ACM, 2006, pp. 12–22.

[6] B. Du Bois, S. Demeyer, and J. Verelst, "Does the "refactor to understand" reverse engineering pattern improve program comprehension?" in *Proc. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE CS, 2005, pp. 334–343.

[7] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[8] M. U. Bhatti, S. Ducasse, and M. Huchard, "Reconsidering classes in procedural object-oriented code," in *Proc. Working Conf. Reverse Engineering (WCRE)*. IEEE, 2008, pp. 257–266.

[9] F. Vonken and A. Zaidman, "Refactoring with unit testing: A match made in heaven?" in *Proc. Working Conf. on Rev. Engineering*, 2012.

[10] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE TSE*, 2014.

[11] J. Kerievsky, *Refactoring to patterns*. Pearson, 2005.

[12] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *Balancing Agility and Formalism in Software Engineering*. Springer, 2008, pp. 252–266.

[13] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. Int'l Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, p. 50.