

PhD Proposal

Refactoring Process Improvement Opportunities, Alternatives and Benefits

Sérgio Bryton
bryton@di.fct.unl.pt

2010



Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia

Supervisor

Prof. Doutor Fernando Brito e Abreu

Referees

Prof. Doutor Ricardo Machado

Prof. Doutor Ademar Aguiar

Prof. Doutor Miguel Pessoa Monteiro

Contents

List of figures	iii
Abstract	1
1 Introduction	3
2 Research statement	5
2.1 Research problems	5
2.2 Motivation	6
2.3 Expected outcomes	7
2.4 Research questions	8
2.5 Validation	8
3 Research context	9
3.1 Refactoring concepts, approaches and trends	9
3.2 Refactoring in practice	9
3.3 Refactoring opportunities identification	10
3.4 Refactoring alternatives selection	12
3.5 Refactoring tools	13
3.6 Refactoring impact on software quality	13
3.7 Synthesis	14
4 Workplan	17
4.1 Methodological approach	17
4.1.1 Introduction	17
4.1.2 Iterations and phases	18
4.1.3 Major deliverables	19
4.1.4 Results presentation	19
4.2 Current work	21
4.2.1 Quantitative evaluation of Fowler's refactoring example	21
4.2.2 Quantitative-Based Refactoring method	21
4.2.3 Fowler's Long Method detection model	22
4.2.4 Refactoring experiment	23
4.2.5 Plan evolution	23
Acknowledgments	27

List of Figures

- 1.1 UML activity diagram for the refactoring process 3
- 4.1 QBR Overview 22

Abstract

Refactoring, in spite of widely acknowledged as a best practice, still has some limitations; namely, it lacks evidence regarding its claimed benefits on software quality, and tasks like the identification of refactoring opportunities and the decision on how to refactor are still performed manually and arbitrarily, making this a cumbersome, irrepeatable and error-prone process and, ultimately, hampering its widespread adoption by the community as a regular practice. This research will address these limitations by providing, on one hand, quantitative evidence on the effects of refactoring on software quality and, on the other hand, providing a method and techniques which are expected to support tool-based refactoring activities. By achieving these results, we expect to clarify the relevance of refactoring to software quality and reduce the time and cost actually put into this activity, thus facilitating its widespread adoption.

Keywords (IEEE): D2 [Software Engineering]: Quality analysis and evaluation; Restructuring, reverse engineering and reengineering; Maintainability.

1 Introduction

Refactoring [1] is the process of modifying software implementation to improve its internal structure, without altering its external behavior. It is claimed to optimize software quality, namely maintainability, reusability and efficiency [2]. Implicitly, this technique encompasses the following five main activities, also represented in Fig. 1.1 :

1. Find where a refactoring is required;
2. Choose the most adequate refactoring;
3. Apply the selected refactoring;
4. Guarantee that the external behavior of the software is not altered;
5. Make sure that the systems' internal structure has effectively improved.

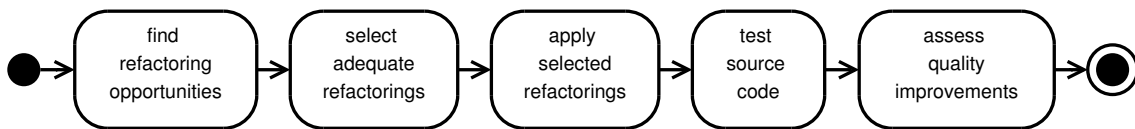


Figure 1.1: UML activity diagram for the refactoring process

In the context of these activities, from our perspective, four main questions arise:

1. Which part of the code should be changed?
2. Which kind of refactoring should be applied?
3. How to evaluate that the external behavior of the software remains the same?
4. How to evaluate the quality improvements?

It is important to mention, though, that these questions, among many other inherent to refactoring, have already been identified [3] and that the purpose of this research is to contribute to answering some of them.

While for assessing the stability of the external behavior, automated regression tests will do a good job, for the remainder questions, the answer is not so straightforward. Supposedly, potential refactoring opportunities, also known as *code smells*, are prejudicing the software quality, and must be found within the code. Therefore, a good way to find them, would be by looking for software quality problems within the code and then establishing a relationship between these problems and the refactoring opportunities. This will then help in identifying each code smell and the set of possible

solutions for it, since the relationship between the refactoring opportunities and the possible refactorings has already been established in catalogues [1, 4, 5, 6]. However, with the possible solutions to improve the code, it is still necessary to select, among these alternative refactorings, which is the most adequate. A serious evaluation of improvements should be formal and quantitative, and the refactoring should only be performed if, in fact, the quality of the internal structure of the code is going to improve, as intended. For this to happen, it is necessary to choose the best refactoring, not only in terms of being an adequate solution to the problem identified, but also in terms of being the solution which best improves the internal software quality. Most of these tasks are currently performed manually and without a clear method to guide them which ensures that, in the end, internal software quality has improved, as supposed to.

The problems to be tackled by this research, which are further discussed in sec. 2.1, are then as follows:

- Difficulty on the identification of refactoring opportunities;
- Difficulty on the selection of the refactorings to apply;
- Lack of a method driven by software quality improvement which encompasses the necessary activities and artifacts;
- Lack of evidence regarding the influence of refactoring on software quality.

To tackle the above mentioned problems we intend to develop, according to chapter 4, a set of techniques and a method for refactoring which improves software quality, experimentally validate them and, consequently, provide evidence of the refactoring influence on software quality. Therefore, our major contributions, which are further discussed in sec. 2.3, are as follows:

- A technique to identify refactoring opportunities;
- A technique to support the decision on which refactorings to apply;
- A method for refactoring driven by software quality improvement;
- Evidence of the influence of refactoring in software quality.

By providing the aforementioned contributions, we expect to clarify the relevance of refactoring to software quality and provide the means necessary to help in reducing the time and cost actually put into this activity, thus facilitating its widespread adoption, as further described in sec. 2.2.

2 Research statement

If refactoring is supposed to improve software structure, and the quality of this structure can be quantified, then this quantification can be used to detect the places to improve and to choose how to improve them, in order to achieve the optimal structure among the alternatives. As such, our research statement is:

“Refactoring can be driven by software quality quantitative assessments”

The problems this research will address, their relevance, the ways to address them and their novelty will be presented within this section.

2.1 Research problems

After roughly 10 years of research, refactoring is still a fragile process, poorly grounded, hard to perform and sometimes producing unwanted results in software quality [7, 8].

Potential refactoring opportunities are probable design flaws which prejudice maintainability and reusability [1, 9], and are positively associated with the class error probability [10]. Several refactoring catalogues [1, 4, 5, 6] have been proposed which, among other things, identify the potential refactoring opportunities and the ways to detect them. The ways proposed, so far, to detect these potential refactoring opportunities are mostly subjective. This problem, makes the whole process quite difficult and dependent on the developer’s experience and knowledge. An example is the heuristic proposed by Fowler [1] to detect the *Long Method* bad smell:

“Whenever we feel the need to comment something, we write a method instead”

There are some attempts to automate the detection of potential refactoring opportunities with software quality metrics [11, 12, 13, 14] and tools [15]. Still, most of these attempts require the configuration of the tools with metrics’ thresholds and, to the best of our knowledge, these thresholds are not consensual and have not been validated, thus not granting objectivity to the detection process.

Another problem is the selection of the refactoring to apply. Currently, to choose among a set of alternatives, the developers have to use the catalogues of reference

[1, 6, 5, 4], eventually aided by quick reference cards [16]. The refactoring selection, supposed to improve internal software quality, is not driven by improvement in software quality and, consequently, the results may contradict the purpose these activities are developed for, like in [7, 8].

The fact of not existing a well defined method, which integrates all refactoring activities and artifacts, focused in delivering higher quality software, leaves room for unexpected results, on one hand and, on the other hand, makes the whole process unrepeatable and prone to errors.

Finally, some of the available quantitative results, regarding the evaluation of refactoring contribution towards software quality improvement, are contradictory [7, 8, 17, 18, 19] and present fragilities, namely regarding the methods used to obtain the results. This means that there is room to provide stronger evidence regarding the claimed benefits of refactoring on internal software quality.

The refactoring problems this research will address are then, as follows:

- P1 - Difficulty on the identification of refactoring opportunities;
- P2 - Difficulty on the selection of the refactorings to apply;
- P3 - Lack of a method driven by software quality improvement which encompasses the necessary activities and artifacts;
- P4 - Lack of evidence regarding the influence of refactoring on internal software quality.

2.2 Motivation

By contributing to the solution of the problems pointed earlier, namely by reducing the time and cost required by this activity, and increasing its effectiveness, we are improving the refactoring process and, consequently, facilitating its adoption. By making easier for refactoring to become a generalized practice in the software development process, while making sure it improves software quality, we are providing a way to improve the overall quality of software and reduce maintenance costs.

If potential refactoring opportunities could be suggested by an IDE¹, instead of being searched randomly and manually throughout the code, without a common and clear criteria, the problems in code could be corrected earlier and faster, or even avoided as they are being introduced by developers.

If the selection of refactorings to apply could also be suggested by an IDE, such that software quality improvement is the main concern, then, not only the developers would have a repeatable and common criteria on how to perform the changes, but also the refactorings would only be performed if indeed software quality benefits were ensured.

¹Integrated Development Environment

If a method existed to integrate all refactoring activities and artifacts, in a way that, in the end, software quality was improved, then the whole refactoring process would be more coherent and robust.

Finally, by providing robust scientific evidence that refactoring indeed can deliver software quality improvements, then this process would be more credible.

Overall, improving refactoring is improving maintainability, which means software less expensive and developed faster.

2.3 Expected outcomes

To tackle the above mentioned problems we will develop and experimentally validate a set of techniques, and a method for refactoring, which improves software quality. We will also provide scientific evidence of the refactoring influence on software quality. Therefore, our expected outcomes are as follows:

- O1 - A technique to identify refactoring opportunities;
- O2 - A technique to support the decision on which refactorings to apply;
- O3 - A method for refactoring driven by software quality improvement which integrates refactoring activities and artifacts;
- O4 - Evidence of the influence of refactoring in software quality.

A possible approach for a technique to identify possible refactoring opportunities could be a model based on the *Binary Logistic Regression* (BLR) [20], where such a model would be defined based on software quality metrics and opinion from experts for each refactoring opportunity. An example of such a model is the one depicted in (2.1) to detect Fowler's *Long Method* [1]. This model is an instance of the BLR with metrics such as *Method Lines of Code* (MLOC), *Number of Parameters* (PAR), *Cyclomatic Complexity* (VG) or *Nested Block Depth* (NBD) [21] and provides a probability of a method being a *Long Method*.

$$P = 1/(1 + e^{-(B0+B1*VG+B2*MLOC+B3*NBD+B4*PA)}) \quad (2.1)$$

According to the available refactoring catalogues [1, 6, 4, 5] each potential refactoring opportunity can, generally, be refactored in different ways. The possible ways depend on the kind of problem which is affecting the code, the way the code is implemented in terms of structure, the semantics of the code, and the quality benefits expected to achieve. Therefore, a technique to support the refactoring decision has to deal, as much as possible, with these concerns. We envision such a technique based on decision support systems and software quality metrics. In [22] the authors proposed a way to rank refactoring options based on the developers decision on

which software quality attribute to privilege. We will consider such an approach, but instead of being the developer to define the required knowledge base (how each refactoring affects each quality property) this should be defined based on the results of experimentation and scientific evidence.

2.4 Research questions

The fundamental research question that will be addressed within this research is :

- Q1 - Can the software refactoring process be improved?

This research question can be further divided into sub questions, as follows:

- Q1.1 - Can refactoring opportunities detection be based on internal software quality attributes?
- Q1.2 - Can the best refactoring alternative selection be based on internal software quality attributes?
- Q1.3 - Can a quantitative-based refactoring process improve internal software quality?
- Q1.4 - Can a quantitative-based refactoring process improve internal software quality more than traditional refactoring approaches?

2.5 Validation

To validate our outcomes, we will develop a series of experiments, following the Scientific Method and Experimental Software Engineering best practices, as described in section chapter 4. In fact, we have already partially performed such an experiment, as described in section sec.4.2.4. To do this, we will require source code samples, which satisfy the requirements for results generalizability, as well as other resources, such as persons or tools, available and able to participate in all phases and activities of the refactoring process. We will also need both code versions, before and after the refactoring, to establish comparisons.

3 Research context

The topic of this research is not new. There are already some published results related to the problems we are trying to solve. However, in our perspective, there is still much to be done. Either because the existing results fall short on expectations, or they present threats to validity, or because they can not be generalized, which hampers its widespread adoption.

3.1 Refactoring concepts, approaches and trends

Since refactoring has been introduced as a software engineering practice, the main references that ground it, regarding the process as a whole, are those from Fowler [1], Kerievsky[6], Monteiro[4] and Laddad[5].

While Fowler provides a catalogue for refactoring within the object-oriented paradigm, Kerievsky provides a catalogue to refactor taking advantage of the GoF Design Patterns, and Monteiro and Laddad propose catalogues to take advantage of the aspect-oriented paradigm.

Mens, provides an overview of Refactoring [3], its trends and open questions, some of which are being addressed by this research and, in [23], he provides a survey on refactoring.

3.2 Refactoring in practice

In [24], an experimental study regarding how programmers refactor is presented. This study is mostly grounded on data related to the Eclipse IDE tool [15] and focuses the activities after the detection of *code smells*. Some fragilities of this study are pointed out by the authors; however, the most relevant, in our opinion, is the absence of statistical methods to test the hypotheses. Among their conclusions we highlight the following:

- Refactoring is a frequent activity, related with the anticipation of large changes to the code base;
- 90% of refactorings are performed manually, without the help of a tool;
- About 90% of configuration defaults of refactoring tools remain unchanged when programmers use the tools;

- The kind of refactoring performed with tools differs from that performed manually;

The authors also argue that the simplicity of user interfaces is a key issue to increase the usage of refactoring tools.

In [25] the authors present an experience report regarding the refactoring of a large commercial system. The authors claim that current refactoring techniques do not scale to the size of large scale redesign, whereas this is a recurring activity, and propose a new refactoring called *Extract subsystem to a separate process*. The authors report that they tried to apply small refactorings to break up the system slowly but failed, however it is not clear why. To find the borders among the components the authors reported to have studied the code with the help of tools and it can be inferred that functional concerns were the main drivers for this task. The authors do not make explicit if small refactorings were used to prepare the task of splitting the three core components (Game process, Reporting process and Shared library) once their borders have been identified.

3.3 Refactoring opportunities identification

The detection of refactoring opportunities is the first activity in refactoring. It is from the moment that potential problems in code are found that the remainder activities are developed. So far, several recent, and not so recent, approaches to help formalize and automate this activity have been developed.

In [26] the authors claim that many refactorings can be detected using low cost syntactic techniques and present a tool (Look#) to detect refactoring opportunities in C# programs. They claim to have an excellent success rate. The authors underline the idea that refactoring is a difficult and time consuming process and also that refactoring tools focus in applying refactorings, rather than on detecting their need. They also believe that finding the refactorings is a harder task than applying them and find highly desirable the existence of tools to perform the first. The authors consider that about one third of Fowler's refactorings are too high-level for a tool to detect or perform and provide, as an example, the *Substitute algorithm* refactoring. They estimate that half of these refactorings can be discovered by syntactic means, although only presented results for 11, and, as such, chose to detect them with simple code metrics and symbol table information rather than with data-flow analysis, which evidences some limitations like performance. To validate their tool all the detected refactoring opportunities were inspected manually. They admit the possible existence of unidentified refactorings in the code but claim their main focus was on avoiding false positives. As such, this tool may be effective, but there is no evidence regarding its efficiency. Furthermore, it is not clear how the manually identified refactoring opportunities were validated, since the refactoring guidelines presented by Fowler are quite subjective in nature. Finally, the conclusions of this paper can

not be generalized since a proper experiment was not carried out. The tool is based on customizable algorithms with thresholds for metrics, for instance. However, the criteria for such thresholds, as well as the thresholds used are not available. According to the authors, 88% of the refactorings were identified correctly and, in most cases, it is reported a 100% accuracy. In this context, the less accurate detection was regarding *extract class* and *move field*. Curiously, the authors adopted a strategy where instead of looking for the code smells, they looked for the refactorings, that is, they looked for a place where a specific solution would fit, instead of finding the problems and then decide which solution to adopt. According to Fowler, a single refactoring can be the right solution for several code smells and a single code smell can have many refactoring alternatives. Take for instance the *move method* refactoring, used by the authors, which can be the solution for the code smells *Alternative Classes for Different Interfaces*, *Data Class*, *Feature Envy* or *Message Chains*, just to name a few. As such, it is highly unlikely that a single algorithm can detect all these different smells, meaning that the probability of false negatives is expected to be quite high.

In [27] the authors treat the prediction of refactoring opportunities as a data mining problem and propose a machine learning based model (Weighted Naïve Bayes with InfoGain heuristic) to predict classes to be refactored, by analyzing 26 code complexity metrics. InfoGain is used as an heuristic to weight the different metrics. Their results were validated experimentally within the four versions of a project from a GSM company and they claim to be able to predict 82% of the classes that need refactoring with 13% of manual inspection effort on the average. Their process is shown to improve with system evolution. The authors underline the idea that a proper tool support for refactoring activities can decrease cost and increase the quality of software. The downsides of this technique are that it is not able to provide information about false negatives neither does provide information regarding where the classes should be refactored, which implies an additional manual inspection to assess this. Its limitations are that the results can not be generalized and the criteria used to weight the different metrics is not clear.

In [28] the authors propose a way to identify *Move Method* refactoring opportunities to eliminate *Feature Envy* code smells. This approach is based on a proposed metric *Entity Placement* and has been validated in two open source projects. Within this approach, very similar to the MORE method [29], the impact of the suggested refactoring is evaluated before the refactoring is actually performed, thus enabling the refactoring alternatives to be ranked according to this criteria. These results can not be generalized since a proper experiment has not been conducted and, although the reported accuracy was high, regarding false positives, there is no information regarding the existence of false negatives. The metric used to assess coupling improvement was *Message Passing Coupling* (MPC), which is quite short, considering the wide range of metrics available to assess coupling at method level. The same goes for the only metric used for evaluating cohesion. The alleged irrelevance of the remainder metrics should have been proved statistically and not subjectively, con-

trusting to the quantitative nature of this work. Also, refactorings produce changes at software properties rather than coupling, therefore, there is no evidence that any of these has not been prejudiced with the suggested and applied changes. Moreover, the independent designer opinion is not statistically significant. Finally, the use of the *Pearson correlation* has not been justified and, in any case, is not enough to establish a causal relationship, meaning that these findings may be occasional. The authors underline the idea that finding the places where refactorings should be applied is neither trivial nor supported by tools.

We can conclude that there is a lot of room to improve the detection of refactoring opportunities. Most of the approaches proposed so far either are not adequately grounded or fall short from expectations by not providing enough information to automate the process. This implies that proper tools can not be developed and suggests that most refactoring opportunities identification activities are currently being made manually.

3.4 Refactoring alternatives selection

After detecting the potential problems in code, it is time to decide whether and how to refactor the code and eliminate the problems, in a way that software quality is improved without altering the original behavior. To support this activity there are not so many initiatives.

In [22], the authors propose a technique to rank refactoring patterns to improve a set of quality attributes of a piece of software. By establishing such a rank, the developer can focus on the refactoring opportunities that, when refactored, will contribute to benefits on specific software quality attributes, thus making the refactoring activity more efficient. This technique uses the *Analytical Hierarchy Process* (AHP) to express the relative importance of the quality attributes and the relative importance of refactoring patterns in regard to those selected quality attributes. The authors demonstrate how the proposed technique should be used and developed a proof-of-concept tool. To establish pairwise comparisons between refactoring alternatives grounded on software quality attributes improvement, a knowledge base is required. We are not aware of the availability of such a knowledge base. If the developer is supposed to make up his own, this is a threat to validity, because the conclusions can not be generalized on the grounds of a non representative (of the real world) knowledge base. Furthermore, the relationship between the refactoring opportunities and the refactorings to apply is not well explained. This is important since a refactoring opportunity can lead to several refactorings, in theory, but in practice, only a few or, most of the times, one, should or can be used, when other software properties like code structure or semantics are taken into consideration. This means that such a rank may be correct in theory but wrong in practice.

The lack of research in this particular area further aggravates the fragility of the refactoring process. So far, the identification of the potential problems in code was

not easy, to say the least, but the lack of support to decide how to change the code is really a major downside of refactoring.

3.5 Refactoring tools

Ideally, tools should provide a fully automated refactoring process, whilst giving the developer a chance to select the most appropriate decisions at their discretion. To fully achieve this purpose, tools must be able to detect, decide upon changes, and assess the results achieved, in a quantitative way, as seen before. However, even though at least 31 refactoring tools for 10 different languages exist [30], none of these seems to fully support the aforementioned requirement. Nevertheless, two of them are worth mentioning for being a step ahead: *JDeodorant* [31] and *TRex* [32]. *JDeodorant* is an *Eclipse* [15] plug-in that identifies *Feature Envy* [1] bad smells in Java projects and resolves them by applying the appropriate *Move Method* refactoring upon ranking them. The whole process is grounded on dissimilarity metrics and no changes to the source occur until the decision to refactor is taken. No post-refactoring assessment is made, and the tool is very limited in scope, since only one smell for Java is supported. Still, the process is quite elegant and promising. *TRex* is also an *Eclipse* plug-in that automates the application of refactorings and the detection of refactoring opportunities for test suites specified using the Standardized Tree and Tabular Combined Notation (TTCN-3) [33]. The whole process is grounded on specific metrics for test suites and pattern-based analysis. Refactorings can be applied in two different ways: either the developer invokes the refactoring from the code location, or the refactoring is invoked directly by a quick fix which is provided by the analysis results of the automated quality assessment. As for this tool, besides being too domain-specific, no post-refactoring assessment is performed. However, this process is also quite elegant and aiming systems with thousands of lines of code. The remainder tools, from which [15, 34, 35, 36] are some examples of the state-of-the-art, where *IntelliJ* [36] seems to be the most advanced one, focus mostly on implementing well the refactorings for their corresponding target language. The assistance some provide to detect refactoring opportunities consists essentially in making available for the end user customizable thresholds for common software metrics. They also do not help on choosing the right refactoring and they do not assess the final result. Basically, the user decides everything, while the tool tries to implement the decisions quick and cleanly.

3.6 Refactoring impact on software quality

Regarding the claimed benefits of refactoring on software quality, while some researchers have tried to provide evidence of it [17, 18, 19], other researchers still get puzzled when confronted to the fact that, although refactorings are used to improve

the quality of their systems, a supposedly adequate set of metrics indicates that this process often has the opposite result [7, 8]. In general, most of the approaches to provide this evidence have not completely achieved its objective, essentially because of the procedures used to do it, which frequently can not be replicated, hampers its results from being generalized or fall short from expectations. In fact, when experimentation is performed, the data and source code used within the experiments should be made available along with the publication, in a way that other researchers could repeat the experiment with the same data and using the same process. Achieving the same results in an experiment conducted by different researchers (experiment replication), grounded on the same data and processes, is a well accepted way of validating the results obtained but, unfortunately, too often not possible.

In [8], the authors try to quantitatively evaluate the claims that refactoring improves software quality attributes such as adaptability, maintainability, understandability, reusability and testability. The benefits of such results are on enabling the decision on whether to put time and cost into refactoring. The authors also expect to increase the low number of quantitative studies available in this matter, as well as increase the number of external software quality attributes evaluated so far. Their approach is based on internal software quality metrics applied to three different applications, each from a different domain. For each application two versions were measured, either by graduate or by undergraduate students, one before and another after the refactoring took place. One of the applications was developed by students while the remainder two were open source. The presented relationship between internal and external quality metrics was based on previously published studies. The authors concluded that refactoring does not necessarily improve the software quality attributes evaluated and that classifying refactoring methods based on their measurable effects on quality attributes may help developers in choosing appropriate refactorings to improve certain software quality attributes. The downsides of this study are the fragility of the experimental process performed which cannot be repeated, due to the lack of relevant information like the sample used, the lack of information regarding how were the performed refactorings validated and the rather small size of the applications chosen.

3.7 Synthesis

In Tab. 3.1 is presented a synthesis of the state-of-the-art. Some of the references have already been previously discussed and some that were not, are mentioned just to highlight that we are aware of their existence but we could not yet review them. By placing them in the matrix we have a better picture of the coverage of the related work. In the columns we find the authors and the four major problems addressed by our research. In the rows we find the year, reference, and downside(s) pointed out within the previous sections. Like this, we can quickly have several perspectives of the state-of-the-art, namely on:

1. What are the less and more addressed problems;
2. For how long these problems are being addressed;
3. The path some authors trailed when trying to address these problems;
4. The downsides found in each approach;
5. What are the less and more common downsides.

The most common downsides pointed earlier have been categorized and each one has been assigned a letter. The legend is then as follows:

- (g) - Conclusions cannot be generalized
- (i) - Does not provide sufficient information to be used or replicated
- (*) - Is still to be analyzed within the context of a systematic literature review

Author	P1 problem identification	P2 solution selection	P3 method	P4 benefits evidence
<i>Franck</i>	2001 [11] (*)			
<i>Gronback</i>	2003 [14] (*)			
<i>Atkinson</i>	2005 [26] (g,i)			
<i>Salehie</i>	2006 [12] (*)			
<i>Piveta</i>	2009 [37] (*)	2008 [22] (i)		
<i>Tsantalis</i>	2009 [28] (g)			
<i>Kosker</i>	2009 [27] (g,i)			
<i>Du Bois</i>		2007 [38] (*)		2006 [18] (*)
<i>Kataoka</i>		2002 [19] (*)		2002 [19] (*)
<i>Stroggylos</i>				2007 [7] (*)
<i>Rao</i>				2009 [17] (*)
<i>Alshayeb</i>				2009 [8] (g)

Table 3.1: Refactoring state-of-the-art matrix

Considering Tab. 3.1, the most important conclusions are that our problem P3 (lack of a method for refactoring driven by quality improvement) has not yet been addressed, and that our problem P2 (lack of a technique to select the best refactoring to apply) has also hardly been addressed. We can also conclude that most conclusions from the research works which try to tackle the same problems we do cannot be generalized, replicated or both.

4 Workplan

The work encompassed by this research proposal will now be described from two perspectives: one regarding how it is organized and scheduled, and another regarding what has already been achieved.

4.1 Methodological approach

To describe the way this research work is organized and scheduled to achieve the proposed research objectives, within the available time, the methodology, workplan, major deliverables and results presentation approach will be presented.

4.1.1 Introduction

This research will essentially be based on the Scientific Method and Experimental Software Engineering, and an iterative and incremental approach to implementation and planning will be used.

The Scientific Method will be followed, since it is appropriate for Software Engineering research [39, 40]. At the heart of the Scientific Method is an iterative and incremental approach, where each iteration encompasses a literature review to define the research questions, as well as gather information and resources; the definition of hypotheses; the realization of experiments, data collection, analysis and interpretation and results publication.

To study the refactoring process, from the identification of refactoring opportunities and the decision on how to change the software, to the quantitative evaluation of the refactoring results, experiments will be conducted to try to identify correlations and cause-effect relationships, and assess how the results achieved can be generalized, for which experiment replication [41], where the same experiment is applied repeatedly to different and independent subjects, is mandatory.

Iterative and incremental approaches to planning implementation have been proposed and are succeeding, mostly but not exclusively in Software Engineering projects, to reduce risk, uncertainty and establish trust [42, 43]. This kind of approach fits well the activities encompassed by the Scientific Method and Experimental Software Engineering, which uses the first, and, by using them, we are reducing the risk of the uncertainty inherent to a research process, while striving to obtain results earlier in

the process, in a way that the next activities can benefit from the previous results and the major deliverables incrementally improved, so that the research objectives can be consistently and timely achieved in the end.

4.1.2 Iterations and phases

Following an iterative and incremental approach, the work plan for this research will encompass three iterations. From iteration to iteration, the results from the validation of the previous iteration will be incorporated, the experiments will be replicated and their scope will be progressively increased, the state-of-the-art will be surveyed and relevant evolutions will be taken into consideration and, finally, a revised draft of the dissertation will be delivered. By the end of the third iteration we will deliver the final version of the dissertation with which this research project will be proposed for public examination.

The iterations are then planned as follows, along the expected time frame for this project (3 years):

- Iteration #1 (12 months) - deadline October 2010
- Iteration #2 (12 months) - deadline October 2011
- Iteration #3 (12 months) - deadline October 2012

Each iteration will have four phases and each phase will ground the next one, ensuring that from a state-of-the-art survey to a dissertation draft all the necessary steps are taken, making sure that the expected results are produced and validated.

As such, each iteration will encompass the following phases:

- Phase I (1 months) - Literature review
- Phase II (4 months) - Experimentation and prototyping
- Phase III (3 months) - Consolidation and support of research contributions
- Phase IV (2 months) - Dissemination and peer validation
- Phase V (2 months) - Dissertation writing

In Phase I, the literature will be reviewed. This review will be essentially, but not exclusively, focused in Refactoring, Software Quality and Metrics, Quantitative Methods, Decision Support Systems and Experimental Software Engineering. The results of each literature review (one per iteration) will be incorporated in a broader body of knowledge which will be the foundation of the related work chapter of the dissertation.

In Phase II the experimentation part of this research will take place and, with it, its inherent activities [41], such as experimental design, data collection and analysis, and results packaging. Taking advantage of the iterative process which will encompass

three major experiments, we will replicate the previous experiments while, at the same time, accommodating the results of previous experiments.

Phase III will be dedicated to the consolidation and support of the research contributions. This phase encompasses extracting the experimental results achieved and incorporating them in the expected results of this research. It also encompasses developing prototypes grounded on the results achieved along the iterations and in the context of other projects such as master thesis. If succeeded, these prototypes may imply a significant evolution in the following experiment, as we expect them to.

In Phase IV we will write journal or conference papers and book chapters based on the results. By doing this we expect to have an earlier validation of our results which will contribute to the overall maturity of the results to be achieved with this research. The results to be submitted and the kind of paper (workshop, conference or journal) to be used will vary according to the maturity of the results achieved and the topics of the conferences or journals available at the time.

Finally, in Phase V, we will write a draft of the dissertation. Our intention is to start anticipating earlier the most appropriate document structure and validate it, as well as its contents in each iteration. This draft will be improved iteration after iteration, until the final version is released, at the end of the third iteration.

4.1.3 Major deliverables

The major deliverables of each iteration result from the scope of each phase and from the expected results of this research. Some of these deliverables are casuistic, that is, they are produced once in each phase and they are not necessarily dependent on previous deliverables of the kind. There are also deliverables which are improved iteration after iteration (continuous improvement). As such, the major deliverables are presented in Tab. 4.1.

We expect the technical reports to be the foundations of the major chapters of the dissertation. The papers will provide the necessary peer validation and the dissertation draft will aggregate all the results achieved in a coherent document.

4.1.4 Results presentation

The contributions of a research project like this are expected to be recognized by the scientific community, which means that the results obtained must be published in reference publications, like conference proceedings or journals, in the field. Therefore, it is important to identify beforehand these publication targets. By doing this, we are also identifying the most important sources of information to follow when surveying related work.

Id.	Type	Subject	Phase	Related outcomes	Evolution
<i>D1</i>	Tech. Report	State-of-the-art survey	1	All	Continuous improvement
<i>D2</i>	Tech. Report	Refactoring experiment performed	2	All	Casuistic
<i>D3</i>	Tech. Report	Identification of refactoring opportunities	3	O1	Continuous improvement
<i>D4</i>	Tech. Report	Selection of refactoring alternatives	3	O2	Continuous improvement
<i>D5</i>	Tech. Report	Quantitative - Based Refactoring (QBR)	3	O3, O4	Continuous improvement
<i>D6</i>	Papers	Results achieved in previous phases	4	Context dependent	Casuistic
<i>D7</i>	Dissertation draft	Refactoring process improvement	5	All	Continuous improvement

Table 4.1: Synthesis of the major deliverables

We have gathered a list of the most important potential publication targets to submit the results of this research. The process we followed to obtain them was by analyzing the sources of the most relevant work in this field published so far, and completed it with level A*/A conference and journals, according to the Computing Research and Education (CORE) [44] rankings of journals and conferences.

The list of journals is then as follows:

- ACM Transactions on Software Engineering and Methodology
- IEEE Transactions on Software Engineering
- Empirical Software Engineering: An international journal
- Software: Practice and Experience

The list of conferences is as follows:

- ASE - Automated Software Engineering Conference
- ATVA - International Symposium on Automated Technology for Verification and Analysis

- CGO - International Symposium on Code Generation and Optimization
- EASE - International Conference on Evaluation and Assessment in Software Engineering
- ESEM - International Symposium on Empirical Software Engineering and Measurement
- ICSE - International Conference on Software Engineering
- ICSM - International Conference on Software Maintenance

4.2 Current work

This research is ongoing, and we have already achieved some results. Besides the state-of-the-art survey on which this proposal and previously published papers have been grounded, we have already performed a small case study with one of Fowler's refactoring examples, outlined the Quantitative-Based Refactoring method, defined a preliminary model for the Fowler's *Long Method* detection and conducted our first refactoring experiment.

4.2.1 Quantitative evaluation of Fowler's refactoring example

In order to have a first glimpse on the impact of refactoring in software quality, we have selected Fowler's *Video Store* example from his book [1], and measured some of its software quality attributes, namely those related with maintainability. To achieve this we used a tool [45] and collected a set of metrics, before and after applying the recommended refactorings. We concluded [46] that there is no evidence that the internal software quality has improved; on the contrary, it got worse in size, cohesion and complexity, in general. These conclusions can not be generalized, since an appropriate procedure has not been used to ensure it and the case study is quite small and not representative of a real world system; however, we can not forget that this case study is an example provided by a recognized author in the field and, as such, one would expect the opposite results.

4.2.2 Quantitative-Based Refactoring method

To drive and ground refactoring on quantitative results we need a method where the required activities are defined, as well as the necessary artifacts. In [29] we have outlined such a method, although, at the time, focused only on modularity and called Modularity Oriented Refactoring (MORe). Later on, we generalized and synthesized this method, in a way that refactoring could then be improved based on software properties other than modularity, and called it Quantitative-Based Refactoring (QBR).

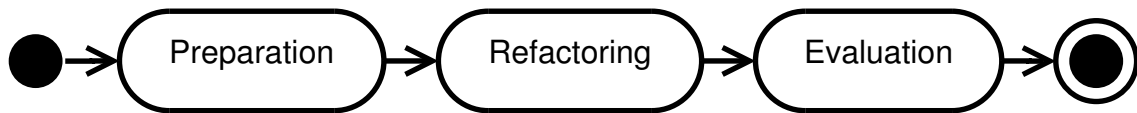


Figure 4.1: QBR Overview

As presented in Fig. 4.1, QBR has three main phases, *Preparation*, *Refactoring* and *Evaluation*. In the preparation phase, the initial product quality is assessed, the code smells are detected, the refactorings to apply are selected and tests are developed. The results of this phase aim to prepare the refactoring, but also to define a quality baseline, to which the resulting system is to be compared, to find out if the refactoring has indeed improved the software quality; otherwise, it would be useless. In the refactoring phase, the previously refactorings are applied. Finally, in the evaluation phase, the quality of the resulting software is assessed and compared to its initial quality, and tests are run to ensure that the behavior has not changed. To use this method, some of its artifacts, namely the smell detection models and the refactoring ranking models are yet to be developed, and are expected outcomes of this research.

4.2.3 Fowler's Long Method detection model

To demonstrate that Fowler's *Long Method* [1] refactoring opportunity can be detected automatically, we defined a model to calculate the probability of a method being a *Long Method*. Our model (4.1) predicts correctly 99% of the methods which are *Long Methods* and 84% of the methods which are not, with a false positive rate of 6% and a false negative rate of 4%, as can be seen from Tab. 4.2.

$$P = 1/(1 + e^{(-(-11.336+0.598*VG-0.057*MLOC+4.701*NBD+0.486*PA))}) \quad (4.1)$$

Observed		Predicted			
		<i>IsLong</i>			
		0	1	Percentage correct	
Step 1	<i>IsLong</i>	0	154	2	98,7
		1	6	31	83,8
	Overall percentage				95,9

a) The cut value is 0,5

Table 4.2: Classification table for BLR Block 1

The most significant predictor in our model, from the set of metrics we used, is the NBD, while the MLOC and PAR are not significant when used with the remainder

variables. The effect of VG, although significant, is much smaller than that of NBD. The PAR is the least important predictor of *Long Methods*, either alone or within the remainder variables in the model.

Most of the software tools with metrics only allow the configuration of those metrics with thresholds, in order to assist the detection of code smells; therefore, software tools should evolve, in order to allow us to configure them with mathematical models, encompassing several metrics, like the one we have used.

Our model cannot be generalized, since the manual identification of the *Long Methods* has been done by a single person, and we have used a single project. However, the goal of demonstrating that the *Long Method* refactoring opportunity can be detected automatically, with mathematical models, grounded on the expert's knowledge, has been fully achieved, with quite satisfactory results.

We believe that the approach we used to detect the *Long Method* can be used to detect other refactoring opportunities and we intend to conduct experiments and propose a general model for the detection of the *Long Method*, as well as using this same approach with more refactoring opportunities to confirm it is also adequate to detect them, and conduct the necessary experiments to define their models.

4.2.4 Refactoring experiment

The first experiment with refactoring has already been performed, however the results are not yet published. Within this experiment we used nine different open source applications, randomly selected, from different domains and with different sizes. These applications are presented in Tab. 4.3.

A subset of classes from each system, up to 40, has been searched, and the refactoring opportunities have been marked, as well as those that were not. As a result, we identified more than 1500 potential refactoring opportunities. A subset of these refactoring opportunities has then been refactored, from which more than 250 refactorings were performed. The refactoring opportunities identification and refactoring operations were conducted by more than 30 master students in software engineering, in the school year of 2009/2010, after being exposed to this technique.

We are now in the process of analyzing the results, which will allow us start developing smell detection models, refactoring ranking models and quality comparisons.

4.2.5 Plan evolution

Presently, we are in the first iteration of this research project, which started in October 2009. This iteration will end by October 2010. Considering the results already achieved and the planned phases and correspondent deliverables, our current situation is as follows:

Name	Description	Reference
<i>Jopt Simple</i>	Java library for parsing command line options	[47]
<i>Apache commons CLI</i>	API for parsing command line options passed to programs	[48]
<i>Natural CLI</i>	Java library providing to developers command line interfaces with human readable sentences	[49]
<i>Monolog</i>	ObjectWeb API for logging	[50]
<i>Commons Logging</i>	Ultra-thin bridge between different logging implementations	[51]
<i>Qflog</i>	Tool for displaying log messages	[52]
<i>Neo4J</i>	Graph database	[53]
<i>tinySQL</i>	SQL engine written in Java	[54]
<i>h2</i>	Java SQL database	[55]

Table 4.3: Applications used within the first refactoring experiment performed

- Phase I (01OCT - 31OCT10) - Literature review
 - TR1 - Technical report on the state-of-the-art survey (ongoing)
 - * List of potential related work (completed)
 - * Revision of papers and books (ongoing)
- Phase II (01NOV - 28FEB10) - Experimentation
 - TR2 - Technical report on the refactoring experiment performed (ongoing)
 - * Experiment #1 experimental design and data collection (completed)
- Phase III (01MAR - 31MAI10) - Consolidation and support of research contributions
 - TR3 - Technical report on the identification of refactoring opportunities (ongoing)
 - * Case study with Fowler's *Long Method* (completed)
 - TR4 - Technical report on the selection of refactoring alternatives (not started)
 - TR5 - Technical report on Quantitative-Based Refactoring (ongoing)
 - * QBR outline and description (ongoing)

- Phase IV (01JUN - 31JUL10) - Dissemination and peer validation
 - Best poster award at the Doctoral Symposium in ICSEA'09 - “Strengthening Refactoring”
 - Paper assessing software quality improvements with Fowler’s “Video Store” case study (ongoing)
 - Paper published in QUATIC2010 - “Reducing Subjectivity in Code Smells Detection: Experimenting with the Long Method” [56]
- Phase V (01AGO - 30SET10) - Dissertation writing
 - Dissertation draft (not started)

Acknowledgments

The work presented herein was partly supported by the *VALSE* project of the *CITI* research center within the Department of Informatics at *FCT/UNL* in Portugal.

References

- [1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
- [2] ISO, *ISO 9126 Standard: Information Technology - Software Product Evaluation - Software Quality Characteristics and Metrics*. International Standards Organization, 2001.
- [3] T. Mens, S. Demeyer, B. D. Bois, H. Stenten, and P. V. Gorp, “Refactoring: Current research and future trends,” *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 3, pp. 483 – 499, 2003. LDTA’2003 - Language descriptions, Tools and Applications.
- [4] M. Monteiro and J. Fernandes, “Towards a Catalogue of Refactorings and Code Smells for AspectJ,” *Transactions on Aspect-Oriented Software Development*, vol. LNCS, no. 3880, pp. 214 – 258, 2006.
- [5] R. Laddad, *Aspect-Oriented Refactoring*. Addison-Wesley, 2006.
- [6] J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2004.
- [7] K. Stroggylos and D. Spinellis, “Refactoring: Does it improve software quality?,” in *Proc. of 5th International Workshop on Software Quality*, ACM Press, 2007.
- [8] M. Alshayeb, “Empirical investigation of refactoring effect on software quality,” *Information and Software Technology*, 2009.
- [9] S. McConnell, *Code Complete*. Microsoft Press, second ed., 2004.
- [10] W. Li and R. Shatnawi, “An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution,” *Information Journal of Systems and Software*, vol. 80, no. 1, pp. 1120–1128, 2007.
- [11] S. Frank, F. Steinbruckner, and C. Lewerentz, “Metrics based refactoring,” in *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pp. 14–16, 2001.
- [12] M. Salehie, “A metric-based heuristic framework to detect object-oriented design flaws,” IEEE Computer Society Press, 2006.
- [13] N. Habra and M. Lopez, “On the use of measurement on software restructuring,” in *Proc. of International ERCIM Workshop on Software Evolution*, 2006.

-
- [14] R. C. Gronback, “Software Remodeling: Improving Design and Implementation Quality, Using audits, metrics and refactoring in Borland Together ControlCenter.” Borland White Paper, 2003.
- [15] E. Foundation, “Eclipse.” <http://www.eclipse.org/>, January 2009.
- [16] Industrial Logic, “Smells to refactorings.” <http://www.industriallogic.com/papers/smellstorefactorings.pdf>, February 2010.
- [17] K. N. Reddy and A. A. Rao, “A quantitative evaluation of software quality enhancement by refactoring using dependency oriented complexity metrics,” *International Conference on Emerging Trends in Engineering & Technology*, pp. 1011–1018, 2009.
- [18] B. Du Bois, *A study of quality improvements by refactoring*. PhD thesis, University of Antwerp, Belgium, 2006. Advisor: Demeyer, S.
- [19] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, “A quantitative evaluation of maintainability enhancement by refactoring,” *IEEE International Conference on Software Maintenance*, p. 0576, 2002.
- [20] S. for Intermediate Statistics, *Leech, N. and Barred, K. and Morgan, G.* Lawrence Erlbaum Associates, second ed., 2005.
- [21] Henderson-Sellers, *Object-Oriented Metrics, measures of Complexity*. Prentice-Hall, 1996.
- [22] E. Piveta, A. Moreira, M. Pimenta, J. Araújo, P. Guerreiro, and T. Price, “Ranking refactoring patterns using the analytical hierarchy process,” 2008.
- [23] T. Mens and T. Tourwe, “A survey of software refactoring,” *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.
- [24] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” in *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, (Washington, DC, USA), pp. 287–297, IEEE Computer Society, 2009.
- [25] P. Adamczyk, A. Zambrano, and F. Balaguer, “Refactoring big balls of mud,” in *Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on Software Engineering*, pp. 50–60, May 2009.
- [26] D. C. Atkinson and T. King, “Lightweight detection of program refactorings,” *Asia-Pacific Software Engineering Conference*, vol. 0, pp. 663–670, 2005.
- [27] Y. Kosker, B. Turhan, and A. Bener, “An expert system for determining candidate software classes for refactoring,” *Expert Syst. Appl.*, vol. 36, no. 6, pp. 10000–10003, 2009.
- [28] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE Transactions on Software Engineering*, vol. 99, no. Rapid-Posts, pp. 347–367, 2009.

- [29] S. Bryton and F. B. Abreu, “Modularity-oriented refactoring,” in *Proc. of the 12th European Conference on Software Maintenance and Reengineering CSMR2008*, pp. 294–297, IEEE Computer Society, 2008.
- [30] M. Fowler, “Refactoring home page.” <http://www.refactoring.com>, August 2008.
- [31] N. Tsantalis, “JDeodorant.” <http://java.uom.gr/nikos/bad-smell-identification.html>, August 2008.
- [32] H. Neukirchen and M. Bisanz, “Utilising Code Smells to detect quality problems in TTCN-3 Test Suites,” in *Proceedings of the 19th IFIP TestCom’2007 and FATES’2007*, pp. 228 – 243, Springer, 2007.
- [33] European Telecommunications Standards Institute (ETSI), “Methods for Testing and Specification (MTS) - The Tree and Tabular Combined Notation version 3 (TTCN-3): Core Language.” <http://www.ttcn-3.org/StandardSuite.htm>, August 2008.
- [34] Oracle, “Netbeans.” <http://netbeans.org/>, 2010.
- [35] Oracle, “Jdeveloper.” <http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html>, 2010.
- [36] IDEA, “Intellij.” <http://www.jetbrains.com/idea/index.html>, 2010.
- [37] E. K. Piveta, M. S. Pimenta, J. Araújo, A. Moreira, P. Guerreiro, and R. T. Price, “Representing refactoring opportunities,” in *Proceedings of SAC 2009* (S. Y. Shin and S. Ossowski, eds.), pp. 1867–1872, ACM, 2009.
- [38] B. Bois and T. Mens, “Describing the impact of refactoring on internal program quality,” in *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*, 2003.
- [39] M. V. Zelkowitz and D. R. Wallace, “Experimental validation in software engineering,” *Information & Software Technology*, vol. 39, no. 11, pp. 735–743, 1997.
- [40] W. Tichy, N. Harbermann, and L. Prechelt, “Future directions in software engineering,” *Software Engineering Notes*, vol. 18, no. 1, pp. 35–48, 1993.
- [41] A. Dean and D. Voss, *Design and Analysis of Experiments*. Springer-Verlag, 1999.
- [42] K. Shwaber, *Agile project management with Scrum*. Microsoft Press, 2004.
- [43] M. Cohn, *Agile estimating and planning*. Prentice-Hall, 2005.
- [44] C. Research and E. A. of Australasia, “Computing Research and Education, CORE.” <http://core.edu.au/>, 2010.
- [45] F. Sauer, “Metrics.” <http://metrics.sourceforge.net>, August 2008.

- [46] S. Bryton and F. B. e Abreu, “Strengthening refactoring: Towards software evolution with quantitative and experimental grounds,” in *ICSEA* (K. Boness, J. M. Fernandes, J. G. Hall, R. J. Machado, and R. Oberhauser, eds.), pp. 570–575, IEEE Computer Society, 2009.
- [47] “Jopt simple.” <http://jopt-simple.sourceforge.net/>, 2009.
- [48] “Apache commons cli.” <http://commons.apache.org/cli/>, 2009.
- [49] “Natural cli.” <http://naturalcli.sourceforge.net/>, 2009.
- [50] “Monolog.” <http://monolog.ow2.org/doc.html>, 2009.
- [51] “Apache commons logging.” <http://commons.apache.org/logging/>, 2009.
- [52] “Qflog.” <http://www.qfs.de/en/qflog/>, 2009.
- [53] “Neo4j.” <http://api.neo4j.org/>, 2009.
- [54] “Tinysql.” <http://www.jepstone.net/tinySQL/>, 2009.
- [55] “H2.” <http://www.h2database.com/html/main.html>, 2009.
- [56] S. Bryton, F. B. e Abreu, and M. P. Monteiro, “Reducing Subjectivity in Code Smell Detection: Experimenting with the Long Method,” in *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology*, Springer, October 2010.