A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team

Raimund Moser[1], Pekka Abrahamsson[2], Witold Pedrycz[3], Alberto Sillitti[1], and Giancarlo Succi[1]

[1] Free University of Bolzano-Bozen, Italy, [2] VTT Electronics, Oulu, Finland, [3] University of Alberta, Canada
rmoser@unibz.it, pekka.abrahamsson@vtt.fi, pedrycz@ee.ualberta.ca, asillitti@unibz.it, gsucci@unibz.it

Abstract. Refactoring is a hot and controversial issue. Supporters claim that it helps increasing the quality of the code, making it easier to understand, modify and maintain. Moreover, there are also claims that refactoring yields higher development productivity – however, there is only limited empirical evidence of such assumption. A case study has been conducted to assess the impact of refactoring in a close-to industrial environment. Results indicate that refactoring not only increases aspects of software quality, but also improves productivity. Our findings are applicable to small teams working in similar, highly volatile domains (ours is application development for mobile devices). However, additional research is needed to ensure that this is indeed true and to generalize it to other contexts.

Keywords: Refactoring, Software process, Methodologies, Software metrics

1 Introduction

Fowler defines refactoring as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior” [17]. In Agile Methods refactoring is an integral part of the development process; it is adopted to improve continuously the structure and understandability of source code during development. In the agile community it is widely accepted that refactoring contributes to confine the complexity of source code and has a positive impact on the understandability and maintainability of a software system: frequently refactored code is believed to be easier to understand, correct and adjust to new requirements.

A growing number of studies address the relationship between refactoring and the internal structure of source code and its impact on program understanding, software quality, and the evolution of a software design: an excellent overview is given in [25]. Most of these studies focus on the following issues:

- Impact of refactoring on the structure of the source code [6]
- Identification of code smells to locate possible refactorings [7], [15], [28], [32]

- In reverse engineering, how refactoring can reconstruct the overall design of existing systems [12] and improve the quality of legacy code [26]

Mens *et al.* [25] define and discuss different refactoring activities related to the issues mentioned above: In this research we focus on one particular topic, namely the assessment of the effect of refactoring on some quality characteristics that depend or have an impact on software maintainability both from the point of view of the software product and the software process [14]. Only few empirical studies analyze the impact of refactoring on code quality: Demeyer [13] analyzes whether refactoring has a negative impact on program performance; Bois and Mens [6] develop a framework for analyzing the impact of refactoring on internal quality metrics, but they do not provide any experimental validation in an industrial environment. Stroulia and Kapoor [33] perform a case study in an academic environment, where it is shown that size and coupling metrics of a software system decrease after refactoring. Bois *et al.* [7] propose refactoring guidelines for enhancing cohesion and coupling metrics and obtain promising results by applying them on an open source project. Simon *et al.* [32] follow a similar strategy. Sahraoui *et al.* [29] use quality estimation models for analyzing whether some object-oriented metrics can be used for detecting situations where a particular transformation of source code (refactoring) can be applied to improve the quality of a software system. Again, they do not validate their approach within an industrial case study or experiment. Yu *et al.* [35] use a modeling framework for non-functional requirements and relate refactorings to soft goals. They perform a case study, which shows that refactoring can be measured as the transformation on the state of program in the quality space. Tahvildari and Kontogiannis [34] investigate the use of metrics for detecting potential design flaws and for suggesting potentially useful transformations for correcting them. Finally, Kataoka *et al.* [20] provide a quantitative evaluation of maintainability enhancement by refactoring. For the purpose of validation they analyze a project developed by a single developer, but do not provide any information on the development environment. Thus, it is questionable if their findings are valid in a different context where development teams follow a structured process and use common software engineering practices for knowledge sharing.

In the context of Agile Methods there are several claims that refactoring provides four significant advantages [17]:

- Refactoring helps developers to program faster
- Refactoring improves the design of the software
- Refactoring makes software easier to understand
- Refactoring helps developers to find bugs

The first advantage relates to productivity and is probably the most important for managers who are mainly concerned with time to market. Nevertheless, there is almost no solid, empirical, and quantitative evidence of such claim, apart from a small case study, where it appeared that refactoring decreased the long-term productivity [1]. Recently Schofield *et al.* [30] performed a return on investment analysis on an open source project in order to estimate savings in effort, given a specific (beneficial) code change. They found that, most of the time, refactorings have beneficial impacts on maintenance activities, and thus are motivated from an economic perspective.

The last three advantages of refactoring refer to software quality attributes. We have previously mentioned some studies that analyze the impact of code restructuring induced by refactorings on internal product metrics, which are typically used to measure quality attributes, such as complexity, coupling and cohesion. Such early results are promising, still there is a need for (a) additional empirical validation to better understand and generalize the findings, and (b) a clear linkage to external quality attributes, such as number of defects.

Altogether, the real advantages of refactoring are still to be fully assessed [24]. In particular, it is not yet clear whether refactoring increases developer productivity and the extent to which refactoring improves software quality. As regards quality, it appears to be a convergence of positive remarks, still, without solid quantification. Needless to say, a major impediment for a deeper understanding of these issues is a lack of empirical investigation, based on hard data coming from industry.

The paper is organized as follows. In Section 2, we describe our research methodology and the experimental set-up. In Section 3, we present a case study and discuss the results obtained from it; in Section 4, we discuss the limitations of our approach. Finally, conclusions and implications of the investigation are drawn in Section 5.

2 Research Methodology and Experimental Set-up

In order to investigate a research problem we have to define (a) the objectives and hypotheses of the study, (b) the variables along with the metrics used to measure them, (c) the instruments used in the experiment and the data collection procedure, and (d) the data analysis method. We will discuss each of these points below. The results of the case study and threats to the validity of the experiment are presented in subsequent sections.

2.1 Research Hypotheses

Software is naturally subjected to continuing change, increasing size and complexity and therefore declining maintainability. In particular, in the one-way traditional development process, internal code measures tend to show a continuous increase in complexity and coupling and a decrease in cohesion as new features are added to a software system. This natural process of code corrosion is even more manifest as time goes by [21]. More complex and intertwined code is more difficult to manage and maintain; therefore, we expect that also development productivity will show a decreasing trend over time. In contrast, in XP-like processes, thanks to its agile practices (in particular constant refactoring, unit testing, frequent releases), the complexity of the code and the effort for adding new functionalities is claimed to remain about constant or to grow very slowly [3]. Unfortunately, due to high costs of industrial software development we are not able to run a formal experiment with an industrial partner where we could analyze two similar projects, one developed using XP practices and one without, and compare directly the evolution of respective quality and productivity metrics.

We have to content ourselves with a simpler approach: We focus only on one XP practice, namely refactoring, and compare changes of productivity before and after explicit refactorings and use such comparison as criteria for assessing the impact of refactoring on it. As regards quality and maintainability, we determine the changes of several design metrics after an *explicit refactoring* has been applied and compare changes with the average daily changes per iteration. If they are significantly different (improved) we then conclude that refactoring has a positive effect on code quality and, as a consequence, on software maintainability. We define in section 2.2 what we intend by *explicit refactorings* in the context of our study.

Framed in terms of research questions, we aim at presenting evidence that will allow us to reject (or accept) the following two null hypotheses:

- H^0_A: after an *explicit refactoring* the average productivity for the consecutive development iteration is the same as for the previous iteration.
- H^0_B: the considered internal quality metrics (complexity, coupling, and cohesion) do not show any improvement after an *explicit refactoring* with respect to their average daily changes.

In order to obtain more reliable and smoother results we do not simply compare the changes of productivity and quality metrics before and after the application of a refactoring. Such changes could happen by chance or because of some other factors we do not control within this case study (for example mood of developers, work on particular part of the code, problems with tools, other XP practices). To minimize the influence of random and uncontrolled changes we compare average productivities between development iterations (the one in which an explicit refactoring has been applied with the following iteration). Also for the quality metrics we compute their average daily changes and compare them with the changes induced by an *explicit refactoring*.

2.2 Explicit Refactorings, Productivity, and Quality

In more traditional development processes, refactoring is present in ordinary maintenance tasks or extraordinary maintenance projects, in order to improve software maintainability [20]. The context of our analysis however is an agile development process, namely a tailored version of Extreme Programming [2]; in such environment refactoring is an integral part of software development. Kent Beck illustrates the principle of agile development with the two hats metaphor: One is adding new functionality (coding) and the other is refactoring. The developer should swap frequently between these two hats but wear only one at a time. Therefore, we assume that developers apply small refactorings like Extract Method, Rename, Simplify Conditional, Move Method/Field, and so on [17] throughout development – without even documenting it. We believe that all these small refactorings improve slightly the quality of the code and increase overall development productivity compared to a development process, which does not use the practice of refactoring.

However, due to the lack of empirical data (of two comparable software projects, one developed using an agile and one using a traditional method) such comparison is out of scope of this research. Instead, we analyze the effect of *explicit refactorings* on productivity and quality within the same project. *Explicit refactoring* means that developers wrote *explicitly* a user story for refactoring tasks and that the implementation of such user story took a considerable amount of time – in our case even several hours.

For the time being, we do not identify different kinds of refactorings and analyze separately the impact of each type of refactoring on productivity or quality. After having defined what we intend by *explicit refactoring* we have to define the other variables of interest for this research, namely development productivity and metrics for software quality and in particular maintainability.

Lots of work has been done on how to measure developers' productivity [16]. However, no definitive measure has been found and perhaps such definite measure does not exist. A very simple measure of productivity is the ratio of lines of code (LOC) produced and effort in hours spent in producing them:

$$productivity = \frac{LOC}{Effort}$$

In this research we use this equation because of its simplicity and expressiveness. In addition, programmers are all working in good faith – they volunteered for this experiment, the effort spent in activities other than coding has been closely monitored and evenly distributed, code reuse has been closely scrutinized also via the CVS repository, and no code generators have been used.

Software quality is a composite property of many internal and external software attributes. There has been a lot of discussion on the meaning of software quality [23], [5]. It is now commonly agreed [16] that software quality is a property defined by several small-scaled and directly measurable attributes. In this research we use complexity, coupling, and cohesion metrics, as defined by Chidamber and Kemerer (CK) [10]; such measures are widely accepted both by practitioners and researchers and validated by several previous studies [4], [9]. In addition, such measures are easy to collect and to understand, a precondition for their effective use [19].

Software maintainability is related both to software quality (it is considered as a quality factor) and cost, as good maintainability of software reduces significantly maintainance effort [11]. An XP project is constantly in the state of maintainance [3], therefore, besides quality measures also evolution of development productivity is a good indicator for its maintainability. The CK metrics include measures for complexity (WMC) and coupling (CBO) of object-oriented systems: Both of them are related to software maintainability as an increase of software complexity and coupling deteriorates its understandability [18].

2.3 Data Collection

The software project we analyze was developed using an agile, XP-like methodology tailored by Abrahamsson *et al.* [2]. Therefore, data collection had to be **(a)** non-invasive to preserve the agile nature of the project itself [27], and **(b)** accurate and reliable for doing meaningful statistics.

In order to achieve these two goals we use the PROM tool [31] for collecting product and effort metrics. PROM is a fully automated measurement framework for software engineering processes and products. Source code metrics are extracted daily from the source code management system employed by the company. PROM enables the automatic collection of the effort associated with different tasks such as reading documents, browsing the web and coding. In particular, a plug-in for the IDE in place collects the time spent by developers for coding activities for individual methods and classes. Effort data for coding is collected as soon as the developer enters the cursor in the source code editor of the IDE and ends if the editor is off focus, the IDE is closed or the screensaver is activated. Moreover, PROM allows the user to specify if one or two programmers are sitting in front of a machine.

The notion of effort adopted in this context is strongly related to only coding activities and does not include the time spent discussing about the design/code on a whiteboard; however in an XP-like process, which itself assigns to coding activities the highest importance, this measure is a reasonable measure for development effort. Both source code metrics and effort data are integrated and stored automatically in a central database, from which we access the data for our analysis.

To collect the product and process metrics listed in Table 1 with the PROM tool, we adopt the following data collection procedure:

- Every day at midnight the source code metrics are extracted from a CVS repository.
- A plug-in for Eclipse (the IDE used by developers) collects automatically the time spent for coding on individual classes and methods.
- We identify the days on which explicit refactorings are applied from the user stories (described in the project plan).

Table 1 provides an overview of the information that come from PROM and is used in this research.

Table 1. Sample data collected by PROM and aggregated at a system level. All metrics are per day.

Day	**LOC**	**CK metrics**	**Effort (hour)**	**Productivity (LOC/hour)**
22	150	30, 7, 5, 3	4.24 h	35.3
...	…	...	...	…

We aggregate metrics at a system level (we add up all single classes) and compute their overall changes per day in the case of product metrics and the total time spent for coding per day in the case of effort. The way we aggregate metrics is a first approach and could be refined: We could for example identify the classes affected by refactoring using a technique presented in [12], [30] and use only them for analyzing changes in quality and productivity. Whether this would change our findings, has to be assessed in a future analysis.

2.5 Data Analysis Method

Our research design is to some extent a one-factor, repeated-measures design: The treatment (in our case refactoring) is applied twice to the same subjects. We use box-plots for comparing the means of different populations (before-after refactoring productivity). In addition we perform a Wilcoxon rank sum test [22], as we cannot assume a normal distribution and homogeneity of variance of data.

As regards the quality metrics we proceed in the following way: first, we compute their changes at the end of a day when developers applied an explicit refactoring with respect to the previous day. Then, we use a Wilcoxon Signed-Rank [22] test to conclude whether these changes are lower than the average daily changes per iteration or not. Our final goal is to disprove the null hypotheses by using the Wilcoxon Signed-Rank tests to determine (a) if the development productivity is higher after refactoring than before, and (b) if quality metrics are significantly improved by refactoring with respect to their average changes.

3 Case Study

In the following section, first we describe the context of the case study; afterwards, we present and discuss the results of our analysis.

3.1 Context of the Case Study

The object under study is a software project in an agile, close-to-industrial development environment ("close-to-industrial" refers to an environment where the development team is composed of both professional software engineers and students [2]). The result is a commercial software product developed at VTT in Oulu, Finland, to monitor applications for mobile, Java enabled devices. The programming language was Java (version 1.4) and the IDE was Eclipse 3.0. The project was a full business success in the sense that it delivered on time and on budget the required product.

Four developers formed the development team. Three developers had an education equivalent to a BSc and limited industrial experience. The fourth developer was an experienced industrial software engineer.

The development process followed a tailored version of the Extreme Programming practices [2], which included all the practices of XP but the "System Metaphor" and the "On-site Customer"; there was instead a local, on-site manager that met daily with the group and had daily conversations with the off-site customer. In particular, the team worked in a collocated environment and used the practice of pair programming. The project lasted eight weeks and was divided into five iterations, starting with a 1-week iteration, which was followed by three 2-weeks iterations, with the project concluding in a final 1-week iteration. Throughout the project, mentoring was provided on XP and other programming issues according to the XP approach. Since the team was exposed for the first time to an XP-like process, a brief training of target XP practices was given before the start of the project.

The total development effort per developer was about 192 hours (6 hours per day for 32 days). Since with PROM we monitored all the interactions of the developer with different applications, we are able to differentiate between coding and other activities: About 75% of the total development effort was spent for pure coding activities inside the IDE while the remaining 25% was spent for other assignments like working on text documents, reading and writing emails, browsing the web and similar tasks. The developed software consists of 30 Java classes and a total of about 1770 Java source code statements (LOC counted as number of semicolons in a Java program).

During development two user stories have been explicitly written for refactoring activities: One at the end of iteration two with the title "Refactor Static Classes to Object Classes" and one at the end of iteration four with the title "Refactor Architecture". We refer to the implementation of these two user stories as explicit refactorings; we analyze changes of productivity and quality measures before and after their completion.

3.2 H^0_A – Does Productivity Increase After "*explicit refactorings*"?

Figure 1 shows the evolution of the average productivity per iteration over the whole development period.

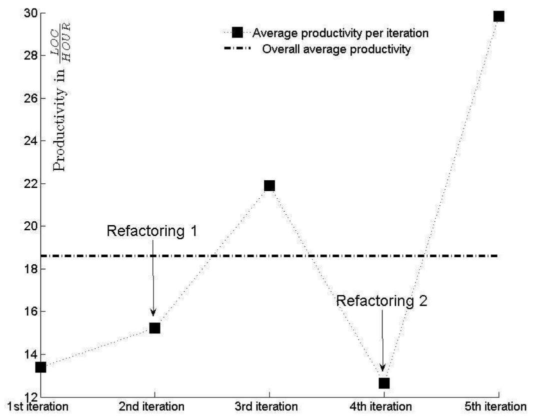

Fig. 1. Average development productivity per iteration.

The productivity is almost the same in iterations 1, 2, and 4, (about 15 LOC/HOUR) while it is significantly higher in iterations 3 and 5 (more than 22 LOC/HOUR). This distribution is interesting for two reasons: First, productivity does not show a decreasing trend during software development as we were expecting due to higher effort for adding new functionality as the system's complexity and coupling is growing. Second, whenever developers perform an *explicit refactoring* – i.e. at the end of iteration 2 and at the end of iteration 4 – productivity of the following iteration is significantly higher than average productivity of the remaining iterations. For the data under scrutiny we can only measure changes of productivity after two *explicit refactorings* (treatments). With such small sample size statistical tests are hardly applicable as significance values are rather meaningless. Instead we prefer to use a box-plot for visualizing the difference in productivity in the before-after refactoring situations.

Figure 2 shows a box-plot of the average productivity per iteration throughout development. Moreover, we draw two dashed lines, one indicating the average productivity for the iteration following the first *explicit refactoring*, the other for the iteration following the second *explicit refactoring*. We can observe a clear improvement of productivity for both cases with respect to average productivity.

For the sake of completeness we perform a Wilcoxon rank sum test to compare productivity after the 2 refactorings with average productivity of the remaining iterations. As expected, given our small sample size, we obtain a p value of 0.2 meaning that we cannot reject H^0_A neither for refactoring 1 nor for refactoring 2. Overall, we can conclude that the productivity data sustain the claim that refactoring raises development productivity in the short-term, thus nullifying to some extent the complexity naturally added during development. However, this conclusion is more a confirmation of a suspicion and not a clear affirmation based on statistical inference from experimental data.

In order to consider the overall evolution of productivity throughout development, we compare the medians of the daily productivity of each iteration using a non-parametric Kruskal-Wallis test [22]. The result is that they are not statistically different from each other. In fact Figure 1 emphasizes that productivity is rather increasing than declining towards the end of the project.

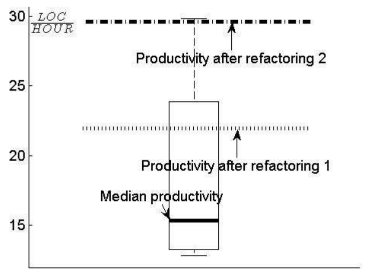

Fig. 2. Box-plot of average productivity per iteration.

Altogether, our findings strongly advocate that refactoring of a software system raises subsequent development productivity and prevents in a long-term its deterioration.

3.3 H^0_B — Cohesion, Coupling and Complexity: Does Refactoring Improve Code Quality?

Findings of prior studies claim that refactoring improves some low-level quality metrics like coupling and cohesion measures [7]. In this research we look at the temporal evolution of the CBO, WMC, RFC, and LCOM metrics and how it is related to refactoring. A visual inspection of the evolution of these metrics (Figure 3) evidences that their changes, from one iteration to the next, tend to decrease starting from the second iteration (1st *explicit refactoring*) for the CBO and RFC metrics, and from the third for the LCOM and WMC metrics. This is a first indication that refactoring could limit the overall decrease of cohesion and increase of coupling and complexity metrics that we expect to occur during software development.

Table 2. p-values for the one-sided Wilcoxon Signed-Rank test for testing if the population mean of the median of the daily changes per iteration of CK metrics is higher than the changes after refactoring.

	WMC	LCOM	CBO	RFC
Refactoring1	0.72	0.5	**0.03**	0.18
Refactoring2	**0.03**	**0.02**	**0.02**	**0.02**

Table 2 gives the p-values (significant values at the 0.05 level are set in bold face) of the Wilcoxon Signed-Rank test for assessing whether or not the changes of the 4 CK metrics after the two *explicit refactorings* are the same with respect to their average changes: We can see that all of them improve after the second refactoring, since their changes are significantly lower (they are in fact negative) than the average of their daily changes. For the first refactoring this is only true for the coupling metric CBO. The results are not strong enough to reject H^0_B for both refactorings, but only for the second and in part for the first. Still, they provide confidence that with more comprehensive experimentation on larger projects it will be possible to significantly prove it.

Visually inspecting the plot (Figure 3) of the changes of LCOM, CBO, RFC and WMC per iteration, we also notice an interesting phenomenon: After an initial phase of remarkable growth of these metrics, they start to decrease, most likely thanks to refactoring. We interpret this as the people gathering a more comprehensive view of the application to develop, and thus being able to better refactor the system, creating simpler, less coupled, and more cohesive code. Moreover, by refactoring the system they acquire a better understanding of the program being developed, which could explain a boost in productivity (this observation is consistent with the findings of other researchers [8]). Yet, this is an interpretation based on a visual inspection rather than on a statistical test: only future research involving larger data samples will be able to assess its statistical significance and validity.

Altogether, this research evidences that there are visual indicators (in part supported by statistical tests) that refactoring prevents an explosion of complexity and coupling metrics by driving developers to simpler design and as a consequence less complex and coupled and easier to maintain code.

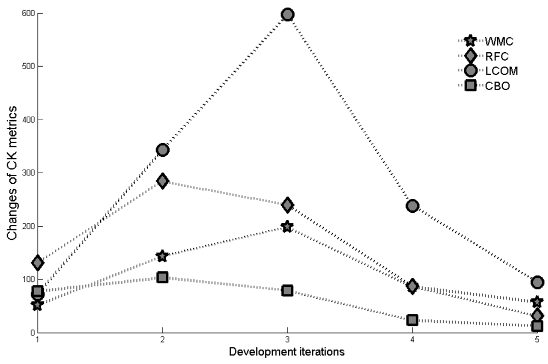

Fig. 3. Evolution of the average changes of LCOM, CBO, RFC, and WMC per iteration.

4 Threats to Validity

This research aims at assessing the impact of refactoring on development productivity and software quality measures. The results of this research are particularly interesting as they come from a case study in a close-to-industry context: A situation, which is quite rare in software engineering. However, there are a number of threats to construct, internal and external validity of the study that have to be addressed properly. Those are in particular:

(a) First, the conclusions we draw depend strongly on the definition we give of productivity and software quality and its validity in industry. We define productivity as the ratio between source code statements and time spent for coding. Several objections have been raised against this measure: A malicious developer could artificially inflate the number of lines of code; only coding is considered ignoring all the other phases of development – analysis, design, etc; code reuse and automatically generated code are not taken properly into account; and other. Despite all the criticism, this equation is by far the most used in industry, as it is very easy to understand and gives clear and absolute numbers, which are easy to compare and to use in statistical calculations. Moreover, in the context of XP much emphasis is put on coding activities; thus, development effort coincides mostly with coding effort. In order to support the validity of the productivity measure used in this study it would be interesting to run similar studies using this definition but also a definition based on other parameters, for instance function points or user stories.

(b) As regarding to internal validity we have to be aware that with a single case study it is not possible to infer whether or not the observed relation is a causal one. We are not able to control and manipulate the influence of other confounding factors such as short release cycles or pair programming. For example, the observed increase in productivity after explicit refactorings could be explained differently: Maybe in the iterations following an *explicit refactoring* developers implemented “easy” user stories or did not do any refactoring at all (refactoring itself decreases to some extent productivity as measured in this study). Moreover, even if we were sure that refactoring is the cause for the observed improvements in productivity due to the small sample size such relation is of low statistical significance. However, confounding factors, which we identified in the context of this study, are averaged over iterations and should impact productivity and quality measures equally (i.e. independent of specific iterations). Therefore, we are confident that the observed effects are due to the explicit refactorings and that a larger study would provide necessary statistical significance, which is only suggested by our results.

(c) We do not consider different kinds of refactorings. Such coarse grained analysis could bias our results: Developers may for example apply only a limited subset of refactorings – due to their inexperience or other reasons – and in such case we can probably not generalize the implications for all other types of refactorings. We plan to take into account different categories of refactorings in a more refined, future study.

(d) We sum averaged quality metrics and productivity over all classes, whereas probably only a few of them have been affected by the two explicit refactorings. In doing so we could misinterpret the real impact of refactoring; we plan in a future work with a larger sample size to analyze the changes of productivity and code quality only for the classes that have been involved directly in a refactoring activity.

(e) The subjects of the case study are heterogeneous (three students and one professional engineer) and use for the first time an XP-like methodology. This could confound our findings, as for example students may behave very different from industrial developers. Moreover, also a learning effect could be visible and for example be the cause for the evolution of the productivity and quality metrics as shown, respectively, in Figure 1 and Figure 3. Developers were aware that they are monitored, but did not know that we measured in particular productivity before and after refactorings; we did not communicate them the objectives of the study as such knowledge could influence their behavior leading to higher productivities after refactorings.

(f) As with every case study, it is hard to generalize to other, larger contexts. We think that our findings are applicable to small teams working in similar, highly volatile domains (ours is application development for mobile devices). However, additional research is needed to ensure that this is indeed true and to generalize it to other contexts.

Furthermore, it would be interesting to analyze how much refactoring is “good enough” to keep productivity high and what kinds of refactorings are important to improve both productivity and quality.

5 Conclusions

Although agile processes and practices are gaining more importance in the software industry there is limited solid empirical evidence of their effectiveness. This research focuses in particular on the practice of refactoring, which is one of the key practices of Extreme Programming and other Agile Methods.

While the majority of software developers and researchers agree that refactoring has long-term benefits on the quality of a software product (in particular on program understanding) there is no such consensus regarding the development productivity. Available empirical results regarding this issue are very limited and not clear [1]. This might refrain managers from adopting refactoring, as they might be scared of loosing resources.

This work contributes to a better understanding of the effects of refactoring both on code quality – in particular on software maintainability - and development productivity in a close-to industrial, agile development environment. It provides new empirical, industrially based evidence that refactoring rather increases than decreases development productivity and improves quality factors, as measured using common internal quality attributes – reduces code complexity and coupling; increases cohesion. The implications on defects are not discussed, as such data are not available. Moreover, we do not contribute in exploring the linkage of refactoring to other external quality attributes. Clearly, this question has to be addressed in a future study.

As regards productivity, these results are in contradiction with the previous work of Abrahamsson and Koskela [1]. However, such older work addressed a case that was too limited to be taken as a reference. For internal quality metrics, our results are in accordance with the existing literature. Altogether, we believe that our findings are particularly relevant, as this work is a case study in a close-to-industry environment, a kind of empirical investigation that is rare for the research problem we discuss here. Clearly, this is a first work in the area. A real, generalizable assessment of the implications of refactoring requires several repetitions of studies like this, possibly also including data on defects.

The findings of this research have major implications for a widespread use of refactoring, as already mentioned by Beck in his first work on XP [3]. Of course, refactoring as any other technique is something a developer has to learn. First, managers have to be convinced that refactoring is very valuable for their business; this research should help them in doing so as it sustains that refactoring – if applied properly – intrinsically improves code maintainability and increases development productivity. Afterwards, they have to provide training and support to change their development process into a new one that includes continuous refactoring.

Case studies in close-to-industry contexts are very rare in software engineering and this gives us a remarkable confidence on the results that we have obtained. However, it is important to remember that, formally, such results are only valid in the specific context of the study. To achieve a high level of confidence of them, it is essential to replicate such case studies, also in other contexts and using different measures.

Acknowledgments. Special thanks go to development team at VTT, Oulu, Finland, which has been disposed to install and use the PROM tool for data collection and in the end, enabled this case study. The authors would also like to acknowledge the support by the Italian Ministry of Education, University and Research via the FIRB Project MAPS (http://www.agilexp.org) and the autonomous province of South Tyrol via the Interreg Project Software District (http://www.caso-synergies.org).

References

1. Abrahamsson, P., Koskela, J.: Extreme programming: Empirical results from a controlled case study. In: ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2004), Redondo Beach CA, USA (2004)
2. Abrahamsson, P., Hanhineva, A., Hulkko, H., Ihme, T., Jäälinoja, J., Korkala, M., Koskela, J., Kyllönen, P., and Salo, O.: Mobile-D: An Agile Approach for Mobile Application Development. In: Proceedings 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'04, Vancouver, British Columbia, Canada (2004)
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
4. Basili, V.R., Briand, L.C., and Melo, W.L.A.: Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering, 22(10): 267-271, October (1996)
5. Boehm, B.W., Brown, Kaspar, J.R., et al.: Characteristics of Software Quality. TRW Series of Software Technology, Amsterdam, North Holland (1978)
6. Bois, B.D., Mens, T.: Describing the impact of refactoring on internal program quality. In: Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), Amsterdam, The Netherlands (2003)
7. Bois, B.D., Demeyer, S., Verelst, J.: Refactoring – Improving Coupling and Cohesion of Existing Code. In: Belgian Symposium on Software Restructuring, Gent, Belgium (2005)
8. Bois, B.D., Demeyer, S., and Verelst, J.: Does the "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension?. In: Proceedings 9th European Conference on Software Maintenance and Reengineering (CSMR 2005), Manchester, UK, 21-23 March (2005)
9. Briand, L.C., Wüst, J.: Modeling Development Effort in Object-Oriented Systems Using Design Properties. IEEE Transactions on Software Engineering, 27(11): 963-986, November (2001)
10. Chidamber, S., Kemerer, C.F.: A metrics suite for object-oriented design. IEEE Transactions on Software Engineering, 20(6): 476-493, June (1994)
11. Corbi, T.A.: Program Understanding: Challenge for the 1990s. IBM Systems Journal, 28(2): 294-306 (1989)
12. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding Refactorings via Change Metrics. In: Proceedings of the 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'00, Minneapolis, USA (2000)
13. Demeyer, S.: Maintainability versus Performance: What's the Effect of Introducing Polymorphism?. Technical report, Lab. on Reeng., Universiteit Antwerpen, Belgium (2002)
14. Van Deursen, A.: Program Comprehension Risks and Opportunities in Extreme Programming. In: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany, 2-5 October (2001)
15. Van Emden, E., and Moonen, L.: Java Quality Assurance by Detecting Code Smells. In: Proceedings of the 9th Working Conference on Reverse Engineering. IEEE Computer Society Press (2002)
16. Fenton, N., Pfleeger, S.L.: Software Metrics A Rigorous & Practical Approach. PWS Publishing Company, Boston (1997)
17. Fowler, M.: Refactoring Improving the Design of Existing Code. Addison-Wesley (2000)
18. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. p. 62, Prentice-Hall PTR, Upper Saddle River, New Jersey, USA (1996)
19. Johnson, P.M., Disney, A.M.: Investigating Data Quality Problems in the PSP. In: Proceedings of Sixth International Symposium on the Foundations of Software Engineering (SIGSOFT 98) (1998)

20. Kataoka, Y., Imai, T., Andou, H., and Fukaya, T.: A Quantitative Evaluation of Maintainability Enhancement by Refactoring. In: Proc. Int'l Conf. Software Maintenance, pp. 576-585, October (2002)
21. Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, P. E., and Turski, W.M.: Metrics and laws of software evolution-the nineties view. In: Proceedings of the Fourth International Software Metrics Symposium, 5-7 November (1997)
22. Lehmann, E.L.: Testing Statistical Hypotheses. Springer-Verlag, Inc., New York (1986)
23. McCall, J.A., Richards, P.K., and Walters, G.F.: Factors in Software Quality. RADC TR-77-369, Vols I, II, III, US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055 (1977)
24. Mens, T., Demeyer, S., Bois, B.D., Stenten, H., van Gorp, P.: Refactoring: Current Research and Future Trends. Electronic Notes in Theoretical Computer Science, 82(3) (2003)
25. Mens, T., and Tourwé, T.A.: Survey of Software Refactoring. IEEE Transactions on Software Engineering, 30(2): 126-139, February (2004)
26. Pizka, M.: Straightening spaghetti-code with refactoring?. In: Proceedings of the Int. Conf. on Software Engineering Research and Practice - SERP, pages 846- 852, Las Vegas, NV (2004)
27. Poppendieck, T., Poppendieck, M.: Lean Software Development: An Agile Toolkit for Software Development Managers. Addison-Wesley (2003)
28. Ratzinger, J., Fischer, M., Gall, H.: Improving Evolvability through Refactoring. In: Proceedings 2nd International Workshop on Mining Software Repositories, MSR'05, Saint Louis, Missouri, USA (2005)
29. Sahraoui, H.A., Godin, R., and Miceli, T.: Can metrics help to bridge the gap between the improvement of oo design quality and its automation?. In: Proc. International Conference on Software Maintenance, pages 154–162, October (2000)
30. Schofield, C., Tansey, B., Xing, Z., Stroulia, E.: Digging the Development Dust for Refactorings. In: Proceedings of the 14th International Conference on Program Comprehension (ICPC'06), Athens, Greece (2006)
31. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data. In: Proceedings of the EUROMICRO 2003, Belek-Antalya, Turkey (2003)
32. Simon, F., Steinbruckner, F., and Lewerentz, C.: Metrics based refactoring. In: Proc. European Conf. Software Maintenance and Reengineering, pp. 30—38, IEEE Computer Society Press, (2001)
33. Stroulia, E., Kapoor, R.V.: Metrics of Refactoring-based Development: An Experience Report. In: The 7th International Conference on Object-Oriented Information Systems, pp. 113-122, Calgary, AB, Canada, Springer Verlag (2001)
34. Tahvildari, L., and Kontogiannis, K.A.: Metric-Based Approach to Enhance Design Quality through Meta-Pattern Transformations. In: Proc. European Conf. Software Maintenance and Reeng., pp. 183-192 (2003)
35. Yu, Y., Mylopoulos, J., Yu, E., Leite, J.C., Liu, L., D'Hollander, E.H.: Software refactoring guided by multiple soft-goals. In: Proceedings of the 1st workshop on Refactoring: Achievements, Challenges, and Effects, in conjunction with the 10th WCRE conference 2003, pp. 7-11, Victoria, Canada, November 13-16 (2003)