

Does Refactoring Improve Reusability?

Raimund Moser¹, Alberto Sillitti¹, Pekka Abrahamsson²,
and Giancarlo Succi¹

¹ Center for Applied Software Engineering, Free University of Bolzano-Bozen,
Piazza Domenicani 3, Italy
{rmoser, asillitti, gsucci}@unibz.it

² VTT Electronics, Oulu, Finland
pekka.abrahamsson@vtt.fi

Abstract. The improvement of the software development process through the development and utilization of high quality and reusable software components has been advocated for a long time. Agile Methods promote some interesting practices, in particular the practice of refactoring, which are supposed to improve understandability and maintainability of source code. In this research we analyze if refactoring promotes ad-hoc reuse of object-oriented classes by improving internal quality metrics. We conduct a case study in a close-to industrial, agile environment in order to analyze the impact of refactoring on internal quality metrics of source code. Our findings sustain the hypothesis that refactoring enhances quality and reusability of – otherwise hard to reuse - classes in an agile development environment. Given such promising results, additional experimentation is required to validate and generalize the results of this work.

1 Introduction

In Extreme Programming (XP) much emphasis is given on an agile, iterative and customer oriented way of how to develop software. Among the top priorities of XP are (a) customer satisfaction through continuous delivery of valuable software and (b) embracing changing requirements (<http://agilemanifesto.org>). The practices of XP are tailored to achieve such goals: iterative and informal planning, simple design, continuous refactoring of the code, pair programming, test first and continuous integration – just to mention a few [3]. Most of these practices are intended to be used during development and maintenance and seem to keep at least in part their promises [21].

However, XP does not address explicitly the issue of software reuse as one of its practices. This may wonder since many believe that software reuse provides “the key to enormous savings and benefits in software development” [24], [13]. XP *per se* does not aim at developing software for possible future reuse in order to avoid overhead during development. Keep it simple and develop only what the customer really wants is one of the key principles of XP.

On the other hand we think that XP – compared to more traditional development methodologies - intrinsically guides software engineers to develop software, which is of high quality and therefore suited for ad-hoc reuse. In particular the practice of continuous refactoring may improve internal quality metrics and affect reusability of a software system in a positive way [7]. Reusability is a rather high-level quality metric

that consists of several internal and external properties of the software and of the development process [15]. Poulin [24] introduces a possible taxonomy of reusability metrics: He differentiates between empirical and qualitative methods and gives an overview of several models to assess reusability. Dandashi and Rine [11] decompose reusability into direct and indirect quality attributes, namely: adaptability, completeness, maintainability, and understandability. Refactoring seems to enhance particularly the last two quality attributes [5], [25]: In his book on refactoring [16] Fowler stresses over and over that one of the key points of refactoring is to make the code easier to understand and to read. We believe this should affect in a positive way also ad-hoc reuse of otherwise not reusable pieces of software.

A proven way to measure reusability is to analyze how often parts (classes, methods, etc.) of a software system are reused in a product. Such reusability frequency and amount of reuse metrics are proposed and used by several researchers [13], [10], [17]. In this research we do not follow this approach, as we do not have such data. Our idea is to analyze whether refactoring enhances reusability by analyzing its impact on some of the internal reusability metrics proposed and validated in the work of Dandashi *et al.* [11]. Thus, in order to assess reusability we employ only a restricted set of internal product attributes that are available and monitored during development. We do not take into account any external and high-level product or process metrics. In doing so we risk failing in characterizing properly software reusability. However, we think that our approach is in part justified by the findings of other researchers, in particular by [8], [2], and [11].

The paper is organized as follows: In Section 2 we present our research methodology and define the research question; in Section 3 the results of a case study are presented and discussed; in Section 4 we refer to some issues regarding the limitations of our approach and future plans. Finally, conclusions and implications of the investigation are drawn in Section 5.

2 A Methodology for Evaluating the Impact of Refactoring on Code Quality

In this section at first we describe the metrics we use for assessing code quality - with particular focus on reusability. Afterwards, we develop a model for evaluating how refactoring may affect internal quality metrics, which are used to assess reusability of the software system during development. Finally we state our research questions.

2.1 Internal Quality Metrics That Affect Reusability

Our research question is to assess whether refactoring facilitates the development of high-quality code and as a consequence ad-hoc reusability or not. To answer such question first we have to define carefully what we mean by high-quality and reusability and how we measure it. Code quality is a rather vague term for describing certain quality attributes of a software system and can be decomposed in different ways into lower-level metrics [15]. In this research we focus on internal properties of the software that are considered to be relevant for its reusability. In particular, we follow the approach proposed by Dandashi and Rine [11] and use two different sets of metrics:

- One for micro level measurements (measurements at a method level)
- And one for macro level measurements (measurements at a class level)

For the micro level measurements we employ McCabe's cyclomatic complexity of a method [23] and the number of Java statements per method. To arrive at a measure for the whole class, the highest measure is used as a representative measure of the corresponding class measure. For the macro level measurements we use the Chidamber and Kemerer (CK) set of object-oriented metrics [8].

The motivation for choosing this set of metrics is twofold: First, some of them such as the CK metrics are among the best-understood and validated metrics for object-oriented systems and therefore we can be more confident in their expressiveness [2]. Second, the tool we use for collecting these metrics is able to collect them in an automatic and non-invasive way - a fundamental requirement for data collection in an XP process [19].

Several empirical studies put the CK metrics into relationship with software quality, in particular with maintainability, reusability, and reliability. Li and Henry [22] for example show that the CK metrics are useful to predict maintainability. Basili *et al.* [2] investigate the relationship between the CK metrics and code quality: Their findings suggest that 5 of the 6 CK metrics are useful quality indicators. However, such studies are rare in XP-like environments and they do not analyze how the evolution of the CK metrics during development is affected by refactoring. Table 1 summarizes the metrics we use in this research as indicators for reusability.

Table 1. Selected internal product metrics as indicators for reusability

Metric name	Level	Definition
MAX_LOC	Class	Maximum number of Java statements of all methods in a class
MAX_MCC	Class	Highest McCabe's cyclomatic complexity of all methods in a class
CBO	Class	Coupling Between Object classes (CK)
LCOM	Class	Lack of Cohesion in Methods (CK)
WMC	Class	Weighted Methods per Class (CK)
RFC	Class	Response Of a Class (CK)
DIT	Class	Depth of Inheritance Tree
NOC	Class	Number Of Children

Dandashi and Rine [11] find in their research a correlation between the CK and complexity measures and external high-level quality attributes for reusability. In this work we rely on their findings, as we do not analyze directly the impact of refactoring on external reusability measures. However, we are well aware that we conduct a study in a very different environment; the fact that we consider only a restricted set of internal product metrics for assessing reusability limits to some extent the validity of our findings and has to be addressed in a future study.

2.2 An Approach to Assess the Impact of Refactoring on Internal Quality Metrics During Development

In section 2.1 we choose a set of metrics, which is known to be useful as internal measures for quality of a software system. However, we do not know a priori the range of values of these metrics that would indicate good or bad quality. Analyzing historical data or several similar projects can only – if at all – derive such thresholds [4]. We follow a different strategy, as we do not seek to associate absolute values of the metrics in Table 1 to different classes of quality, but rather analyze the changes of them during development.

Our approach is the following: First, we identify a set of candidate classes that are likely to be considered for reuse. Afterwards, we monitor the daily changes of our quality metrics for each class during development. Finally we compare the average of these daily changes with the change each class gains after it has been refactored. This allows us to quantify the impact of refactoring on internal quality metrics compared to their overall evolution during development.

A bit more formally we can define our method as follows.

Let $M_i \in M = \{MAX_MCC, MAX_LOC, CBO, RFC, WMC, DIT, NOC, LCOM\}$ be one of the quality metrics listed in Table 1. In a first step we average their daily changes for each candidate class over the whole development period not including days when the class has been refactored. We denote this average value for metric M_i by ΔM_i . N in equation (1) is the total number of development days; Δt is a time interval of 1 day and R is the set of all days during which developers have refactored a particular class.

$$\Delta M_i = \frac{\sum_{k \in R}^N M_i(k \cdot \Delta t) - M_i((k-1) \cdot \Delta t)}{N - |R|} \quad (1)$$

By ΔR_i we denote the average of the daily changes of quality metric M_i only for the days ($k \in R$) in which a class has been refactored. To assess whether refactoring improves quality of a class we compute its ΔR_i and ΔM_i values and compare them with each other: If ΔR_i is negative and significantly lower than ΔM_i we may conclude that refactoring improves quality metric M_i compared to its standard evolution during development.

In order to apply our method to a real system we not only need to collect the daily evolution of source code metrics but also to identify a set of candidate classes and refactoring activities. Regarding the first issue we proceed as follows: We analyze the design document and use the description provided by developers and our own experience to find classes, which are either explicitly developed for reuse or at least are promising to be reused in the same or similar products. We exclude any classes that are highly dependent on the specific application such as classes dealing with the user interface, product specific data representation/processing or classes holding hard coded data (constants). The identification of candidate classes is a subjective process and therefore we may not identify all relevant classes. However, in an XP process development is not targeted specifically to reusability and in principle every class that is not tightened too much with a particular application feature could be reused in an ad-hoc manner.

The second issue we have to address is: How can we identify days in which a class has been refactored? Currently we are working on a method that extracts such information automatically from a CVS repository by using source code change metrics information (for the basic idea see [12]). This work is still in an early phase and cannot be used for this research. However, for the case study we present in section 3 developers have created user stories for refactoring activities and by analyzing them we know which classes have been refactored when.

To summarize our method for assessing the impact of refactoring on quality and reusability we stress again that it has to be taken with a grain of salt, as we do not include many important factors such as experience of developers, development tools, or the stability of the application domain. However, we think that by analyzing the change of important internal quality metrics induced by refactoring we can indicate whether refactoring - by delivering easy to reuse and maintain code - supports ad-hoc reuse or not.

2.3 Research Question

The goal of this research is to determine whether refactoring improves code quality and as a consequence supports ad-hoc reuse. Our objective is to present evidence that will allow us to reject the null hypothesis:

- H_0 : The changes of quality metric M_i induced by refactoring (ΔR_i) are not different from the average changes during development (ΔM_i) for classes that are likely to be reused in an ad-hoc manner.

And to accept the alternative hypothesis:

- H_1 : The changes of quality metrics M_i induced by refactoring (ΔR_i) are different (preferably lower) from the average changes during development (ΔM_i) for classes that are likely to be reused in an ad-hoc manner.

In section 3 we present a case study we run in order to reject or accept the null hypothesis stated above.

3 Case Study

In this section we present a case study we conducted in a close-to industrial environment in order to analyze quality enhancement and promotion of ad-hoc reuse by refactoring in a software project developed using an agile, XP-like methodology [1]. The objective of the case study is to answer our research question posed in section 2: First, we collected in a non-invasive way the metrics listed in Table 1; afterwards, we analyzed their time evolution and fed them into equation (1) in order to compute their values for ΔM_i . Then we selected candidate classes for reuse and collected their change metrics after they have been refactored (ΔR_i). Finally, we used a statistical test to determine whether or not it is possible to reject our null hypothesis.

3.1 Description of the Project and Data Collection Process

The object under study is a commercial software project developed at VTT in Oulu, Finland. The programming language in use was Java. The project was a full business

success in the sense that it delivered on time and on budget the required product, a production monitoring application for mobile, Java enabled devices. The development process followed a tailored version of the Extreme Programming practices [1], 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. Two pairs of programmers (four people) have worked for a total of eight weeks. The project was divided into five iterations, starting with a 1-week iteration, which was followed by three 2-week iterations, with the project concluding in a final 1-week iteration.

The developed software consists of 30 Java classes and a total of 1770 Java source code statements (denoted as LOC). Throughout the project mentoring was provided on XP and other programming issues according to the XP approach. Three of the four developers had an education equivalent to a BSc and limited industrial experience. The fourth developer was an experienced industrial software engineer. The team worked in a collocated environment. Since it was exposed for the first time to the XP process a brief training of the XP practices, in particular of the test-first method was provided prior to the beginning of the project.

To collect the metrics listed in Table 1 we used our in-house developed tool PROM [26]. PROM is able to extract from a CVS repository a variety of standard and user defined source code metrics including the CK metric suite. Not to disrupt developers we set up the tool in the following way: Every day at midnight automatically a check-out of the CVS repository was performed, the tool computed the values of the CK and complexity metrics and stored them in a relational database. In this way we obtained directly the daily evolution of the CK metrics, LOC and McCabe’s cyclomatic complexity.

3.2 Results

We were able to collect the daily evolution of the metrics in Table 1 for the entire period of development, which was 8 weeks, apart from 3 days. In these days developers apparently did not check-in the source code and therefore we had to omit them from our analysis.

The design of the developed system is based on the MVC pattern [6], the Broker architectural pattern [6] and several standard design patterns described in [18]. We think that some basic classes of these patterns – their importance is also emphasized by the design document – are particularly interesting to be considered for reuse. Out of them we choose a subset of classes, which have been refactored during development. We can infer this information from two user stories that have been implemented specifically for refactoring tasks and comments added in the respective classes.

We select in total five candidate classes and compute in a first step the daily changes of the metrics for each of them omitting the days when they have been refactored. We denote the five classes by A, B, C, D, and E. After we compute the average of these changes for all days in which a class has been refactored (the considered classes have been refactored at most on two different days during development). Table 2 shows the results: For each metric and candidate class we indicate the average changes during development (without refactoring), ΔM_i , the average changes induced by refactoring, ΔR_i , and whether or not we can reject our null hypothesis, H_0 . We

accept or reject H_0 by applying a one-sample Wilcoxon rank sum test [20]: We test whether a sample of changes for metric M_i has a median ΔR_i or not. For the test we use a significance level of $\alpha=0.05$.

Table 2. Average daily changes of quality metrics in case of refactoring (ΔR) and development (ΔM). A **1** in the column with heading H means that we can reject the null hypothesis for the particular class and metric, 0 means that we cannot reject the null hypothesis. Values are rounded to their closest integer.

Class	CBO			RFC			WMC			LCOM		
	ΔM	ΔR	H	ΔM	ΔR	H	ΔM	ΔR	H	ΔM	ΔR	H
A	0	0	0	0	1	1	1	-1	1	0	-1	1
B	1	-4	1	1	-4	1	0	0	0	0	0	0
C	1	0	0	2	-5	1	4	0	0	1	0	0
D	1	-1	1	1.4	-2	1	2	0	0	1	0	0
E	1	-1	1	3.5	-2	1	2	3	1	0	0	0
	MAX_MCC			MAX_LOC			DIT			NOC		
A	0	-1	1	0	0	0	0	0	0	0	0	0
B	0	0	0	2	-2	1	0	0	0	0	0	0
C	3	0	0	6	-46	1	0	0	0	0	0	0
D	3	-2	1	0	0	0	0	0	0	0	0	0
E	1	0	0	10	-20	1	0	0	0	0	0	0

The interpretation of the numbers in Table 2 is straightforward: For every candidate class there are at least two quality metrics that improve significantly after it has been refactored (compared to the average evolution during development). In particular classes A and E show a notable enhancement: These two classes provide general interfaces to the user interface and database and it is likely that they will be reused in a similar application.

By investigating the different metrics we notice that not all of them are affected in the same way by refactoring: The metrics related to inheritance and cohesion for example are not at all or only in a negligible way changed by the refactorings applied in the project. This could be explained by the fact that the software under scrutiny is relatively small: It does not use deep inheritance hierarchies and only in a limited way inheritance as a mechanism for reuse. Therefore, it is quiet obvious that no refactoring dealing with inheritance has been applied (it was not necessary to restructure code due to complexity caused by inheritance). As for LCOM several researchers have questioned its meaning and the way it is defined by Chidamber and Kemerer [9]; the impact of LCOM on software reusability is little understood by today and therefore we do not analyze it further in this research.

The highest benefit of refactoring show the CBO and RFC metrics: They express the coupling between different classes and the complexity of a class in terms of method definitions and method invocations. We believe that these two metrics are strong indicators for how difficult it is to reuse a class: A high value of RFC makes it difficult to understand what the class is doing and a high value of CBO means that the class is dependent on many external classes and difficult to reuse in isolation. Both situations prevent it from being easily reused. For three out of the five candidate classes refactoring improves significantly both the RFC and CBO values and as such clearly makes them more suitable for ad-hoc reuse.

Refactoring seems also to lower method complexity: In all the classes either the method with the maximum lines of code or the one with the highest cyclomatic complexity have gained a notably improvement after refactoring. Again, classes with less complex methods are easier to reuse.

Summarizing our results we can reject hypothesis H_0 for several metrics M_i (in particular for the RFC and CBO metric) but not for all of them (like the inheritance related metrics) and not for all classes we selected. We can conclude that refactoring improves for every class we analyze at least two internal metrics that are important for reusability; moreover, for most of them it lowers significantly coupling and method invocation complexity – two “code smells” [14] that often prevent classes from being reused in an ad-hoc manner. Overall the results of this case study give strong evidence that refactoring supports ad-hoc reuse in an XP-like development environment.

4 Threats to Validity and Future Work

This research addresses the question whether refactoring supports ad-hoc reuse or not. We try to answer it by analyzing and comparing the evolution of quality metrics of a software system during “traditional” development and after phases of refactoring. Our approach considers only a set of internal product metrics that may be useful and important as reusability indicators. Of course, this is only half of the story and a complete model should also consider external product and process metrics that characterize reusability.

Regarding the internal validity of this research we have to address the following threats:

- The subjects of the case study are heterogeneous (three junior and one experienced developers) and use for the first time an XP-like methodology. This could affect seriously our findings, as for example junior developers may behave very different than experienced ones. Also the kind and application of refactorings depend highly on the experience of a developer and could lead to different results in other environments. Moreover, a learning effect could be visible and for example influence the evolution of reusability metrics during the project.
- We do not validate the approach we propose and use in this case study. I.e. we do not analyze if refactored classes that show an improvement of internal quality metrics are reused more often and more easily in the same or similar projects. Such validation has to be addressed in a future study.

- Finally, the choice of candidate classes, quality metrics and the time interval we use to compute their changes is subjective. Although we tried to motivate our choices we plan to consider variations in metrics and time interval in future experiments in order to confirm or reject the conclusions of this research.

Altogether, as with every case study the results we obtain are valid only in the specific context of the experiment. In this research we analyze a rather small software project in a highly volatile domain. A generalization to other application domains and XP projects is only possible by future replications of the experiment in such environments.

5 Conclusions

Although agile processes and practices are gaining more and more importance in the software industry much more work has to be done to convince managers to introduce new and innovative development concepts in their companies. This research focuses on whether refactoring, a key practice of XP, supports ad-hoc reuse or not. Software reuse is a key success factor for software development and should be supported as much as possible by the development process itself. We believe that refactoring supports and enhances ad-hoc reuse in a software project, which does not address reusability as one of its primary goals.

The contribution of this research is twofold: First, we propose a methodology for assessing if refactoring improves quality and therefore promotes ad-hoc reuse of object-oriented classes during development. Second, we conduct a case study in which we apply our methodology and which allows us to provide in a quantitative way and in a close-to industrial environment an answer to our research question.

The main conclusion of this research can be summarized as follows:

Refactoring seems to improve significantly important internal measures for reusability of object-oriented classes written in Java. Therefore, we can sustain our claim that refactoring has a positive effect on reusability and for sure promotes ad-hoc reuse in an XP-like development environment.

Of course refactoring as any other technique is something a developer has to learn and to train. 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 delivers code, which is easier to reuse than code which has not been refactored. Afterwards, they have to provide training and support to change their development process into a new one that includes continuous refactoring. Agile Methods already use refactoring as one of their key practices and could be a first choice for developing code in a way that supports - among other benefits such as good maintainability - also reusability.

References

1. 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. Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'04, Vancouver, British Columbia, Canada (2004)

2. Basili, V., Briand, L., and Melo, W.L.: A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, **22**(10): 267-271 (1996)
3. Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley (2000)
4. Benlarbi, S., El Emam, K., Goel, N., Rai, S.: Thresholds for Object-Oriented Measures. *Proceedings of 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, p. 24 (2000)
5. Bois, B. D., Demeyer, S., Verelst, J.: *Refactoring – Improving Coupling and Cohesion of Existing Code*. Belgian Symposium on Software Restructuring, Gent, Belgium (2005)
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M.: *Pattern oriented software architecture. Volume 1: A System of Patterns*. John Wiley & Sons (1996)
7. Caballero, R., and Demurjian, S.A.: Towards the Formalization of a Reusability Framework for Refactoring. *Proceedings of the 7th International Conference Software Reuse: Methods, Techniques, and Tools, ICSR-7, Austin, TX, USA (2002)*
8. Chidamber, S., Kemerer, C.F.: A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, **20**(6): 476-493 (1994)
9. Counsell, S., Mendes, E., Swift, S.: Comprehension of object-oriented software cohesion: the empirical quagmire. *Proceedings of the 10th International Workshop on in Program Comprehension, Paris, France (2002)* 33 – 42
10. Curry, W.E., Succi, G., Smith, M.R., Liu, E., and Wong, R.W.: Empirical Analysis of the Correlation between Amount of Reuse Metrics in the C Programming Language. *Proceedings of the 1999 Symposium on Software Reusability (SSR'99), Los Angeles, Ca, USA (1999)*
11. Dandashi, F., and Rine, D.C.: A Method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements. *Proceedings of 17th ACM Symposium on Applied Computing (SAC 2002), Madrid (2002)*
12. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding Refactorings via Change Metrics. *Proceedings of the 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'00, Minneapolis, USA (2000)*
13. Devanbu, P., Karstu, S., Melo, W., and Thomas, W.: Analytical and Empirical Evaluation of Software Reuse Metrics. *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany (1996)*
14. van Emden, E., and Moonen, L.: *Java Quality Assurance by Detecting Code Smells*. *Proceedings of the 9th Working Conference on Reverse Engineering, IEEE Computer Society Press (2002)*
15. Fenton, N., and Pfleeger, S.L.: *Software Metrics A Rigorous & Practical Approach*. PWS Publishing Company, Boston (1997) pp. 408
16. Fowler, M.: *Refactoring Improving the Design of Existing Code*. Addison-Wesley (2000)
17. Frakes, W., and Terry, C.: Reuse Level Metrics. *Proceedings of the 3rd International Conference on Software Reuse, Rio de Janeiro, Brazil (1994)*
18. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design patterns: Elements of Reusable Object-Oriented Software*. New York, Addison-Wesley (1995)
19. Johnson, P.M., Disney, A.M.: Investigating Data Quality Problems in the PSP. *Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (SIGSOFT 98) (1998)*
20. Hollander, M., Wolfe, D.A.: *Nonparametric statistical inference*. New York: John Wiley & Sons (1973) 27-33
21. Layman, L., Williams, L., Cunningham, L.: Exploring Extreme Programming in Context: An Industrial Case Study. *Agile Development Conference (2004)* 32-41

22. Li, W., Henry, S.: Maintenance Metrics for the Object Oriented Paradigm. Proceedings of the First International Software Metrics Symposium, Baltimore, MD (1993) 52-60
23. McCabe, T.: Complexity Measure. *IEEE Transactions on Software Engineering*, **2**(4): 308-320 (1976)
24. Poulin, J.S.: Measuring Software Reusability. Proceedings of the Third Conference on Software Reuse, Rio de Janeiro, Brazil (1994)
25. Ratzinger, J., Fischer, M., Gall, H.: Improving Evolvability through Refactoring. Proceedings of the 2nd International Workshop on Mining Software Repositories (MSR'05), Saint Louis, Missouri, USA (2005)
26. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data. Proceedings of the EUROMICRO 2003 (2003)