Refactoring Embedded Software: A Study in Healthcare Domain

Paraskevi Smiari[1], Stamatia Bibi[1], Apostolos Ampatzoglou[2], Elvira-Maria Arvanitou[2]

[1] Department of Electrical and Computer Engineering, University of Western Macedonia, Kozani, Greece

[2] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

psmiari@uowm.grx, sbibi@uowm.gr, a.ampatzoglou@uom.edu.gr, e.arvanitou@uom.edu.gr

Context: In embedded software industry, stakeholders usually promote run-time properties (e.g., performance, energy efficiency, etc.) as quality drivers, which in many cases leads to a compromise at the levels of design-time qualities (e.g., maintainability, reusability, etc.). Such a compromise does not come without a cost; since embedded systems need heavy maintenance cycles. To assure effective bug-fixing, shorten the time required for releasing updates, a refactoring of the software codebase needs to take place regularly.
Objective: This study aims to investigate how refactorings are applied in ES industry; and propose a systematic approach that can guide refactoring through a 3-step process for refactoring: (a) planning; (b) design; and (c) evaluation.
Method: The aforementioned goals were achieved by conducting a single case study in a company that develops medical applications for bio-impedance devices; and follows a rather systematic refactoring process in periodic timestamps. Three data collection approaches have been used: surveys, interviews (10 practitioners), and artifact analysis (51 refactoring activities).
Results: The results of the study suggest that: (a) *maintainability* and *reusability* are the design-time quality attributes that motivate the refactoring of Embedded Software (ES), with 30% of the participants considering them as of "*Very High*" importance; (b) the refactorings that are most frequently performed are "*Extract Method*", "*Replace Magic Number with Constant*" and "*Remove Parameter*". We note that the "Extract Method" refactoring has an applicability of more than over 80%; and (c) to evaluate the refactoring process engineers use *tools* producing *structural metrics, internal standards*, and *reviews*.
Conclusions: The outcomes of this study can be useful to both researchers and practitioners, in the sense that the former can focus their efforts on aspects that are meaningful to industry, whereas the latter are provided with a systematic refactoring process.

Keywords: Embedded Software, OO Refactorings, Maintenance, Software Evaluation, Reuse, Software Tools, Performance

1. Introduction

Embedded Software (ES) is gaining ground in the software industry as it is considered to be a critical component of embedded systems that enables the management, control and monitoring of devices [38]. Many companies are engaged in continuously upgrading and developing diverse families of embedded systems through sophisticated software to satisfy newly appearing application requirements [32]. A possible explanation for this, is the software's negligible replication cost and its greater flexibility compared to hardware, which makes it easier to change (e.g., due to the arrival of new requirements, run-time optimization activities, or bug-fixing). Thus, product development managers often allow for some software additions or changes late in the product development cycle to correct hardware problems or add new functionality [33].

As the software is enhanced, modified and adapted to new requirements, the code becomes more complex presenting deviations from its original design that lead to reduced internal quality. Therefore, there is an urgent need for intense maintenance activities that aim at preserving the initial quality of the ES code. Efficient maintenance is far from trivial; in the sense that maintenance is one of the most effort consuming activities in the software lifecycle (maintenance consumes 50 - 75% of the total time / effort budget of a typical software project [44]).

In the literature, one of the most established maintenance activities for improving internal software quality is the application of software refactorings. According to Fowler et al. [16] refactorings are defined as transformations that improve certain quality attributes, but do not affect the external behavior of the software [16]. In their seminal book on refactorings, Fowler et al. [16] describe more than 70 object-oriented (OO) refactoring techniques for resolving potential bad smells. The need for refactoring is even more urgent in ES, in the sense that their design-time quality attributes (e.g., maintainability, reusability, etc.) are often compromised in favor of run-time ones (e.g., performance, reliability, etc.) [14], since embedded systems should conform to several run-time constraints (e.g., execution time, energy consumption, limited memory, failure rate, etc.). Therefore, the ES domain is in need for refactoring techniques that will improve the design-time quality of the software, after its initial development, while preserving its functionality and run-time quality standards.

Although the applicability of OO refactorings [16] in "*traditional*" software engineering (i.e., non-ES) has already been studied [11, 25, 31], their relevance to the embedded software industry, has yet not received significant attention (see Section 2). Research has shown that object-orientation has many benefits to offer in embedded software development [13, 15, 21] (e.g., producing simpler and more modular designs to reduce development time and prototyping effort [2, 17, 43]). Therefore, the ES development domain is in need of techniques that can improve design-time quality attributes. By considering the aforementioned need, along with the popularity of refactoring as a solution to this problem, in this paper we *explore the use of refactoring in the context of ES as a mechanism for quality improvement*. The envisioned outcome of this work is the provision of a systematic process for applying refactorings in ES development. We note that in this paper as "*refactorings*" we refer solely to the ones introduced by Fowler et al. [16]; i.e., not to code transformations aiming to improve the run-time performance of the system, known as "*performance improvements*" [3]. Nevertheless, several refactorings might affect the performance of the system, as a side-effect, yielding for a trade-off analysis [1]. Therefore, any trade-offs between deign- and run-time qualities considered during refactoring design and evaluation falls within the scope of this work.

This study considers the refactoring process as an "*engineering cycle"* of design science [45]. According to Wieringa [45], every engineering problem can be treated as a 4-step process: (a) identifying the need and specifying the problem; (b) design the proposed solution; (c) evaluate the proposed solution; and (d) apply the solution. The first 3 steps can be mapped to the refactoring strategy of a company, whereas the last one on the application of refactoring per se. This study focuses on refactoring strategy, since the application of OO refactorings is straightforward and in many cases, even automated by IDEs. By mapping the engineering cycle to refactorings, the following steps are defined in the proposed refactoring strategy (Figure 1):

- ***Refactoring Planning***: In this step, the software engineers have understood that the codebase needs refactoring and has decided to apply the most beneficial ones. Thus, according to Haendler and Frysak [19] the software engineer needs to: (a) select the quality attributes that need to be improved by identifying possible problems; and (b) the sub-systems that need refactoring.
- ***Refactoring Design***: The software engineer selects which refactorings need to be applied. This decision is primarily driven by two factors: (a) the problems that the code-base suffers from; and the related refactoring possibilities [20] and (b) the quality attributes that have been targeted [24].
- ***Refactoring Evaluation***: The software engineer selects the success criteria that are used for evaluating the benefit from refactoring application [28].

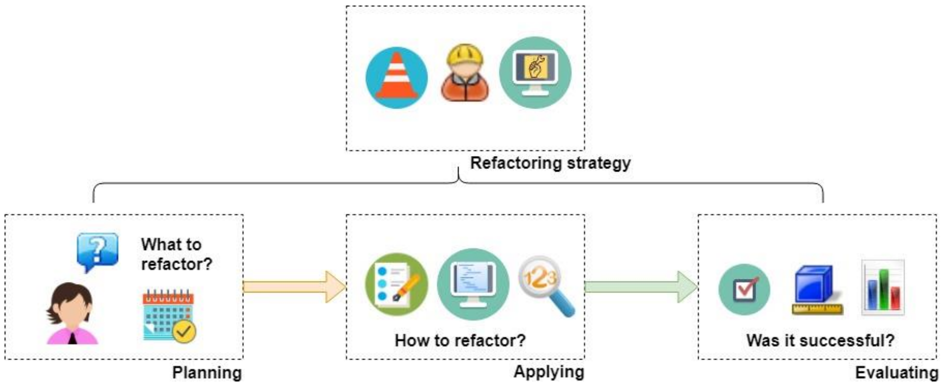


Figure 1. Refactoring Strategy

The goal of this paper is to propose a systematic refactoring strategy, based on empirical evidence and current industrial experiences. To achieve this goal, we have explored the state-of-practice in the embedded software industry, through a single-case study, in a company that already applies a rather systematic refactoring process. The rest of the paper is organized as follows: Section 2 presents background information and related work, whereas Section 3 the study design. The results are presented in Section 4 and discussed in Section 5. Finally, in Section 6 we present threats to validity, and we conclude the paper in Section 7.

2. Background Information and Related Work

In this section, the classification schema of quality attributes, used in this paper, is presented as background information (see Section 2.1). On the other hand, empirical studies that target the application of refactorings in the ES industry are presented in Section 2.2.

2.1 Quality Attributes Classification

The quality attributes that are considered for the purpose of this study, have been retrieved from Oliveira et al. [30], who reported, through a literature review, the main quality attributes of interest for embedded systems. In total 17 quality attributes are examined in this study as key motivators of ES refactoring that according to [30] are closely related to embedded systems. These attributes can be classified according to Bass et al. [22] into: (a) ***Run-time*** Quality Attributes (QAs), i.e., those that can be assessed as the system executes; and (b) ***Design- time*** QAs, i.e., those aspects that are related to the development of the system (not discernible at run-time). Regarding run-time QAs it is observed that ES are often used in a safe critical context; therefore, they must be *reliable*. Also, *security*, *safety*, *functionality*, *efficiency* (i.e., efficient consumption of hardware resources, such as processor, memory, and battery), and *portability* (i.e., ability of being transferred and used in a different environment) are identified as being important [30]. Other run-time QAs addressed are: *performance*, *usability* (i.e., ability of being understood, learned, configured, and used), *availability*, *fault tolerance*, *recoverability* (reparability) and *interoperability*. Among **design-time** QAs [30], we find *maintainability* (i.e., the ability of a system to preserve a successful state), *testability* (i.e., the ability of a system to support verification procedures), *extensibility* (i.e., the ability of a system to be extended in the future, *reusability* (i.e., the level at which a system can reuse its assets) and *flexibility* (i.e., the level at which a system can adapt to changes) as important for ES design.

2.2 Related Work

In this section we will present studies that have investigated the application of refactorings in an industrial context. The planning process of refactorings has been explored by [6, 10, 23], and [34]. Kim et al. [23] through a survey investigated the drivers that lead to a refactoring as well as the process and the analysis that is followed in order to choose the appropriate refactorings to be applied. A survey was also conducted by [10] to validate their approach. The authors mainly focused their research on ways to extract code clones for refactoring by combining clone metrics and consider them to be motivational for developers. Ribeiro and Travassos [34] also conducted an exploratory survey in an embedded software company and created and evaluated guidelines for writing code to achieve readability and understandability. To understand if the developers were doing actual refactoring or rewriting of code the authors formed research questions that focused on the meaning of code quality and when exactly was considered refactoring necessary. The construction of code guidelines was done by analyzing which attributes lead to readability and understandability of the code, how are they measured as well as their in between relationship. Andrade et al. [6] created guidelines to help engineers in the automotive domain to identify which refactorings to apply according to the existing architecture design. The authors created a framework consisting of a set of questions guiding the developers into making the right refactoring decisions according to their technical and architecture needs.

The process of applying refactorings has been thoroughly explored by [23, 26, 29, 39], and [40]. Kim et al. [23] investigated through the survey how software changes from refactorings were integrated and how was that knowledge passed to the rest of the developers. Also, the authors examined the usage of specific tools that enabled refactoring. Sharma et al. [39] conducted a survey as well on Siemens' architects and their main focus was to identify the challenges the architects were facing while adopting refactorings. A special emphasis in the survey was given in the level of satisfaction and the problems faced when adopting refactoring tools concluding that the improvement of refactoring tools is crucial. Murphy-Hill et al. [29] also focused their research on tools that enable refactoring. Specifically, they collected eight datasets, previously used in other studies, and analyzed them in order to investigate common assumptions in the domain of software refactorings. The authors mainly focused their analysis in the configuration of refactoring tools before their usage as well as the frequency that they are used and the difference of refactorings performed using tools and those being done manually. They further investigated the different use of refactorings, based on the expertise of the developers on the used refactoring tools, and the presence of refactorings in commit logs. Simons et al. [40] was mostly focused on quality and they conducted a survey on industrial practitioners from the Association of C and C++ Users and the British Computer Society to gather their results. They analyze the opinion of software engineers about Search-based Software Engineering refactorings and focused their research in the context of refactoring and whether or not metrics constitute a decent guide to achieve quality. The authors also investigated the reason why metrics are considered an intermediary of software quality as well as the formation of the correlation between metrics and software quality. The quality attributes that they mainly focus on are reusability, flexibility, and understandability. Understandability was also a key motivator in the research conducted on refactoring embedded software by Mooij et al. [26]. Among the findings of this study is that refactoring is helpful when it comes to model-based rejuvenation, making the whole process more controllable. The proposed refactoring application technique consists of defining the refactoring language, as well as, the refactoring operations to reduce code complexity.

Finally, the evaluation process of refactorings has been explored by several studies [23, 27, 28, 37, 41] by comparing quality metrics before and after the application of a refactoring. Moser et al. [28] used two sets of metrics one measures at a method level and the other at a class level. For the ones measured at a method level they used McCabe's cyclomatic complexity and the number of Java statements. For the ones measured at a class level they used the Chidamber and Kemerer object-oriented metrics. To evaluate the refactoring, they observed the changes made in specific classes that were likely to contain reuse and compared those changes. Moser et al. [27] also used Chidamber and Kemerer object-oriented metrics and specifically complexity, coupling, and cohesion metrics. They compared the results before and after the refactoring was applied in order to assess the impact of it. The results of the study showed an increase of productivity while refactoring showed evidence that it prevents the increase of complexity and coupling leading to a more maintainable code. Furthermore, Szőke et al. [41] focused their research on one quality attribute, maintainability. By using the Columbus Quality Model, they measured the maintainability before and after the application of refactorings which indicated that extensive refactoring periods have a very positive effect to the specific quality attribute. Kim et al. [23] briefly evaluated the refactoring by examining version history data and focusing the refactoring impact by measuring reduction in dependencies and defects. Schuts et al. [37] focused their research in legacy code refactorings in an embedded software at Philips. The authors applied model learning in both legacy and refactored code and compared the results with a model checker. The challenge was that the refactored code had to have the same behavior as the legacy code and model learning made that feasible.

The scope and the goal of the aforementioned studies is summarized in Table 1, along with their mapping to the 3 steps of the envisioned refactoring strategy. From the Planning phase through the Evaluation phase there has been extensive research around the area of refactoring, as examined above. Most of the studies focus on a specific area and do not explore the refactoring process as a whole. Andrade et al. [6], Choi et al. [10], and Ribeiro and Travassos [34] focused their research only in the Planning phase of the refactoring strategy by either creating code guidelines [34] and frameworks [6] or identifying code clones [10]. Mooij et al [26], Murphy-Hill et al. [29], Sharma et al. [39] and Simons et al. [40] specifically focused on the Design phase by exploring tools as refactoring enablers [29, 39]; or used certain quality attributes (such as Understandability) as drivers for refactoring [26, 40]. The Evaluation phase of the refactoring strategy has gotten more attention by Moser et al. [27, 28], Schuts et al. [37] and Szőke et al. [41]. In these studies, the researchers evaluated the applied refactorings by performing a comparison before and after the application of the refactorings. The gap identified by the current literature on software refactoring is that most of the studies do not approach the refactoring process as a whole. Though Kim et al. [23] through their analysis have explored all of the refactoring steps, their work appears to have a gap in the Embedded Software industry, whereas Andrade et al. [6], Mooij et al. [26], Ribeiro and Travassos [34] and Schuts et al. [37] have made progress in that area although their focus is only on one refactoring phase. Our work will be targeting the Embedded Software Industry as we have observed that the research activity is quite limited. Thus, the main points of differentiation of this study compared to the state-of-the-art are: (a) the focus in the Embedded Software industry, and (b) the comprehensive analysis of all refactoring steps in a single study.

Table 1. Comparison to Related Work

Study	**Envision Refactoring Strategy Step**			**Embedded Software**
	Planning	**Design**	**Evaluation**	
[6]		X		X
[10]	X			
[23]	X	X	X	
[26]			X	X
[27]			X	
[28]			X	
[29]		X		
[34]	X			X
[37]			X	X
[39]		X		
[40]		X		
[41]			X	
Our study	X	X	X	X

3. Case Study Design

In this section, we present the design of the performed case study. The case study is designed and reported following the guidelines provided by Runeson et al. [36]. Therefore, based on the linear-analytic structure, we first elaborate on the derived research questions, then we present the sample of the study (i.e., the selected case and units of analysis). Finally, we discuss data collection and analysis methods applied per research question. We remind that the high-level (HL) goal of this study is to *understand a structured refactorings processes for ES and to propose a systematic refactoring strategy*.

3.1 Objectives & Research Questions

The aforementioned HL goal can be refined, based on the Goal Question Metrics formulation [8], as follows: "*analyse* refactoring practices *for the purpose of* understanding *with respect to planning*, designing, and evaluating refactorings, *from the point of view of* software engineers *in the context of* embedded software development". According to the aforementioned goal we have derived three research questions (RQ), based on the envisioned refactoring strategy (see Section 1) that will guide the case study design and reporting of the results.

RQ_1: How do practitioners plan refactoring?

Through this research questions, we first attempt to explore *which quality attributes drive the refactoring process* ($RQ_{1.1}$). In this context we record the most common quality attributes (as defined in Section 2.1) that refactorings aim to improve. Second, we explore *how the stakeholders identify the spots of the system that needs refactoring* ($RQ_{1.2}$). $RQ_{1.2}$ intends to examine the practices that are employed in order to identify the spots that are candidates for refactorings and the process adopted for selecting the order of refactorings to be applied.

RQ_2: How do practitioners design refactoring?

Through this research question, we explore *which refactorings are most commonly applied* in the ES industry ($RQ_{2.1}$). For this purpose, we consider the documented refactorings as defined by Fowler et al. [16]. Second, through $RQ_{2.2}$, we explore *which quality attributes are affected most by refactorings*. Based on $RQ_{2.2}$ we examine the relationship between the types of refactoring and the quality attributes (both design- or run-time QAs) that they are considered to affect, as drivers for the refactoring or as side-effects.

RQ_3: Which evaluation methods are used to assess the effect of refactorings?

This research question aims to record the different evaluation methods that are used to assess the effect of the refactoring application, with respect to the quality attribute that the refactoring is considered to improve or indirectly affect. Given the fact that refactoring is by definition improving design-time QAs, emphasis is placed upon the identification of cases in which the refactoring negatively affects run-time qualities.

3.2 Case Selection and Units of Analysis

The single ***case*** of our study is the refactoring process of ImpediMed, i.e., an Australian large-scale software enterprise that specializes in the development of bio-impedance devices focusing mainly on medical applications in the fluid status area. This study has been conducted in the Thessaloniki branch of ImpediMed, which is mainly involved with the development of the software and periodically applies refactorings. The refactoring activities performed by the company can fall into two categories: (a) *immediate refactorings* that are performed as soon as a highly critical issue is identified; and (b) *organized refactorings* that are performed in scheduled time periods, usually before implementing a new release of the application. The system under analysis is SOZO: a physical medical device used for fluid and tissue analysis. SOZO includes hand and foot plates that obtain measurements, monitoring the condition of the patient. The device is controlled by an Android tablet/iPad (SOZOapp), which is paired with the device via Bluetooth. The data gathered by the device are transferred through the app and stored in the cloud (MySOZO). The ***units***

of analysis for our work are the participants of the study: software engineers. As participants we selected experienced software engineers (more than 5 years in the specific company), who are actively involved in the refactoring process. The participants served different roles in the company: software analysis and design, mobile application development, web application development, database engineering and testing.

3.3 Data Collection

The data collection process is comprised of three methods (survey, interviews, and artifact analysis) aiming to achieve method triangulation for all research questions. A mapping between the research questions and data collection methods, as well as the duration of each data collection activity is presented in Table 2.

Table 2. Summary of Collection Methods per RQ

Collection method	RQs	Duration
Survey	RQ_1, RQ_2	45’
Semi structured interviews	RQ_1, RQ_2, RQ_3	30’
Analysis of work artifacts in plenary	RQ_3	40’

Below, we discuss in detail each data collection method, and how it was applied for the purpose of our study. All data collection instruments are available in Appendices B and C.

- ***Survey***: Each participant was provided with an online questionnaire[1] focusing on the reasons for applying refactorings (RQ_1), the frequency that such refactorings took place (RQ_2), and their expected impact on quality attributes (RQ_1). The QAs that have been used are described in Section 2.1.
- ***Semi-Structured Interviews***: Next, the participants were interviewed for discussing the aforementioned topics, this time in the form of open-ended question. In these interviews, an additional subject was discussed, i.e., the way the effect of refactorings is evaluated in practice (RQ_3).
- ***Analysis of Work Artifacts in Plenary***: On the completion of the interviews several records of the refactoring logs[2] have been analyzed, so as to explore concrete examples of already applied refactorings. We discussed these refactorings in plenary with all the participants, especially regarding the measurement of success criteria (RQ_3), which is a technical question that might have been left answered from the previous data collection sessions. The discussions in plenary did not bias the opinion of the participants, since it was conducted as the final session.

3.4 Data Analysis

The data obtained from the survey, the interviews and the analysis of the work artifacts were analyzed with a combination of quantitative and qualitative methods. The responses to the open-ended questions and to the interviews were analyzed using Qualitative Content Analysis (QCA) [12], which is a research method

1 https://docs.google.com/forms/d/e/1FAIpQLSf-OhIvvGQyjM15bz-vlyWKTUknQAHXOU3OyARc7kRdd8lnFA/viewform?usp=sf_link

2 The refactoring log is an artifact held in ImpediMed for keeping a history of refactoring activities. Each record describes the applied refactoring, the targeted quality attribute, and the used evaluation method.

for the subjective interpretation of the content of text data through the systematic classification process of coding and identifying themes or patterns. We followed an inductive approach, where theories are proposed towards the end of the research process as a result of observations. This process involved open coding, creating categories, and abstraction. The codes that were found along with their classification to categories and then the abstraction on the three refactoring steps (planning, designing and evaluating) are presented in Appendix A. Initially we transcribed the audio file from the interview and analyzed it along with the notes we kept during the interview. We ensured that the questions and topics of conversation in our interview covered the whole process of conducting refactorings and consequently helping us answer RQ_1, RQ_2, and RQ_3. This information was valuable since through the interviews the participants could justify their answers in the questionnaire providing greater clarity and completeness. Finally, through the analysis of work artifacts we were able to have a clearer view of recent refactorings, the QA they aimed, and the way they evaluated that refactoring, and created a dataset that helped us answer RQ_3. For enabling the replication of the study all data collected, including the results of QCA are made publicly available[3].

The responses to closed-ended questions, on the other hand, were analysed following quantitative analysis methods, including data visualization and statistical analysis [46]. More specifically, descriptive statistics, including frequencies and percentages, were used to describe the characteristics of the sample, and the relationships between the variables. First, we gathered all the answers from the questionnaire in a table of 164 columns, including the role of the respondent and 10 rows (one for each response). The columns include the frequency of the specific refactoring type as well as the quality attribute that the respondents considered that it affects. Table 3 presents an overview of the data analysis methods used. In the column "Instrument Item" we refer to the questions posed at the participants. For example, Q_{1B} refers to Question 1 presented in Appendix B, Q_{5C} refers to Question 5 presented in Appendix C.

Table 3. Data Analysis Overview

Question	Used Data-points		
	Collection Method	**Instrument Item**	**Analysis Method**
$RQ_{1.1}$	Questionnaire	Q_{1B}, Q_{2B}	Pie chart, Stacked Bar chart
	Interviews	Q_{5C}, Q_{6C}	QCA
$RQ_{1.2}$	Questionnaire	Q_{3B}	QCA
	Interviews	Q_{1C}, Q_{2C}, Q_{3C}, Q_{4C}	Flow chart
$RQ_{2.1}$	Questionnaire	Q_{4B} – Q_{10B}	Stacked Bar chart
	Interviews	Q_{5C}, Q_{6C}	QCA
	Analysis of Work	Q_{9C}	QCA
$RQ_{2.2}$	Questionnaire	Q_{3B} – Q_{10B}	Radar charts
	Interviews	Q_{5C}, Q_{6C}	QCA

[3] The data can be downloaded from the following link:
https://www.dropbox.com/sh/fzakcszlhphkhrn/AAAf570i43U-ZKaBSjYSmKVpa?dl=0

Question	Used Data-points		
	Collection Method	Instrument Item	Analysis Method
RQ_3	Interviews	Q_{7C}, Q_{8C}	Venn chart, Stacked Bar chart
	Analysis of Work	Q_{9C}	QCA

For all RQs we analyse the interview questions (see Table 3) by performing QCA. Additionally, for $RQ_{1.1}$ we present the type of QAs that the refactoring process targets at, in the form of a pie chart and for each particular QA we provide a stacked bar that shows the level of its importance. For $RQ_{1.2}$ we describe the process of identifying and prioritizing the spots that need refactoring in the form of a flow chart. In $RQ_{2.1}$ we present the most commonly applied refactorings in the form of stacked bars and for $RQ_{2.2}$ we employ radar charts to present the applied refactorings with respect to the QA they intend to improve. For RQ_3 we present the methods used to evaluate the refactorings applied in the form of a Venn chart to identify methods are used in combination. Also, we use stacked bars to present the frequency of each evaluation method.

4. Results

In this section we report our findings organized by research question. First, we present the QAs that drive the refactoring process along with the procedures followed to identify the spots for improvement (RQ_1). Subsequently, we discuss the most frequently applied refactorings accompanied by the QAs they expect to improve (RQ_2). Last, we focus on the evaluation methods for assessing the success of refactorings (RQ_3). When discussing the results of each RQ we also enumerate the major finding and point out the codes identified by the QCA analysis is capital letters. We note that in a parenthesis we refer to the used item of the data collection instrument, as appearing in the appendix, i.e., (Q_{1B}) refers to Question 1 of Appendix B.

4.1 How do practitioners plan refactoring (RQ_1)?

QAs considered in the refactoring process ($RQ_{1.1}$): In this section we present the design-time quality attributes that drive the refactoring process of ImpediMed. Overall, the engineering team considers that the improvement of design-time QAs is very important (90% of the respondents rated the necessity of refactoring as "*Very High*" and 10% as "*High*") (Q_{1B}). Figure 2 presents the results regarding the type of quality improvement that the process is usually targeting (Q_{2B}). For each quality attribute we can see the percentage of participants that consider this attribute as being important or not: e.g., through the Maintainability bar we can observe that around 30% of the respondents consider this attribute being of "*Very High*" (30%) or "*High*" (70%) importance.

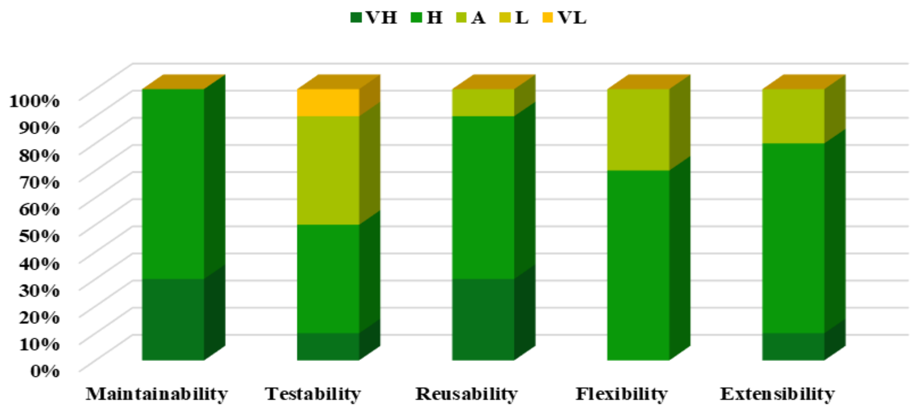


Figure 2. Refactoring quality attribute drivers

In the interview questions (Q5C, Q6C), respondents were asked to describe a refactoring that they have recently performed and the quality attribute the consider, while refactoring. Five codes emerged as the most prominent ones after applying QCA to the responses.

- The three of the most frequent codes are: RUNTIME ATTRIBUTES ARE DEALED EARLY, MAINTANABILITY IS THE MOST IMPORTANT FACTOR and REUSE. These codes seem to be related in the sense that the former could be reinforcing the latter, since placing emphasis upon run-time attributes may leave design-time attributes unattended and handled in a later time through refactorings. Nevertheless, any improvement of design-time quality attributes must not affect or compromise run-time performance. ***Run-time performance should not be affected in any case by refactoring, since it should confront to the strict perceptions and the restrictions posed by third-parties such as the client, the end-user and the specific business requirements / standards related to the legislation governing Health systems along with the operating environment of the application (Finding 1)***. With respect to maintainability, as stated by the software director: "MAINTANABILITY IS THE MOST IMPORTANT FACTOR *that drives the refactoring process is the improvement of maintainability. This task is usually the most time- consuming and challenging task since every* ***internal engineering team has many suggestions for improvement in that scope (Finding 2)****. REUSE is also very important since there are parts of code that can be used as libraries in subsequent releases or similar products"*. This finding complies with the results of Figure 2, Maintainability and Reusability are considered as key motivators for applying refactorings.
- The next two codes identified are: PREPARE FOR IMPLENTING THE NEXT RELEASE and IMPROVE TESTING PROCEDURES. The first is related with *Extendibility* quality attribute while the second with *Testability*. The results of QCA are also verified by the quantitative results. *Testability* is given a high priority by 40% of the respondents. A tester mentioned that "*We often need to IMPROVE TESTING PROCEDURES. Since the application is highly critical, we often need to exhaustively test every code path in order to achieve high code coverage. In this context not all code*

structures can be testable. We often need to change the structure of a class so as to be able to test it". *Extensibility* is also a key refactoring driver since 70% of the participants recognized that it is of "High" importance. It was mentioned by a developer that "*One of our refactoring goals is to change parts of the current version of the application in order to be able to support the upcoming requirements and* PREPARE FOR IMPLENTING THE NEXT RELEASE".

Maintainability and Reusability are significant driving factors for refactoring. Though any refactorings made for improving design-time QAs should not affect Run-time QAs since these are strictly tied to third-party restrictions, (i.e health regulations) and therefore are most of the times obligatory.

Process to Identify refactorings ($RQ_{1.2}$): Next, we discuss the process employed by ImpediMed for identifying the spots in the source code that need improvement. To answer this research sub-question, we used information from the survey question Q_{3B} as well as from the interview questions Q_{1C} to Q_{4C}. QCA identified three frequent codes: TIME REQUIRED, CODE READABILITY and ORGANIZED REFACTORINGS. The process employed by ImpediMed to plan refactoring is presented in Figure 3. In Figure 3, we observe that refactorings targeting design-time QAs (ORGANIZED REFACTORING)[4] are planned by the head department and are usually scheduled before implementing the subsequent release of the application. Organized refactorings may last from two weeks to two months depending: (a) on the time constraints posed on the new release, and (b) the criticality and the volume of the issues identified during the operational period of the current release. The process for organized refactorings is ad-hoc, since the majority of decisions are made by the Software Director. **Initially each Team Leader, after discussing with the team, provides a list of the suggested refactorings based mostly on his experience and the metrics provided by the static analysis tools (Finding 3). Then the Team Leaders and the Software Director make a discussion regarding the candidate spots for refactoring (Finding 4).**

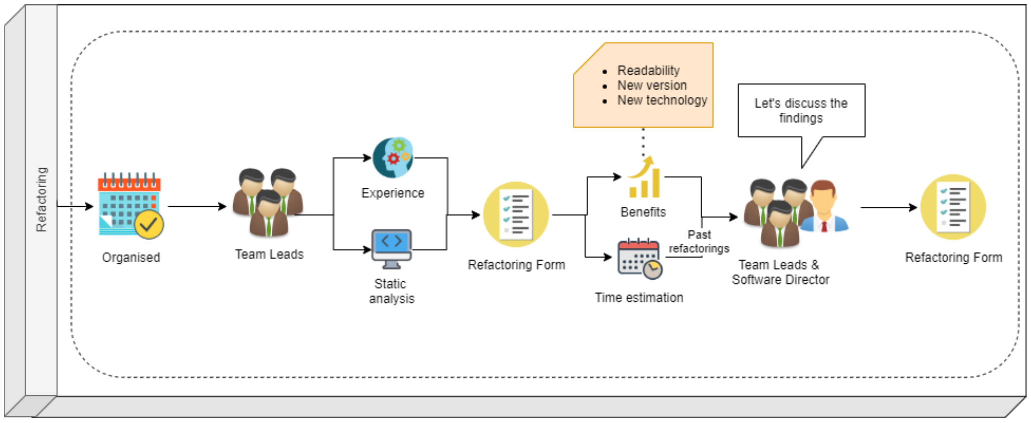


Figure 3. **Process employed by ImpediMed to identify spots for refactorings**

[4] We note that the company terminology also includes the term "IMMEDIATE REFACTORING" which refers to *"performance improvements"* (see Section 1) that are handled immediately as they appear.

The software director makes the final decision considering: (a) the TIME REQUIRED to refactor, and (b) the potential benefits acquired based on the goals that the company has set (Finding 5). The estimated time to refactor is approached by the software director, based mostly on data from previous refactorings. Regarding the benefits acquired from each refactoring the company has set three distinct goals:

- The first goal, in order of importance, is to improve the CODE READABILITY of the source code since the company is in the process of hiring new employees. As it was mentioned by the software director "*The code should be in a state that allows for a newly hired employee to understand, learn and adapt within a few months*".
- The second goal is to PREPARE FOR IMPLEMENTING THE NEXT RELEASE The general opinion of the team is that "*after every release the code needs to be highly maintainable, extensible, flexible, and reusable so that the next development cycle won't inherit any flaws from the previously released version*".
- The last goal of refactoring is to improve the quality of the end-product by the USE OF NEW TECHNOLOGIES: *"It is important to keep pace with advance in Software development CASE tools".*

The process of identifying Design-time QAs that need refactoring is semi- planned, leaving most of the decisions to the Software Director. *Code readability*, *Use of new technologies* and *Code preparation* for the next release are some of the goals when targeted by refactoring.

4.2 How do practitioners design refactoring (RQ$_2$)?

Commonly Applied Refactorings (RQ$_{2.1}$): The next phase after planning the refactorings is designing how to apply them. Software engineers need to identify which refactorings to apply to improve the quality attributes mentioned in RQ$_1$. To answer this RQ we used information from survey questions Q$_{4B}$ to Q$_{10B}$, interview questions Q$_{5C}$ to Q$_{6C}$, and information from the analysis of work databases (Q$_{9C}$). As mentioned by software director: ***"The selection of the refactorings that need to be applied is based on the engineers' experience, previous refactorings and the suggestions of Lint tools. We still do not have any formal procedures to record data produced during the refactoring process and this is something we need to work on"* (Finding 6)**. According to Figure 4 the top-rated refactorings are presented, with respect to the type of changes performed as described by Fowler et al. [16]. We selected to present the top-20 applied refactorings that represent over 80% of the refactorings performed during the most recent planned refactoring of the company. Among the most prominent codes identified by QCA are the EXTRACT METHOD and the REPLACE MAGIC NUMBER WITH SYMBOLIC CONSTANT.

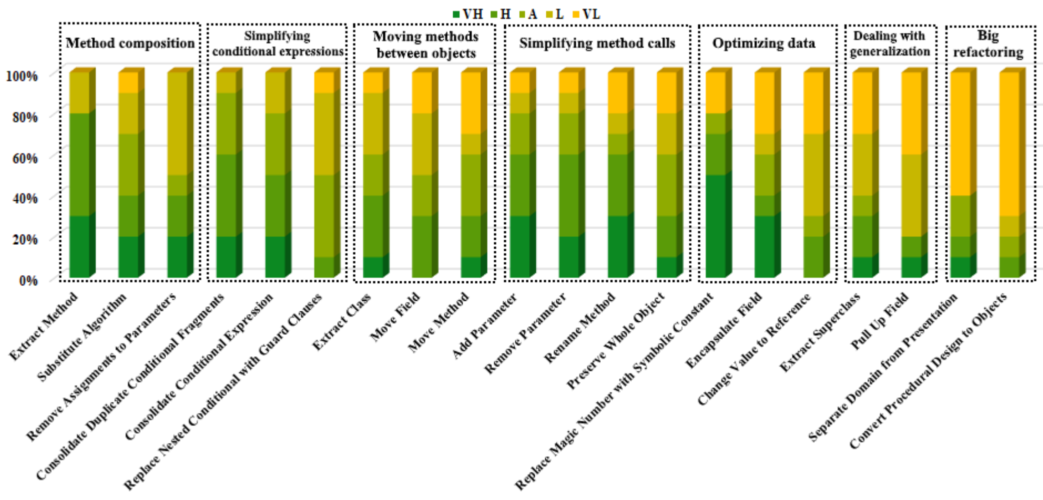
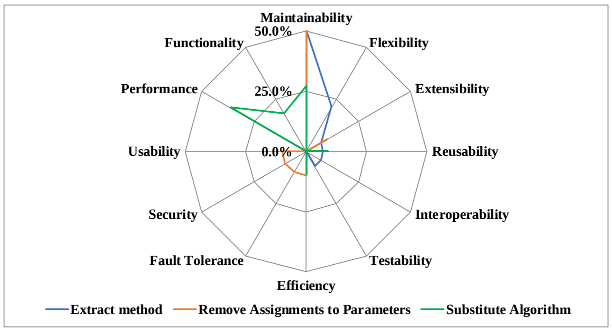


Figure 4. Frequency of Refactorings

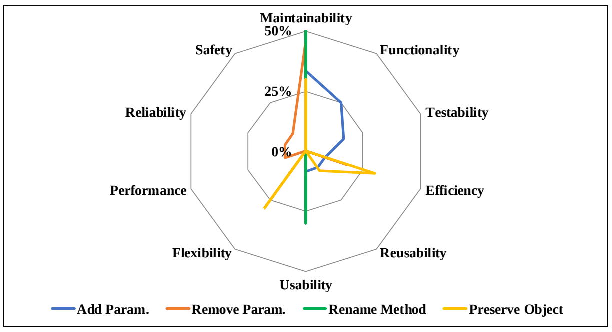
The most frequently applied refactoring, by almost all team members, is the EXTRACT METHOD. A front-end developer mentioned: "*We had several views that contained the same logic, in various parts of the code, so we extracted them into a separate module*", similarly a tester stated: "*During testing we try to simulate a process but the functionality in some methods is very complex to be simulated. In that case we reduce this complexity by extracting specific functionalities into separate methods.*" Additionally, a mobile developer mentioned: "*I initially implemented a process, into one method, that contained several steps. After some while I had to return to this method and apply a small change. I realized I forgot the rationale behind the implementation of this method, a fact that leaded to time delays. At the subsequent refactoring I extracted the steps of this process into separate methods in order increase the readability of the code*".
EXTRACT CLASS refactoring was also applied in the last refactoring session. The Software Director mentioned "*In the backend part of the application a big change appeared as a necessity in order to make the code architecture clearer. That change was to extract all the main functionality components of the system, implemented by the controllers, into different services (implemented as classes). After this change the services implemented in the new classes are separated by the controllers who now just make calls to them. "Consolidate Conditional Expression"* as well as *"Consolidate Duplicate Conditional Fragments"* refactorings appear to be highly applied. **These types of refactorings are appointed by the inspection tool (Lint tool) the programming IDE provides based on the code smells identified (Finding 7).** As static analysis of the code based on Lint is one of the main methods adopted by the company to spot code deficiencies, these types of refactorings are often applied.

In Figure 4 we can observe that over 50% of the participants apply the REPLACE MAGIC NUMBER WITH SYMBOLIC CONSTANT and the REMOVE PARAMETER refactorings. The high usage of these refactoring types is reasonable if we consider the statement of a database developer who specifically said "*We had some fields in the database that were defined to support hypothetical future requirements. Those requirements actually never came, leaving us with unused fields*". Additionally, *"Replace magic number with symbolic constant refactoring is highly applied as among others it is considered to be mandatory by the* *internal company rules*". On the other hand, refactorings related to the categories DEALING WITH THE GENERALIZATION and BIG REFACTORINGS are rarely applied. Overall, we observe that method-level, small-scale refactorings are preferred compared to big refactorings. The low frequency of the later is justified by the development process of the company, which emphasizes in the design phase, when detailed class diagrams are designed. Therefore, there is little space for high-level and large-scale refactorings. This is also supported by the fact that refactorings are more frequently applied at method-level, instead of class-level. This finding suggests that it is easier for the developers to specify (in the design phase) the classes through which they will structure the source code, instead of their methods and functionalities. Additionally, developers seem to prefer method-level refactorings, since usually they: (a) are small scale, (b) easy to apply, (c) require limited time, and (d) they present a lower chance of leading to "code breaks", compared to big refactorings. Method-level, small-scale refactorings are preferred compared to big refactorings. EXTRACT METHOD and the REPLACE MAGIC NUMBER WITH SYMBOLIC CONSTANT are the most applied refactorings. The experience of the engineer along with the suggestions of tools (i.e., Lint tools) drive the process of identifying the candidate spots for refactoring.

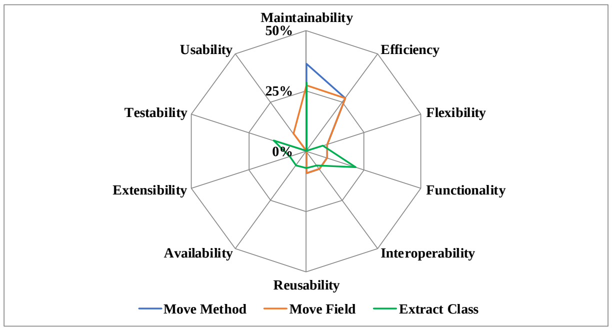
QA Affected by Refactorings ($RQ_{2.2}$): In this research sub-question the developers were asked to associate the different types of refactorings applied to the specific quality attributes they improve (survey questions Q_{3B}-Q_{10B} and interview questions Q_{5C}, Q_{6C}). As noted in Section 1, for answering this research question, we include in our analysis run-time quality attributes, since the application of refactoring may be subject to quality trade-offs. QCA results revealed two frequent codes IMPROVE MAINTANABILITY, IMPROVE MODULARITY. As it can be observed in Figure 5 the quantitative results show that the primary goal while refactoring is to build a system that will be *Maintainable* as well as *Efficient*.

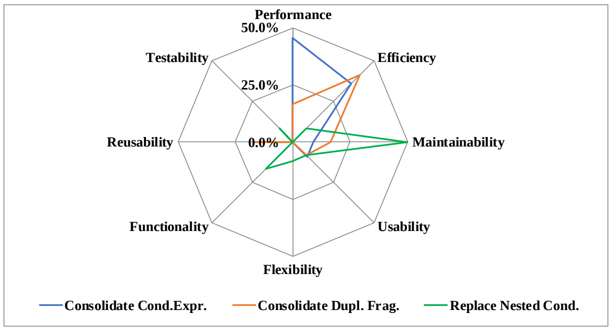


(a) Method composition refactoring



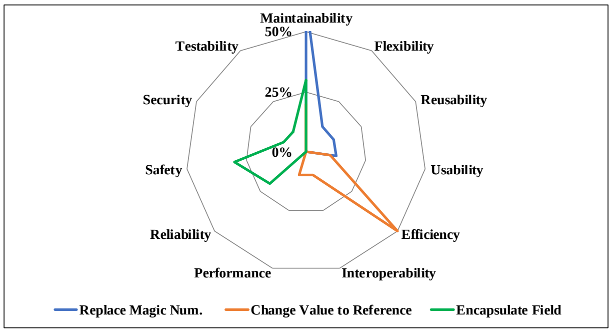
(b) Simplifying conditional expression

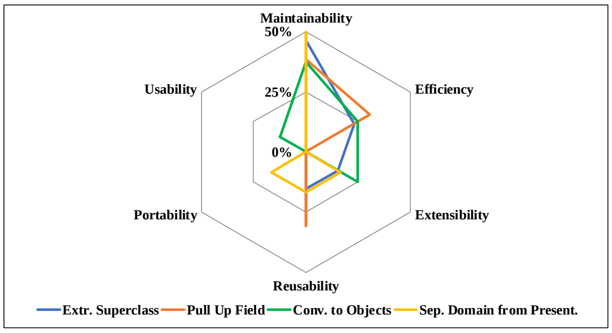




(c) Moving methods between Objects

(d) Simplifying method calls





(e) Optimizing data

(f) Dealing with generalization & Big Refactoring

Figure 5. Quality Attributes targeted by refactorings.

From Figure 5 we can draw some useful conclusions:

- When focusing to refactorings that aim at MAINTANABILITY, we can observe that the refactoring category METHOD COMPOSITION is associated with systems that are more maintainable. More specifically the refactorings EXTRACT METHOD and REMOVE ASSIGNMENTS TO PARAMETERS are highly associated with maintainability.
- Refactorings that simplify method calls, such as CONSOLIDATE CONDITIONAL EXPRESSION and CONSOLIDATE DUPLICATE CONDITIONAL FRAGMENTS are highly associated with PERFORMANCE and efficiency.
- Furthermore, the refactorings that deal with generalization, such as EXTRACT SUPERCLASS and PULL UP FIELD are highly associated with reusability.

The refactorings performed related to METHOD COMPOSITION improved Maintainability, refactorings related to simplifying method calls improved the Performance of the system while refactorings related to generalization improved the Reusability.

4.3 How do practitioners evaluate refactoring (RQ$_3$)?

In this research question we discuss the methods employed by ImpediMed to evaluate the impact of refactorings with respect to the quality attributes they intend to improve. To answer RQ$_3$ we analyzed interview questions Q$_{C7}$, Q$_{C8}$, Q$_{C9}$ and used information from the work artifacts with the most recently applied refactorings. Based on the interviews we observed that engineers focus the refactoring evaluation on run-time QAs. This finding is intuitive since:

- The refactoring is by definition improving design-time QAs and therefore this viewpoint of quality is taken for granted by the software engineers.
- The compromise of run-time QAs is non-negotiable in ES (as mentioned in Section 1) engineers perform exhaustive evaluation of the refactoring with respect to run-time qualities (see Finding 1).

Therefore, the answer to RQ$_3$ is naturally build around run-time QAs. Figure 6 presents the evaluation methods used in the company to assess the impact of refactorings. **In total four evaluation methods are**

referenced by the development team based on QCA: the use of WIKI RULES, the use of automated TOOLS such as the LINT TOOL and the INTERNAL COMPANY TOOL, the use of TEST CASES and the REVIEWS (Finding 8).

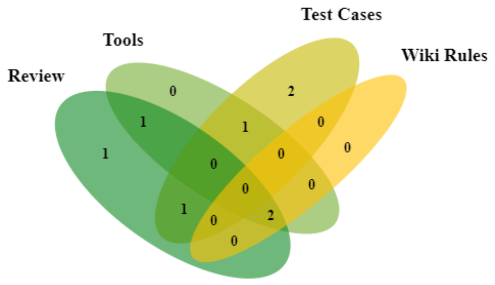


Figure 6. Evaluation methods usage

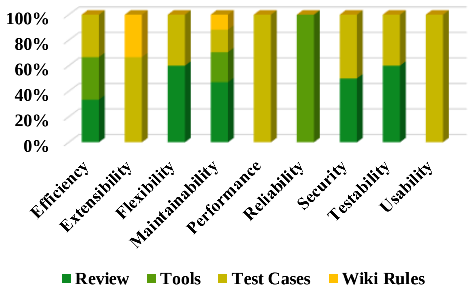


Figure 7. Evaluation methods per quality attributes

Additionally, Figure 7 presents the method employed to evaluate the impact of refactoring with respect to the quality attribute that they affect. According to the engineers the methods that are mostly preferred to evaluate the effect of Maintainability are: (a) 47.1% through code Reviews, (b) 23.5% through the usage of Tools, (c) 17.6% by utilizing Test Cases, and (d) 11.8% by taking advantage of the Wiki Rules, as explained below:

- REVIEW is the most popular method for evaluating the refactored code. A web developer mentioned that: "*When we refactor, especially when it comes to the UI, experience and thorough reviews is the only way to evaluate our changes and to make sure that nothing has broken*". Additionally, through reviews the software engineer also checks the readability of the code as well as the conformance of the new code to the styling conventions of the company.
- **TOOLS is the next most popular refactoring evaluation method to record the values of several structural metrics and to address PERFORMANCE indicators (Finding 9)**. It is important for the company to apply refactoring targeting at improving the code understandability but also to ensure that the overall performance of the system has not been compromised. The company uses both Lint tools and an internally developed company tool.
  - Lint tools provide insights regarding code metrics and are considered to be a supplementary method when evaluating the refactored code. According to an android developer: *"Lint checks are extremely useful for assessing refactorings in the web front-end part. Linters can help us record performance metrics related to memory consumption, thread deadlocks and bottlenecks and therefore correct any problems that arise during code refactoring"*. Though the Lint tools cannot be applied in any code artifact, for example there are no such tools to accommodate the needs of database development.
  - The internal tool is a validator tool for performance testing, it checks whether the application reaches the performance indicators. **PERFORMANCE indicators such as maximum time to login, maximum time of response etc. are set by the engineering team in cooperation with Business department and the client (Finding 10)**. The Internal

tool is exclusively used by the backend team in all refactoring cases, as it is obligatory by the company standards to validate code through this tool.

- TEST CASES are also very frequently used to validate the refactored code. It was stated by a database developer that "*When we refactor parts of the database transactions the evaluation is done through test cases to make sure nothing in the functionality has changed. We prefer to validate the correctness of the transactions performed in test case scenarios, and check the results in the client side instead of reviewing that the database is updated at the server side*".
- WIKI RULES are the least preferred evaluation method, but apprised by the software director. He mentioned that "*Thorough reviews against the Wiki Rules is among the things I consult during the evaluation of the refactorings. The Wiki rules for me guide the final reviews of the refactored code.*"

On the other hand, when the software director was asked whether the team recorded any refactoring process metrics (i.e., actual time to perform a refactoring, time saved when adding new functionalities due to refactoring, overall number of changes) he mentioned: **"At the moment we do not gather data related to process metrics. These data would actually be very useful, but it seems to me time-consuming to keep those meta-data manually. It would be useful to have a tool that would help us automatically record the changes performed during a refactoring and the code affected" (Finding 11).**

The preferred method for evaluating refactorings is the REVIEW of the refactored code. TOOLS are also used when validating Run-time QAs such as PERFORMANCE. The demand for a refactoring tool that will potentially automate the process is highlighted.

5. Discussion

5.1 Interpretation of Results

In this study we examined an existing ES industry refactoring process in terms of the: (a) QAs that drive refactoring; (b) most frequently applied refactoring accompanied by their impact on QAs; and (c) methods used to evaluate them. The main outcome of the study is illustrated in Figure 8. From Figure 8, we can observe that for improving Maintainability (the key motivator for applying refactorings) the preferred refactorings are: "*Extract Method*" (12%), "*Add Parameter*" (7%), "*Remove Parameter*" (8%), "*Remove Method*" (12%), "*Consolidate Conditional Expressions*" (2%), and "*Replaced Magic Number*" (8%). Those refactorings indicate the existence of specific code smells such as "*Feature Envy*", "*Long Method*", "*Duplicate Code*", "*Alternative Classes with Different Interfaces*", "*Speculative Generality*", as well as the code smell "*Magic Number*". Additionally, to validate the improvement of Maintainability the methods that can be used are: Code Reviews (47.1%), Tools (23.5%), Test Cases (17.6%) and Wiki rules (11.8%).

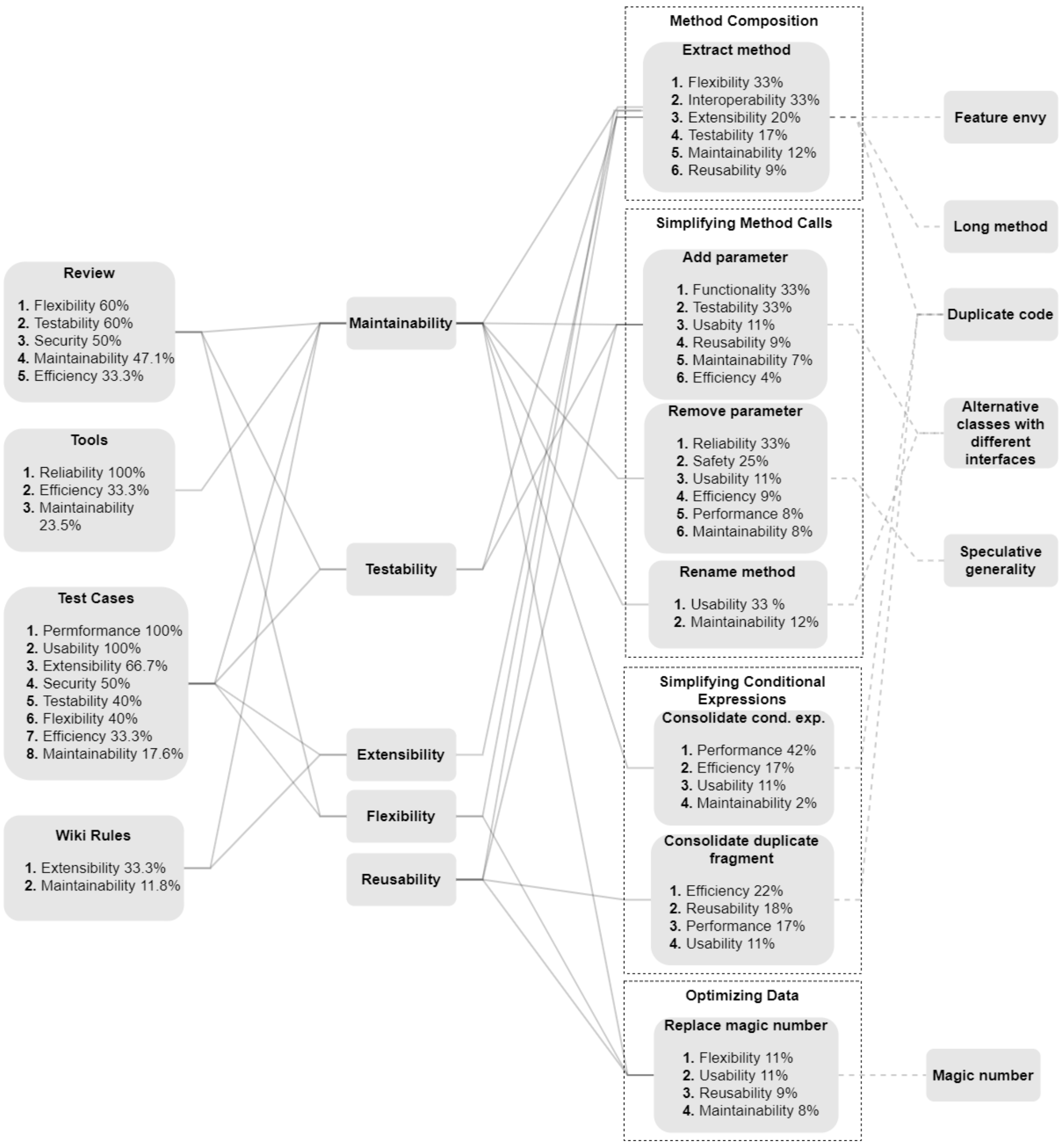


Figure 8. **Association between QA, code smells and Fowler's refactorings**

The results of our study have highlighted ***maintainability*** and ***reusability*** as the main the ***motivators for refactoring*** ES. The finding with respect to *maintainability* is in accordance with Kim et al. [23] and Ribeiro and Travassos [34], who also recognize maintainability as an important quality attribute that drives the refactoring process. Despite the fact that the implementation of ES presents several variations compared to the implementation of "traditional" software [14], it seems that in the perception of software engineers' maintainability remains an important quality attribute that needs to be monitored and preserved in certain levels through refactoring. *Reusability* was the next most important QA that drives the refactoring process in ES. This finding is contradictory to Lacerda et al. [18] and Kim et al. [23], who argued that reusability has a weaker relationship to the refactoring process compared to the other quality attributes. Though when focusing on ES, software reuse is both a challenge and a goal: a challenge due to the shortcomings of layered software [42], and a necessity due to the fact that software needs to be reused in the various product families of ES [32]. However, we need to note that the study of Kim et al. [23] considers reuse rather reusability, in the scope that the refactoring is motivated by repurposing existing code to be tailored so as to be executed in a different environment. In this study we consider reusability, as the ease with which existing code can be reapplied in a different occasion. With respect to the process of identifying spots for refactoring, we have assessed ***the refactoring process*** employed in the company as semi-organized, since it ***lacks the support of formal tools and methods to plan refactoring***. Despite the fact that in research literature we can find a variety of studies related to process models for refactoring [9, 19, 25] still in practice it seems that the existing body of knowledge about refactoring and automated tools is not exploited.

In addition to that, we discovered that ***most refactorings are applied in the method-level*** in an attempt to improve code readability and organization. The most frequently applied refactorings are related to the renaming of methods, the organization of parameters, the replacement of magic numbers which is are also appointed as popular refactorings in conventional software refactoring according to Murphy-Hill et al. [29]. Our results indicated that "*Extract Method*", "*Rename Method*" and "*Add/Delete*" types of refactoring have a great appeal to engineers. This finding is in accordance with the study of Murphy-Hill et al. [29] that has shown that "*Rename*" refactoring has very high application, and with the study of Kim et al. [23] where the "*Remove Parameter*" refactoring appeared to have the highest applicability. Moreover, based on the types of the refactorings applied we can conclude that the intention of software engineers in ES is not to perform large-scale refactoring (re-engineering or re-architecting) but rather to remove "code smells" to improve overall the system state and support feature additions, which is a common finding in several other studies [18, 20, 24].

Finally, regarding ***refactoring evaluation*** the targeted company uses ***reviews*** and ***test cases*** to ensure that the functionality remains stable after refactoring. Related research promotes the use of tools as an evaluation method, either to identify whether refactoring has inserted new bugs [37], or to calculate source code quality metrics [23]. The use of tools in this study is also pointed out as important for calculating source code metrics and detecting code smells. The participants though mentioned that they are willing to adapt more tools but do not have the "know-how" yet. Kim et al. [23], Mooij et al. [26] as well as Murphy-Hill et al. [29] have also highlighted the need of such refactoring tools. The evaluation of refactoring by examining process metrics such as the productivity of the development team [27] is not performed at all within the examined company. Participants though recognized the impact of refactoring in increasing the team productivity but thought that such an evaluation would increase the overhead of the team.

5.2 Implications to Researchers and Practitioners

This case study provides several implications to researchers and practitioners. On the one hand, regarding ***researchers***, our findings point out that as their future work they can focus on *refactorings related to the reusability of the source code*. Reusability in ES is considered to be very important and still challenging, due to the fact that software is closely related to the specificities of the hardware. Currently the refactorings found in literature are general purpose ones, related to object-oriented software. There is the need to define new types of refactorings that will handle the specificities of ES, i.e., to decouple business requirements from application requirements and remove constraints related to the operating environment to enable for reuse. Additionally, this study identified the need to *establish new approaches and tools for supporting all phases of the refactoring process of ES*. The ES industry seeks for automated tools that will be able to support in an IDE the assessment of: (a) code, (b) performance, and (c) process metrics. In this context researchers can work on newly addressed metrics related to ES that will correlate the impact of code refactorings to performance and security indicators. Additionally, there is the need for tools that support the application of complex refactoring operations that will allow for the traceability of refactorings, since as mentioned one change can cause chain effects to the rest of the code.

On the other hand, ***practitioners*** are advised to *follow an organized and well-documented refactoring process that will ease the application of refactorings, allow for the reuse of the refactored process and its continuous improvement*. For this reason, based on our findings we present a generic process that can guide the ES industry on performing quality improvement. The suggested process as presented in Figure 9, follows the design science engineering cycle, as explained in Section 1 and is detailed based on the findings of this study.

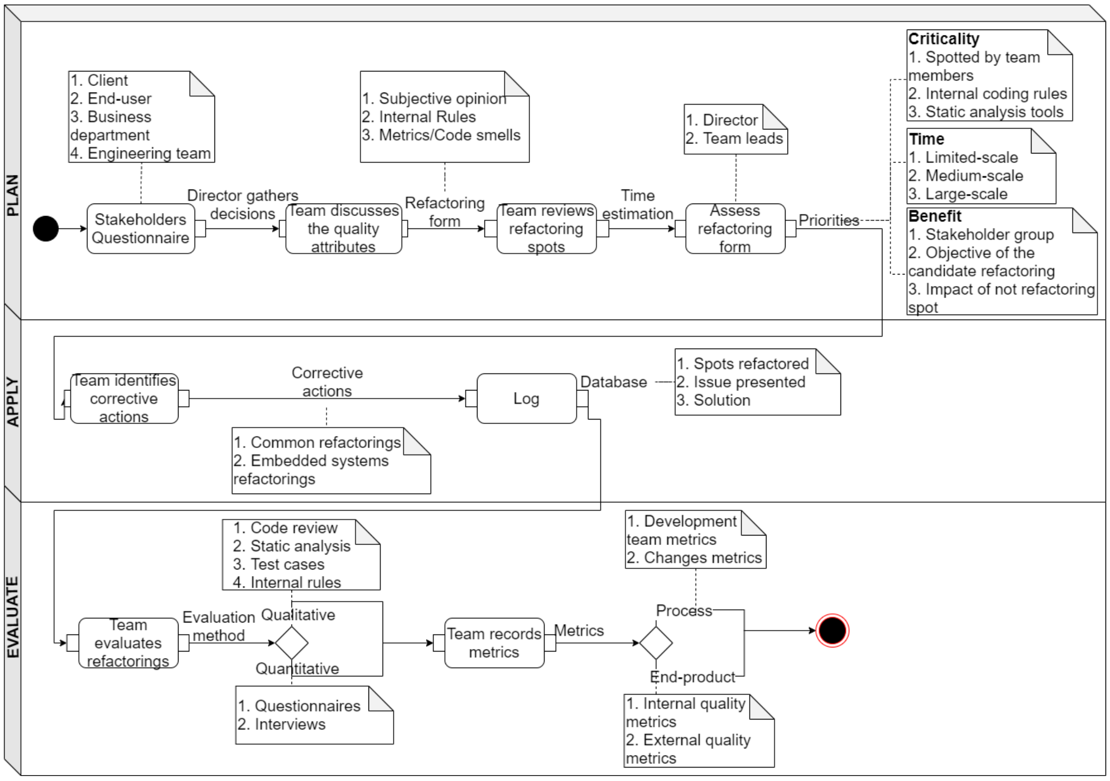


Figure 9. **Planning the refactoring process**

Plan Improvements: During the planning phase the software engineers need to focus on the quality attributes that need improvement and subsequently identify the spots that present flaws.

- The first step (***step 1.1***) is to decide upon the QAs that will drive the refactoring procedure. Such a decision regarding ES involves many stakeholders i.e., the client, the end-user, the business department and of course the software engineering team (**Findings 1, 2**). It includes the quality assessment of the application, based on the view and goals of each stakeholder. The assessment can provide an insight regarding the quality of experience of the end-user, the objectives of the client, the milestones of the business department and the difficulties that the engineering team faces. The software director will then have an overall view of the end-product and the quality attributes that can be improved.
- As a second step (***step 1.2***) of the planning phase, the software director circulates a list of the targeted QAs to each development team. Every team discusses internally the spots that affect the quality attributes **(Finding 2)**. For this purpose, team members record into a form the spots that need refactoring based: (a) on their subjective opinion; (b) the internal rules or globally accepted standards (i.e., Wiki rules, ISO standards, GDPR); and (c) metric values or code smells, as derived from static analysis tools **(Finding 3)**. The team leader collects and circulates the forms to all team members **(Finding 4)**. Then the team discusses the candidate spots to be refactored and estimates the refactoring time. Regarding the rationale that supports the estimation of the time required to apply a refactoring the team members can follow an agile approach. At a high level they review the spots that are candidates for refactoring and make an intuitive estimate of the time required to apply the refactorings. If the source code that needs to be refactored is complicated and requires many operations then it is decomposed into small spots whose refactoring can be better controlled and estimated. This process requires very good knowledge of the code; therefore, it is important to be performed at team level **(Finding 2)**.
- At the final step of this phase (***step 1.3***) the lead developers of every team and the software director discusses the findings of the previous steps. All candidate spots for refactoring are assigned a value in a three-scale system (LOW, MEDIUM, HIGH) based on three decision drivers: (a) their criticality, (b) the estimated time to refactor; and (c) the potential benefits acquired **(Finding 5)**. Regarding the ***criticality*** the candidate refactorings appointed by team members can be ordered first, then come the refactorings related to internal coding rules/standards compliance and lastly the refactorings spotted by static analysis tools. Regarding the ***time*** required for refactoring, the estimations performed in previous steps are used to classify candidate refactoring spots into limited-scale refactoring spots (those requiring less than 1 day), medium-scale refactoring spots (those requiring 2 – 5 days) and large-scale refactoring spots (those requiring more than 5 days). Regarding the ***benefits*** acquired from each refactoring for each candidate spot the decision can be based on the following information: the stakeholder group (s) that will benefit from the refactoring, the objective (as recorded is step 1.1) of the candidate refactoring with respect to the quality attribute it intends to improve and the impact of not refactoring the particular spot. At the end, the Software Director along with the team leads **(Finding 4)** assign values to each of the three decision drivers for each candidate refactoring spot. Then the candidate spots are ordered and the top ones that fit into the time period assigned for the refactoring process are selected.

Design Refactoring: During the design phase software engineers need to focus on the refactorings that need to be applied in order to improve the spots that present flaws.

- The first step (***step 2.1***) of this phase is to identify the corrective actions that can be employed to improve the refactoring spots. At this step, the engineers may apply both refactoring related to the special nature of embedded software [26] and common refactoring targeting at design-time quality systems. In the first case, the engineers are advised to follow standards and rules as imposed by the relevant regulations that rule the domain of the application **(Findings 1)**. For improving design-time attributes the engineers are advised to use common classifications of code smells / problems to refactoring solutions **(Finding 7)**. This approach is widely applied in industry and object-oriented software [24].
- This step (***step 2.2***) involves logging the process. It is important for the team to keep records regarding the spots that are refactored, the issue they presented and the solution that was applied **(Finding 6)**. The creation of a database containing common issues presented within a company along with the refactoring solutions can help towards preventing future repetition of the issues while it stores valuable knowledge that can be reused in future to refactor similar issues.

Evaluate Improvements: During the evaluation phase software engineers need to focus on the evaluation methods that need to be applied in order to assess the validity of the refactorings.

- The first step (***step 3.1***) of this phase is to evaluate the refactorings in terms of the QAs they are targeting to improve. Software engineers should identify a set of qualitative or quantitative methods that can help them towards that direction **(Finding 8)**. In this context qualitative methods may include questionnaires or interviews with the stakeholders to assure that the refactorings applied reached the objectives set in step 1.3. Reviews (or inspections) of the code, application of static analysis tools, test cases and compliance against rules / standards can form a set of quantitative methods that can be used for checking the effectiveness of the refactorings **(Findings 8, 9, 10)**.
- At this step (***step 3.2***) it is important to record metrics related to: (a) the refactored end-product **(Findings 9, 10)**, and (b) the refactoring process **(Finding 11)**. Regarding (a), internal (e.g., size, complexity, deadlocks) or external quality (number of operational bugs, response time, number of malfunctions) metrics can be used. Regarding (b), the refactoring process can be measured through development team metrics (e.g., number of engineers occupied in refactoring, the level of their experience), and change metrics (e.g., number of refactorings applied, number of changes made for each refactoring, time required for each refactoring) [23, 27].

6. Threats to Validity

In this section we present threats to the validity of this case study. These threats will be organized into construct, internal, and external validity, as well as reliability threats. Internal validity will not be applicable to this study, since in our research we do not examine causal relationships. For the mitigation of construct validity, which demonstrates if the conducted case study actually encompasses all of the research questions [36], we followed specific steps. In our data collection process, we established more than one data set in order to form data and method triangulation. With method and data triangulation we prevented the usage of one data source that would potentially cause misleading results. Another potential threat to construct validity would be the number of the participants. We believe that the threat is non-existent since we chose a diverse set of participants that included all the different software development roles of the company. The inclusion of the software director as a participant helped us have a broader perspective in our data set.

With respect to external validity, which concerns the generalization of our findings and the application of research in similar domains [36], we understand that it may seem challenging for other embedded software companies to agree with our findings. However, since ImpediMed is a well-established company in the embedded software and medical devices domain and our participants consisted of diverse roles with at least 2 years of experience we believe that the application of our method in similar domains will convey similar results. Finally, we note that the results of this study are not directly comparable to other studies that: (a) have a different definition of quality attributes, or (b) are performed in a different context.

Finally, regarding the reliability of the case study we made sure that the findings from the data collection and analysis process can be recreated [36]. To achieve that, we created the questionnaire online so that it can be accessed by other researchers who want to reproduce the results. Additionally, during our interviews we asked open ended questions along with a motivation for each answer. The questions asked during the interview are also provided online. The data analysis process was conducted by two researchers in order to avoid bias. Also, we made all data publicly available so as to enable the replication of the study[5].

7. Conclusions

Software refactoring has proved to be an effective technique to improve the overall quality of a software system. However, applying refactoring in embedded software is a challenging task since ES needs to comply with strict constraints during run-time operation (i.e., performance, security). Moreover, the embedded software domain is driven by cost and time-to-market factors, which also influence the refactoring decisions taken by software engineers in ES. This paper explores the refactoring strategy adopted by a company developing ES in the medical domain through a holistic industrial study. We analysed three sources of data (surveys, interviews and artifact analysis) in order to understand the strategy adopted by the company in order to plan, apply and evaluate refactoring. The results show that the refactoring strategy followed by the company is semi-organized, mostly driven by design-time quality attributes (100% of the respondents consider them of “High” importance) such as Maintainability and Reusability (30% of the participants considering them of “Very High” importance). The refactorings are applied more frequently in the method-level, in an attempt to improve code readability and organization. In particular the most frequently performed refactorings are “*Extract Method*”, “*Replace Magic Number With Constant*” and “*Remove Parameter*”, with the “*Extract Method*” presenting over 80% applicability. The evaluation of refactorings is performed mostly through reviews (62%), test cases and is complementary supported with tools. Based on the findings of this study we proposed a generalized refactoring process model for ES that can guide practi-

[5] The data can be downloaded from the following link:
https://www.dropbox.com/sh/fzakcszlhphkhrn/AAAf570i43U-ZKaBSjYSmKVpa?dl=0

tioners during the refactoring process and inspire researchers to work on topics related to ES, such as quality metrics associating run-time and design-time attributes. As a future work we plan to work on a) the application of more specialized code refactorings customized to the needs of embedded software taking into consideration the increased need for building reusable components and b) evaluate the impact of these refactorings on run-time quality attributes.

Acknowledgements

The authors gratefully acknowledge the assistance of ImpediMed Ltd. for giving us insight in their refactoring processes required to make this study possible. The valuable contribution of the volunteer practitioners is gratefully acknowledged. The work of Dr. Arvanitou was financially supported by the action "*Strengthening Human Resources Research Potential via Doctorate Research*" of the Operational Program "*Human Resources Development Program, Education and Lifelong Learning, 2014-2020*", implemented from State Scholarship Foundation (IKY) and co-financed by the European Social Fund and the Greek public (National Strategic Reference Framework (NSRF) 2014–2020).

References

1. Abebe, M., & Yoo, C. J. (2014). Trends, opportunities and challenges of software refactoring: A systematic literature review. international Journal of software engineering and its Applications, 8(6), pp. 299-318.
2. Agarwal, A., Rajput, S., & Pandya, A. S. (2006, May). Power management system for embedded RTOS: An object-oriented approach. In 2006 Canadian Conference on Electrical and Computer Engineering pp. 2305-2309. IEEE.
3. Arcelli, D., Cortellessa, V., Di Pompeo, D., Eramo, R., & Tucci, M. (2019, March). Exploiting architecture/runtime model-driven traceability for performance improvement. In 2019 IEEE International Conference on Software Architecture (ICSA) (pp. 81-90). IEEE.
4. Alkharabsheh, K., Crespo, Y., Manso, E., & Taboada, J. A. (2019). Software Design Smell Detection: a systematic mapping study. Software Quality Journal, 27(3), 1069-1148.
5. Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., & Avgeriou, P. (2015, August). The financial aspect of managing technical debt: A systematic literature review. Inf. Softw. Technol., vol. 64, pp. 52–73.
6. Andrade, H., Crnkovic, I., & Bosch, J. (2020, July). Refactoring software in the automotive domain for execution on heterogeneous platforms. In 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC) (pp. 1534-1541). IEEE.
7. Bandi, A., Williams, B. J., & Allen, E. B. (2013, October). Empirical evidence of code decay: A systematic mapping study. In 2013 20th Working Conference on Reverse Engineering (WCRE) (pp. 341-350). IEEE.
8. Basili, V., Caldiera, G., Rombach, D. (1994). The Goal Question Metric Approach. Encyclopedia of Software Engineering, John Wiley & Sons, pp. 528-532.
9. Brown, W. J., Malveau, R. C., McCormick III, H. W., & Mowbray, T. J. (1998). Refactoring software, architectures, and projects in crisis.

10. Choi, E., Yoshida, N., Ishio, T., Inoue, K., & Sano, T. (2011, May). Extracting code clones for refactoring using combinations of clone metrics. In Proceedings of the 5th International Workshop on Software Clones (pp. 7-13).
11. Al Dallal, J. (2015). Identifying refactoring opportunities in object-oriented code: A systematic literature review. Information and software Technology, 58, 231-249.
12. Elo, S., & Kyngäs, H. (2008). The qualitative content analysis process. Journal of advanced nursing, 62(1), 107-115.
13. Farkas, T., Neumann, C., & Hinnerichs, A. (2009, July). An integrative approach for embedded software design with UML and Simulink. In 2009 33rd Annual IEEE International Computer Software and Applications Conference (Vol. 2, pp. 516-521). IEEE.
14. Feitosa, D., Ampatzoglou, A., Avgeriou, P., & Yumi Nakagawa E. (2015). Investigating Quality Trade-offs in Open-Source Critical Embedded Systems. In 11th International Conference on the Quality of Software Architectures (QoSA). ACM.
15. Fernandes, J. M., Machado, R. J., & Santos, H. D. (2000, May). Modeling industrial embedded systems with UML. In Proceedings of the eighth international workshop on Hardware/software codesign (pp. 18-22). ACM.
16. Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999, July). Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1 edition.
17. Gu, Z., Kodase, S., Wang, S., & Shin, K. G. (2003, May). A model-based approach to system-level dependency and real-time analysis of embedded software. In The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. (pp. 78-85). IEEE.
18. Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y. G., Code smells and refactoring: A tertiary systematic review of challenges and observations, Journal of Systems and Software, Volume 167, 2020.
19. Haendler, T., & Frysak, J. (2018). Deconstructing the Refactoring Process from a Problem-solving and Decision-making Perspective. In International Conference on Software and Data Technologies (pp. 397-406).
20. Hamza, H., Counsell, S., Loizou, G., & Hall, T. (2008, September). Code smell eradication and associated refactoring. In proceedings of the European Computing Conference (ECC).
21. Jenko, M., Medjeral, N., & Butala, P. (2001). Component-based software as a framework for concurrent design of programs and platforms—an industrial kitchen appliance embedded system. Microprocessors and Microsystems, 25(6), 287-296.
22. Keeling, M. (2017). Design It!: From Programmer to Software Architect. Pragmatic Bookshelf.
23. Kim, M., Zimmermann, T., & Nagappan, N. (2012, November). A field study of refactoring challenges and benefits. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (pp. 1-11).
24. Lacerda, G., Petrillo, F., Pimenta, M., & Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. Journal of Systems and Software, 167, 110610.
25. Mens, T., & Tourwé, T. (2004). A survey of software refactoring. IEEE Transactions on software engineering, 30(2), 126-139.

26. Mooij, A. J., Ketema, J., Klusener, S., & Schuts, M. (2020, February). Reducing Code Complexity through Code Refactoring and Model-Based Rejuvenation. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 617-621). IEEE.
27. Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., & Succi, G. (2007, October). A case study on the impact of refactoring on quality and productivity in an agile team. In IFIP Central and East European Conference on Software Engineering Techniques (pp. 252-266). Springer, Berlin, Heidelberg.
28. Moser, R., Sillitti, A., Abrahamsson, P., & Succi, G. (2006, June). Does refactoring improve reusability?. In International conference on software reuse (pp. 287-297). Springer, Berlin, Heidelberg.
29. Murphy-Hill, E., Parnin, C., & Black, A. P. (2011). How we refactor, and how we know it. IEEE Transactions on Software Engineering, 38(1), 5-18.
30. Oliveira, L. B. R., Guessi, M., Feitosa, D., Manteuffel, C., Galster, M., Oquendo, F., & Nakagawa, E. Y. (2013). An investigation on quality models and quality attributes for embedded systems. ICSEA, 13, 1-6.
31. Opdyke, W. F. (1992). Refactoring object-oriented frameworks.
32. Polaczek, J. & Sosnowski J., Exploring the software repositories of embedded systems: An industrial experience, Information and Software Technology, Volume 131, 2021.
33. Rauscher, T. G., & Smith, P. G. (1995). Time-Driven Development of Software in Manufactured Goods. Journal of Product Innovation Management: An International Publication of the Product Development & Management Association, 12(3), pp. 186-19.
34. Ribeiro, T. V., & Travassos, G. H. (2015, May). On the alignment of source code quality perspectives through experimentation: an industrial case. In 2015 IEEE/ACM 3rd International Workshop on Conducting Empirical Studies in Industry (pp. 26-33). IEEE.
35. Van Rompaey, B., Du Bois, B., Demeyer, S., Pleunis, J., Putman, R., Meijfroidt, K., Duenas, J. C. & García, B. (2009, March). Serious: Software evolution, refactoring, improvement of operational and usable systems. In 2009 13th European Conference on Software Maintenance and Reengineering (pp. 277-280). IEEE.
36. Runeson, P., Host, M., Rainer, A., & Regnell, B. (2012). Case study research in software engineering: Guidelines and examples. John Wiley & Sons.
37. Schuts, M., Hooman, J., & Vaandrager, F. (2016, June). Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In International Conference on Integrated Formal Methods (pp. 311-325). Springer, Cham.
38. Schrom, H., Schwartze, J., & Diekmann, S. (2017, September). Building automation by an intelligent embedded infrastructure: Combining medical, smart energy, smart environment and heating. In 2017 International Smart Cities Conference (ISC2) (pp. 1-4). IEEE.
39. Sharma, T., Suryanarayana, G., & Samarthyam, G. (2015). Challenges to and solutions for refactoring adoption: An industrial perspective. IEEE Software, 32(6), 44-51.
40. Simons, C., Singer, J., & White, D. R. (2015, September). Search-based refactoring: Metrics are not enough. In International Symposium on Search Based Software Engineering (pp. 47-61). Springer, Cham.

41. Szőke, G., Antal, G., Nagy, C., Ferenc, R., & Gyimóthy, T. (2017). Empirical study on refactoring large-scale industrial systems and its effects on maintainability. Journal of Systems and Software, 129, 107-126.
42. Trudeau, J. (2013), Software Reuse By Design in Embedded Systems, in Software Engineering for Embedded Systems, pp. 261-280,
43. Vallius, T., Haverinen, J., & Röning, J. (2007). Object-oriented embedded system development method for easy and fast prototyping. In Mechatronics for Safety, Security and Dependability in a New Era (pp. 265-270). Elsevier.
44. Van Vliet, H., Van Vliet, H., & Van Vliet, J. C. (2008). Software engineering: principles and practice (Vol. 13). Hoboken, NJ: John Wiley & Sons.
45. Wieringa, R. J. (2014). Design science methodology for information systems and software engineering. Springer.
46. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in software engineering. Springer Science & Business Media.