Investigating Quality Trade-offs in Open Source Critical Embedded Systems

Daniel Feitosa[1], Apostolos Ampatzoglou[1], Paris Avgeriou[1], Elisa Yumi Nakagawa[2]
[1] Department of Mathematics and Computer Science, University of Groningen, Groningen, The Netherlands
[2] Department of Computer Systems, University of São Paulo, São Carlos, SP, Brazil
d.feitosa@rug.nl, a.ampatzoglou@rug.nl, paris@cs.rug.nl, elisa@icmc.usp.br

ABSTRACT
During the development of Critical Embedded Systems (CES), quality attributes that are critical for them (e.g., correctness, security, etc.) must be guaranteed. However, this often leads to complex quality trade-offs, since non-critical qualities (e.g., reusability, understandability, etc.) may be compromised. In this study, we aim at empirically investigating the existence of quality trade-offs, on the implemented architecture, among versions of open source CESs, and compare them with those of systems from other application domains. The results of the study suggest that in CES, non-critical quality attributes are usually compromised in favor of critical quality attributes. On the contrary, we have not observed compromises of critical qualities in favor of non-critical ones in either CES or other application domains. Furthermore, quality trade-offs are more frequent among critical quality attributes, compared to trade-offs among non-critical quality attributes. Our study has implications for both practitioners when making trade-offs in practice, as well as researchers that investigate quality trade-offs.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics; D.2.11 [**Software Engineering**]: Software Architectures – *domain-specific architectures.*

General Terms
Measurement, Performance, Design, Experimentation, Security

Keywords
embedded systems; quality trade-offs; software metrics

1. INTRODUCTION
Critical Embedded Systems (CESs) are among the most significant types of software-intensive systems, since they are extremely pervasive in modern society, being used from cars to power plants [19]. CESs are embedded systems in which design errors can potentially be catastrophic [12], in terms of causing serious damage to the environment or to human lives, or non-recoverable material and financial losses [2]. Due to the criticality of such systems, the satisfaction of multiple quality constraints must be guaranteed, which is far from trivial, as it entails complex trade-offs: compared to other application domains, in CES such trade-offs to a large extent concern safeguarding the levels of critical against other non-critical qualities [5], [18]. As critical quality attributes (QAs), we characterize those that can cause catastrophic failures, as mentioned before, and usually concern performance, security and reliability.

Trade-offs occur because almost every design decision has the potential to positively affect some QAs and negatively affect others. For example, solutions that aim at enhancing security might, as a side effect, harm the performance of the system. Resolving a QA trade-off is a complex process, as it touches upon multiple design decisions. If a trade-off is not resolved well, it can lead to poor satisfaction of QAs, or an overkill in their satisfaction [8]. Understanding the nature of such trade-offs is of paramount importance to guide practitioners in making optimal trade-offs, and researchers in facilitating the practitioners in their job.

Until now, trade-offs between quality attributes have not received sufficient empirical investigation [8] in real-life systems, but have mostly been addressed at a theoretical level. Specifically, we lack empirical evidence on the types of trade-offs performed in the domain of CES, and how exactly these trade-offs differ from other application domains. The goal of this study is to provide such evidence, by *examining trade-offs in the implementation of real-life systems for both CES and other domains*. Although QA trade-off analysis is usually investigated at the architecture design level, we work at the architecture implementation level (i.e., source code) for two reasons. First, the implemented architecture (derived from the source code) may deviate from the intended (as designed) architecture, in a phenomenon known as architectural drift [23]. But we want to study the quality trade-offs as they exist in real systems, not as they may have been intended during design. Therefore, as a side-effect of this decision, we emphasize that in this study both intentional and unintentional trade-offs are being considered without distinction between them. Second, the availability of source code is much greater than the availability of architecture design documentation (especially with information about quality trade-offs) in both open-source systems (OSS) and commercial systems.

Thus, in this study, we aim at exploring:

- (**goal - a**) the existence of quality trade-offs in the implemented architecture of CES, by investigating their source code; and
- (**goal - b**) whether trade-offs differ between CES and systems of other application domains.

In order to explore the existence of quality trade-offs from source code, we need to use methods, such as static analysis, to explore the evolution of quality attributes (i.e., changes in the levels of quality across successive versions), since no documentation

QoSA'15, May 4-8, 2015, Montréal, QC, Canada.

http://dx.doi.org/10.1145/2737182.2737190

regarding quality trade-offs is available at the source code level. In this sense, trade-offs refer to cases where changes in source code correlate with the improvement of one quality attribute and deterioration of a second. As aforementioned, this means that we extract both intentional and unintentional trade-offs, thus being inclusive rather than exclusive.

To accomplish the aforementioned goals, we performed an embedded multiple-case study on multiple versions of twenty one OSS projects [25]. The results of the study suggest the existence of QAs trade-offs in the CES domain, as well as in other domains, and highlight differences between them. The remainder of this paper is organized as follows: related work is presented in Section 2, along with a discussion of the main contributions of this study. In Section 3, we present the design of the case study. In Sections 4 and 5 we present the results and discuss the most important findings respectively. In Section 6, we report on the identified threats to validity and actions taken to mitigate them. Finally, in Section 7 we conclude the paper and present some interesting extensions for this study.

2. RELATED WORK

In this section we present related work that discusses software quality attributes. In the software engineering literature, QAs can be characterized based on many classifications (e.g., [1] and [17]); however, since our work is focused on the domain of CES, we decided to simply classify them as either critical or non-critical.

We organize this section by first presenting studies on the domain of embedded systems (Section 2.1) and next on the evolution of software qualities in general (Section 2.2)[1], as trade-offs are inherent in evolution. When presenting related work, we emphasize (in bold italic) the number of cases considered in the studies, as well as the tackled QAs. This information will be further summarized and compared with the main points of advancement of our work (Section 2.3). Most of the related work discussed in this section is based on a mapping study on software quality trade-offs by Barney et al. [8].

2.1 Quality Trade-offs in Embedded Systems

Concerning the interplay between QAs in the domain of embedded systems, Del Rosso presents an architectural approach for improving the ***performance*** of software products derived from a product family for real-time embedded systems, and its possible implications to ***maintainability*** [24]. To validate his approach, he conducts two cases studies on assessing the performance: (a) of ***one specific product line***; and (b) on ***four scenarios involving derived products*** during product line evolution (the addition of new features). The first study involved one case, while the second involved four cases. The performance is measured by *run-time metrics* related to *memory allocation*, and the author discusses the trade-offs with maintainability. The results suggest that, by analyzing the commonalities and differences among derived products, one can extract bottlenecks and problems in core architecture (e.g., *God class*).

In a similar context, Oliveira et al. investigate the relationship between non-critical quality attributes, measured by metrics obtained from source code, and ***performance***, measured by physical metrics (i.e., *memory*, *time*, and *energy*) obtained from run-time monitoring [21]. The explored quality attributes are the following: ***complexity***, ***coupling***, ***cohesion***, ***extendibility***/ ***reuse***, and ***population*** / ***size***. The study comprises a case study involving the evaluation of ***four alternative designs of an example system***, in which measurements are collected for each design solution, showing potential trade-offs between the aforementioned metrics, and supporting the decision-making regarding the selection of a design solution. Results indicate the existence of trade-offs between quality and physical metrics, as well as the fact that quality metrics can provide information regarding high-level QAs, guiding the design solution selection at early stages, which might lead to significant gain in physical characteristics latter on. A main difference of this work, compared to ours, is that we are investigating the relationship between QAs within the evolution of the software, rather than trade-offs among possible designs from the solution space.

2.2 Quality Analysis through Evolution

Concerning the investigation of quality attributes through source code evolution; Buyens et al. [13] present an analysis of the interaction, ***on three cases***, between ***security***, measured by two metrics, and ***maintainability***, measured by two metrics as well. Each security metric is based on one security principle, namely *Least Privilege* (the metric is the number of violations) and *Attack Surface* (the metric is the estimation of the attackers' effort). Whereas, the two maintainability metrics are: *Coupling Between Compone*nts, and *Components Instability*. The metrics are measured while applying modification in the implementation of each security principle (i.e., changing the involved components), causing changes in the system. Results of this study indicate that: (a) transformations are more effective when applied jointly, and (b) trade-offs exist between security metrics, and between security and maintainability QAs.

Additionally, Barros et al. characterize the evolution of ***one open source project*** (Apache Ant) in terms of **size**, **changeability**, **cohesion**, and **coupling** QAs, through an exploratory study [9]. A main point of discussion is regarding the investigation of the high cohesion and low coupling principle. The results suggest that the original design was "lost" throughout the evolution of the system, while architectural optimization is hard, leading to a more complex to maintain resulting design.

Finally, Di Penta et al. [22] study ***security*** in software systems by investigating the life-span of vulnerabilities among software versions. The authors define *vulnerability* as "any instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy" [22]. Di Penta et al. investigate a total of 14 vulnerabilities, organized into four categories: *input validation*, *memory safety*, *race / control flow condition*, and *other*. To investigate the evolution of these vulnerabilities, they perform a case study on ***three open source software projects***. Results indicate that: (a) vulnerabilities tend to be removed from the source code (between 56% and 93 %), (b) functions with ***security*** issues tend to be replaced, and (c) new ***functionality*** tends to introduce new vulnerabilities.

2.3 Overview of Related Work

The main differences of this study compared to the related work are summarized in Table 1. Specifically, we present the

[1] We note that the presentation of related work on software evolution is indicative, since the amount of research on this domain is too large to include in this paper, due to size limitations.

Table 1. Overview of related work

#ref	App. Domain		QA		#cases	
	CES	Others	#critical	#non-critical	CES	Others
[25]	√	X	1	1	4	N/A
[22]	√	X	1	5	4	N/A
[14]	X	√	1	1	N/A	3
[10]	X	√	0	4	N/A	1
[23]	X	√	1	1	N/A	3
this	√	√	3	6	4	17

differences in: (a) the studied application domains, (b) the studied QAs, and (c) the size of the performed case studies.

Therefore, the main contributions of this study with respect to the research state-of-the-art are:

c1: it compares trade-offs that appear in CESs with other application domains. To the best of our knowledge, this is the first study that presents empirical evidence on this matter; and

c2: it investigates the interplay among 9 QAs. To the best of our knowledge, this is the most inclusive study of this type in terms of investigated QAs.

3. CASE STUDY DESIGN

This section describes the case study protocol, which was designed according to the guidelines of Runeson et al. [25], and is reported based on the Linear Analytic Structure [25].

3.1 Objectives and Research Questions

The goal of this study is described using the Goal-Question-Metrics (GQM) approach [10], as follows: "***analyze*** *open source software* ***for the purpose of*** *understanding quality attributes trade-offs* ***with respect to*** *the application domain of CES and others* ***from the point of view of*** *software developers* ***in the context of*** *Open Source projects*". Based on the goal of this study, we defined the following research questions:

RQ_1: Are there trade-offs between quality attributes of CES?

- **$RQ_{1.1}$**: Are there trade-offs between non-critical quality attributes?
- **$RQ_{1.2}$**: Are there trade-offs between critical quality attributes?
- **$RQ_{1.3}$**: Are there trade-offs between critical and non-critical quality attributes?
- **$RQ_{1.4}$**: Are trade-offs between pairs of quality attributes bi-directional?

RQ_1 aims at investigating *goal-a,* i.e., investigating the existence of quality trade-offs. In order to further investigate the nature of such trade-offs (**RQ_1**), we employ the QA classification into critical and non-critical attributes and explore interactions between them (**$RQ_{1.1}$** - **$RQ_{1.3}$**). Intuitively, we would expect that quality trade-offs in CES would be different among critical and non-critical qualities categories. Subsequently, it is relevant to explore whether trade-offs between QAs occur on both directions, i.e., if the improvement of a certain QA "causes" another to decrease, does the vice-versa phenomenon occur? (**$RQ_{1.4}$**).

RQ_2: What is the difference in quality attributes trade-offs among CES and non-CES domains?

- **$RQ_{2.1}$**: Are there similar trade-offs among CES and non-CES domains?
- **$RQ_{2.2}$**: Are there different trade-offs among CES and non-CES domains?

Since one of the most prevalent characteristics of the CES domain is the distinction between critical and non-critical qualities, it is interesting to compare it to other domains (regarding the same QAs), in order to see how this distinction is reflected in quality trade-offs (**RQ_2**). Therefore, it is important to identify similar trade-offs among CES and other domains (**$RQ_{2.1}$**), as well as the differences in trade-offs between QAs (**$RQ_{2.2}$**).

3.2 Case Selection and Unit of Analysis

This study is an embedded multiple-case study, in which each case is represented by one project. As unit of analysis we refer to changes in the quality attributes between subsequent versions of any project. In order to select appropriate cases for our study we needed to retrieve successive versions of OSS projects of two different groups: CES projects, and non-CES projects. The projects used in our analysis were required: (a) to be written in Java, due to limitations of the used tools (see Section 3.3.3), (b) to have an adequate number of versions for evolution analysis, and (c) not to be considered as "toy examples". The selected projects, accompanied by some additional information, are presented in Table 2. We clarify that although the selected CESs do not provide high-level end-user functionalities (e.g., move a robot), they are high-quality systems (more specifically, virtual machines) tailored for CES. Therefore, they are subject to the same, or stricter, (critical) constraints when compared to applications running on top of the virtual machine.

In order to ensure that the sample of non-CES represents a number of different application domains, thus avoiding bias from

Table 2. Projects considered in the case study

Project Name	Starting Year	Size**	NoV*	Group	NoV*
Java-SE-Embedded	2010	834k	14	CES	50
LeJOS	2000	81k	16		
LeJOS-EV3	2013	30k	9		
LeJOS-NXJ	2006	52k	11		
Art of Illusion	2000	53k	32	Non-CES	572
DrJava	2002	180k	24		
FileBot	2007	532k	25		
FreeCol	2002	51k	34		
FreeMind	2000	28k	34		
Hibernate	2001	123k	28		
HomePlayer	2005	24k	32		
HtmlUnit	2002	27k	26		
iText	2000	56k	23		
JFreeChart	2000	62k	56		
Lightweight-Java-Game-Library	2002	72k	40		
MediathekView	2008	17k	41		
Mondrian	2001	51k	33		
OpenRocket	2009	182k	27		
Pixelitor	2009	27k	33		
Subsonic	2004	282k	42		
TuxGuitar	2005	28k	19		

*NoV = Number of Versions

**Size, of the last version, in lines of code

specific non-CES domains, we have selected systems[2] from 10 different domains. Therefore, our dataset contains four CES and an average of 3.1 systems from each of the 10 different non-CES domains. This means that the number of CES is not comparable to the total number of non-CES, but it is comparable to the number of systems in each of the different application domains.

3.3 Data Collection and Pre-processing

In order to answer the research questions, we extracted three sets of variables from each unit of analysis (see Section 3.3.3). The first set comprises data related to project identification and classification (V1 and V2). The second and third sets comprise variables for the quantification of critical (V3 – V5) and non-critical (V6 – V11) QAs. We clarify that in this paper QAs are assessed based on a set of metrics. To this end, we selected several metrics that, to the best of our knowledge, are able to quantify the levels of quality. The two sets of metrics, for critical and non-critical QAs respectively, are presented in detail in the following sections.

3.3.1 Assessment of Critical Quality Attributes

Bugs have been extensively investigated as indicators of quality. More specifically, Misra and Bhavsar [20] have explored bugs as indicators for correctness, and Zaman et al. [26] have explored bugs as indicators for security and performance. When using bugs to quantify quality, it is a common practice to classify them into categories. For example, Zaman et al. [26] classified bugs according to their effect on specific QAs (e.g., security and performance). Therefore, to evaluate software projects with respect to their critical quality attributes, we performed static analysis by collecting the amount of several different types of bugs. For that, we used the tool *FindBugs*[3]. *FindBugs* is capable of detecting vulnerabilities in software by using bug patterns [16]. In this case study, we have chosen to use *FindBugs* because it provides:

- adequate performance (with respect to precision) when compared to similar tools [16] [27];
- a collection of over 400 bug patterns; and
- a grouping of these bug patterns in nine high-level categories (i.e., *Security, Correctness, Multithreaded Correctness, Performance, Malicious Code, Bad Practice, Internationalization*, *Experimental* and *Dodgy Code*), which can in turn be mapped into quality attributes.

In this study, in order to evaluate critical quality attributes, we considered the first five categories (in total 246 bug patterns), as they can be mapped to three critical QAs: *correctness* (*Correctness and Multithreaded Correctness* categories), *performance* (*Performance* category), and *security* (*Security* and *Malicious Code* categories). Therefore, the level of quality for the three aforementioned QAs is measured by the quantity of detected bugs. We clarify that for the correctness and security QAs, the number of bugs is the sum of the two categories each QA is comprised of. For example, security is measured by summing the number of bugs from both *Security* and *Malicious Code* categories. For all QAs a lower number of bugs reflects a higher level of quality.

3.3.2 Assessment of Non-critical QAs

Regarding the quantification of non-critical quality attributes, we selected to use the Quality Model for Object-Oriented Design (QMOOD) [7]. QMOOD is a well-known hierarchical quality model that provides an approach for assessing six high-level quality attributes: reusability, understandability, functionality, extendibility, effectiveness, and flexibility [7]. These attributes are quantified based on 11 structural object-oriented design properties: design size, hierarchies, abstractions, encapsulation, coupling, cohesion, composition, inheritance, polymorphism, messaging, and complexity [7]. The definition for the aforementioned quality attributes and properties, and the equations to calculate the score of each quality attribute (by using weighted sum) can be found in the work of Bansiya and Davis [7]. Although the QMOOD quality model seems rather simplistic in its calculations, weighted sum is the most classical and, therefore, used approach for combining metrics [14]. Additionally, Bansiya and Davis [7] validated it empirically by using 13 appraisers, with 2-7 years of experience in commercial software development, to evaluate 14 software projects. Their evaluation was compared to the quality model output, which showed to be significantly correlated. Therefore, we selected to use QMOOD since it:

- uses simple calculations, which can be easily automated;
- provides clear definitions of low-level properties and direct mapping to quality attributes; and
- presents a fair amount of quality attributes.

In order to assess the QAs for each project we used *Percerons Client*[4], i.e., a tool developed in our research group, which automates the assessment of these QAs for provided Java classes. *Percerons* is a software engineering platform [6], created by one of the authors, to facilitate empirical research in software engineering, by providing: (a) indications of componentizable parts of source code, (b) quality assessment, and (c) design pattern instances. The platform has been used for similar reasons in [5], [15], and [3].

3.3.3 Collection Procedure and Pre-processing

The data collection phase was a two-step process. First, the QAs assessment variables were extracted from every unit of analysis, using *FindBugs* and *Percerons Client*. Both tools work on Java binary code, so we provided them with a set of *.jar* files (one per version), and recorded the outcome in ***an initial dataset***. For *FindBugs*, we used the command line version 3.0.0, for easy reproduction and automation purposes. During execution, we requested maximum effort (i.e., enabling analysis that increases precision), and reported bugs from all urgency priorities (i.e., from least to most harmful to the system).

The initial dataset was compiled in a single file for each project, containing all extracted data (from both tools) for each version. This file comprises a table with the following fields for each row of data: *version, number of correctness bugs, number of performance bugs, number of security bugs, reusability score, understandability score, functionality score, extendibility score,*

[2] We collected these systems from https://sourceforge.net, and considered the root from each category as the domain. Additionally, each system may belong to more than one domain.

[3] http://findbugs.sourceforge.net/

[4] http://www.percerons.com

effectiveness score, and *flexibility score*. The number of bugs from each aforementioned QA was obtained by counting the rule violations with medium and high confidence from *Findbugs* output. We decided to filter out the bugs with low confidence level for increasing the precision of the automatic rule violation identification process. Specifically, we manually analyzed/validated a sample of 15 bugs per level of confidence (chosen randomly), and we estimated that the precision for low, medium, and high categories were 26.67%, 60%, and 73.33% respectively.

Next, the ***final dataset*** was created by calculating the difference between two consecutive versions ($\delta_{variable} = variable_v - variable_{v-1}$), for every version *v* of each project. This was performed for each estimator (number of bugs or design-time attribute quality score). Then all data were merged in a single table consisting of the following fields: *project name, type of project (i.e., CES or non-CES),* $\delta_{correctness}$, $\delta_{performance}$, $\delta_{security}$, $\delta_{reusability}$, $\delta_{understandability}$, $\delta_{functionality}$, $\delta_{extendibility}$, $\delta_{effectiveness}$, *and* $\delta_{flexibility}$. Finally, the values of the δ* variables were classified as *improvement cases, deterioration cases, or neutral cases*, based on the sign of the corresponding δ value.

Summarizing, the full list of variables collected from each unit of analysis, together with their description, is presented in Table 3.

Table 3. List of collected variables

Variable	Description	Tool
[V1]	Software project: the name of the OSS project from which data were extracted.	-
[V2]	Domain Group: project belongs either to CES or non-CES	-
[V3]	Difference between two versions, in the count of security rule violations: count of bug pattern instances in "Malicious code vulnerability" and "Security" categories.	FindBugs
[V4]	Difference between two versions, in the count of "Performance" rule violations: count of bug pattern instances in "Performance" category.	
[V5]	Difference between two versions, in the count of correctness rule violations: count of bug patterns in "Correctness" and "Multithread correctness" categories.	
[V6]	Difference between two versions, in the reusability score: the reusability assessment computed as defined by Bansiya and Davis [8].	Percerons Client
[V7]	Difference between two versions, in the flexibility score: the flexibility assessment computed as defined by Bansiya and Davis [8].	
[V8]	Difference between two versions, in the Understandability score: the understandability assessment computed as defined by Bansiya and Davis [8].	
[V9]	Difference between two versions, in the Functionality score: the functionality assessment computed as defined by Bansiya and Davis [8].	
[V10]	Difference between two versions, in the Extendibility score: the extendibility assessment computed as defined by Bansiya and Davis [8].	
[V11]	Difference between two versions, in the Effectiveness score: the effectiveness assessment computed as defined by Bansiya and Davis [8].	

3.4 Data Analysis

During this phase, we analyzed the previously described δ* fields (V3 – V11), in order to identify trade-offs, which will be further used for comparison between CES and non-CES groups. We clarify that these fields represent assessments of the studied QAs, and, therefore, when referring to the attributes, we are in practice referring to their assessments. The analysis of the collected data is split in three steps:

(step 1) *Analysis of pairs of QAs*: For both groups (CES and non-CES projects), we have to seek evidence on the existence of trade-offs, in every pair of QAs. For instance, Figure 1 depicts the analysis of qualities V6 vs. V10, for CES. Therefore, for every pair of QAs, we proceed as follows:

(step 1.1) *Filter improvement cases*: As we are looking for cases of occurrences of trade-offs, it is important to select only cases in which one of the two QAs has improved. For this reason, we create two sub-datasets, each one consisting of the cases having positive scores for the respective QA. For instance, in Figure 1, the two sub-datasets consist of the cases in which V6 improves (positive values of V6), and the cases in which V10 improves (positive values of V10). This ensures that we are tracking the cases in which perfective maintenance tasks may have been performed in order to improve the tracked aspect of the software.

(step 1.2) *Calculate statistics of sub-dataset*: Each sub-dataset (corresponding to an improving QA) is analyzed by creating a frequency table for the second QA. In the example presented in Figure 1, for the sub-dataset comprising cases of improvement of V6, we calculate the frequencies for V10; and vice-versa. Thus, when exploring each sub-dataset, we calculate the frequency percentages of the classes of the second QA (i.e., improvement cases, deterioration cases, stable cases). Next, the improvement cases are marked as *co-evolution*, the deterioration cases are marked as *trade-offs*, whereas the neutral cases as *neutral*. For instance, in Figure 1, in the sub-dataset comprising improvement cases of V6, we calculated "V10 -" (trade-off), "V10 +" (co-evolution), and "V10 0" (neutral).

(step 1.3*)* *Filter evidence*: To identify the trade-offs occurring between QAs, we keep out of the two sub-datasets of step 1.2, only those in which the percentage of the *trade-off* cases is higher than the percentage of *co-evolution* and *neutral.* In the example of Figure 1, we identify a possible trade-off at the sub-dataset in which the improvement of V6 affects negatively V10.

(step 2) *Synthesis of presentation*: In this step we synthesize and graphically represent the results of step 1, so as to answer the research questions.

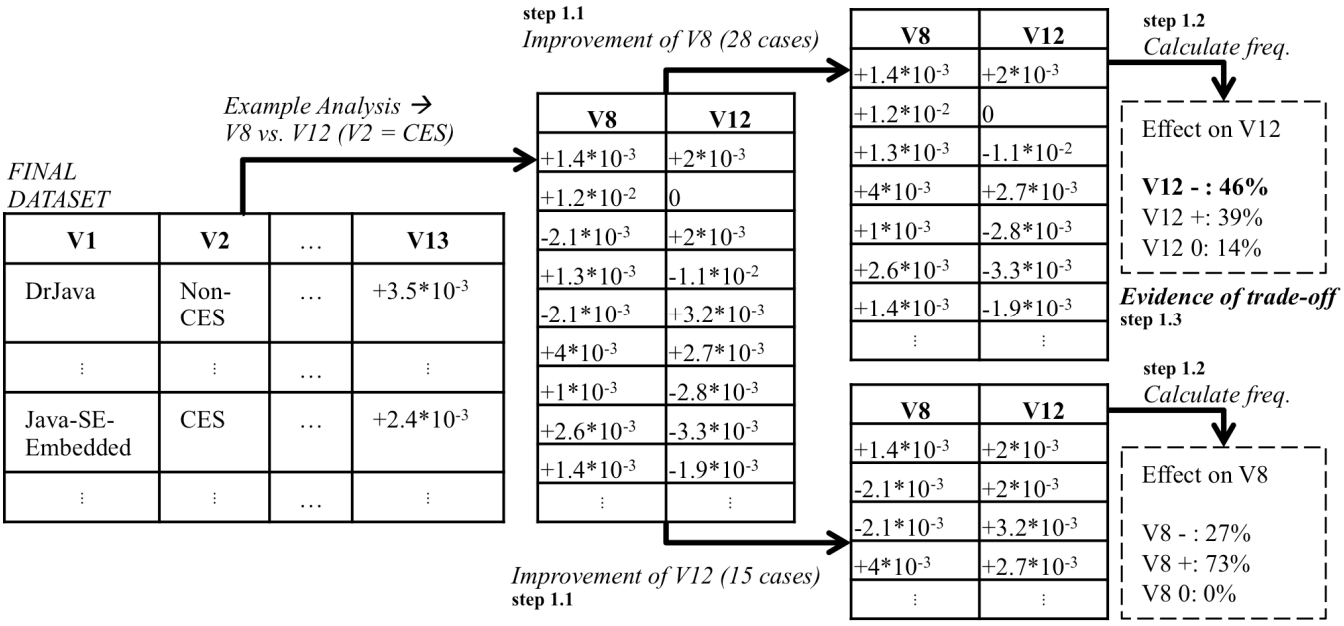

Figure 1. Example of trade-off analysis within the final dataset

Two heat maps are derived from the information on step 1: one depicting the trade-offs within the CES group (row: improved QAs, columns: affected QAs, intensity: the frequency of *trade-offs*); and another showing the comparison of trade-offs between the two groups (the difference and similarities between CES and non-CES trade-offs).

(step 3) *Comparison of evidence in CES projects*: With the data collected and summarized, it is analyzed within the CES group, aiming at comparing the interactions between the QAs in order to answer $\mathbf{RQ_1}$ and its sub-questions. For this step the heat map on CES trade-offs is used.

(step 4) *Comparison of evidence from groups*: The analysis is now extended to the non-CES group, aiming, therefore, at comparing all collected data in order to answer $\mathbf{RQ_2}$ and its sub-questions. For this step the heat map on the comparison between CES and non-CES groups is used.

Summarizing the procedure for answering the RQs, Table 4 presents the mapping between each RQ, the used variables, as well as the step of the analysis in which RQs are answered and the presentation methods that are used.

Table 4. Mapping of RQs to variables, steps, and presentation

Research Question	Variables Used	Step	Presentation Method
$\mathbf{RQ_1}$	[V2-V13]	3	Heat Map on CES trade-offs
$\mathbf{RQ_{1.2}}$	[V2] [V8-V13]		
$\mathbf{RQ_{1.2}}$	[V2-V7]		
$\mathbf{RQ_{1.3}}$	[V2-V13]		
$\mathbf{RQ_{1.4}}$	[V2-V13]		
$\mathbf{RQ_2}$	[V2-V13]	4	Heat Map on comparison between CES and non-CES
$\mathbf{RQ_{2.1}}$	[V2-V13]		
$\mathbf{RQ_{2.2}}$	[V2-V13]		

4. RESULTS

In this section we present the output of the analysis, and answer the research questions. To answer $\mathbf{RQ_1}$ and its sub-questions, we explore the findings obtained from step 3. In order to visualize the interaction between the QAs, we compiled raw data (omitted from this manuscript due to space limitations[5]) into a heat map (see Figure 2). In the heat map of Figure 2, each cell represents the effect of improving one QA (vertical axis) over another (horizontal axis). The intensity of the heat map (i.e., color darkness – also written inside the cell) represents the percentage of the cases that constitute valid trade-offs (see step 1.3). Moreover, the two bold lines in the map divide it into quadrants in order to highlight the interactions within and between the critical and non-critical groups of QAs. Hence, the top-left quadrant represents the interactions between critical QAs, the bottom-right quadrant represents the interactions between non-critical QAs,

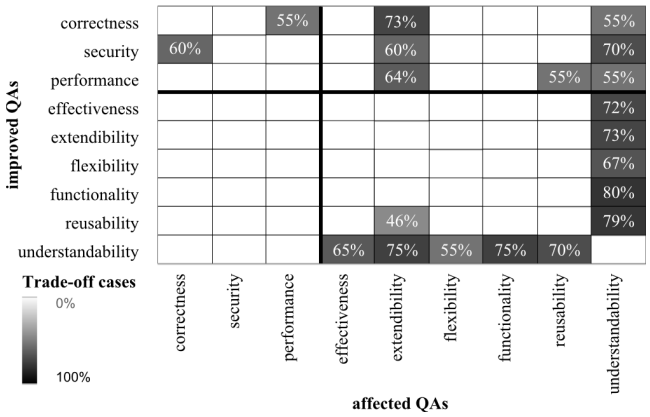

Figure 2. Trade-offs in CES domain

[5] Supplementary material on the collected data during the study is available at: http://www.rug.nl/research/software-engineering/publication_files/QA-tradeoffs-TR-2015-01-08.pdf

while the other two represent the interaction between QAs of the two groups.

Based on Figure 2, we are able to answer all sub-questions of **RQ$_1$**, by confirming the existence of trade-offs between QAs (see Section 3.4, step 1), and answering affirmatively **RQ$_{1.1}$ - RQ$_{1.3}$**. Consequently, by investigating each quadrant separately, it's also possible to point out possible trade-offs between critical QAs (second quadrant), non-critical QAs (fourth quadrant), and between QAs of the two groups (first and third quadrants). The findings from exploring **RQ$_{1.1}$ - RQ $_{1.3}$** is the existence of trade-offs[6] between:

- *understandability* and the other non-critical QAs (and vice-versa);
- *correctness* and *performance*, as well as between *security* and *correctness*;
- all critical QAs and *extendibility,* and between all QAs and *understandability*;
- *performance* and *reusability*;
- *reusability* and *extendibility*.

Subsequently we examine whether the interactions between two QAs are bi-directional (**RQ$_{1.4}$**), i.e., if the improvement of one QA negatively affects another QA, the opposite relationship also holds. To answer this research question, we examine Figure 2 for symmetries. We observe that although we identified some bi-directional interactions, it is not possible to conclude that all identified trade-offs between QAs are bi-directional. However, for some pairs of QAs, bi-directional trade-offs can be identified, i.e., between *understandability* and the rest of the non-critical QAs (*effectiveness*, *extendibility*, *flexibility*, *functionality*, and *reusability*). Moreover, we highlight one interesting finding regarding the interactions between critical and non-critical QAs: although the improvement of some critical QAs negatively affects non-critical QAs, the opposite phenomenon never appears, i.e., in this study we found no evidence of non-critical QAs negatively affecting critical QAs.

Finally, having examined the trade-offs in the CES domain, we can compare them with other application domains (**RQ$_2$**): in what

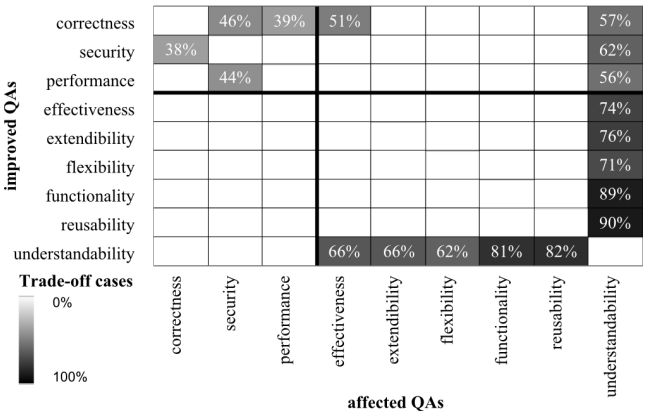

Figure 3. Trade-offs in non-CES domain

[6] We note that, in this study, when reporting trade-offs in the form of "*trade-off between QA_A and QA_B*", we refer to a compromise in the levels of QA_B in favor of an improvement in the levels of QA_A.

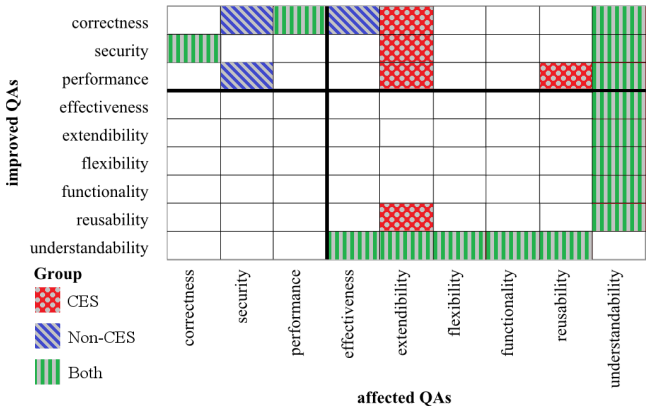

Figure 4. Comparison between CES and non-CEs groups

aspects they are similar (**RQ$_{2.1}$**), and the ones in which they differ (**RQ$_{2.2}$**). To answer these questions, we created two heat maps: one akin to that depicted on Figure 2 (see Figure 3), but considering data from the non-CES projects, and another representing the difference of the two heat maps (see Figure 4). In Figure 4, three different filling patterns (with their respective colors) represent the possible classifications for the observed trade-offs: evident only in the group of CES projects (red background with circles); evident only in the group of non-CES projects (blue with slanted lines); and evident in both groups (green background with vertical lines). Based on this figure, we answer affirmatively **RQ$_{2.1}$ - RQ $_{2.2}$,** and, additionally, make the following observations:

- *Similarities between the two groups of projects:* Trade-offs between security and correctness; between correctness and performance; between all QAs and understandability; and between understandability and non-critical QAs.
- *Trade-offs occurred only in the group of CES projects*: between the critical QAs and extendibility; between reusability and extendibility; and between performance and reusability.
- *Trade-offs occurred only in the group of non-CES projects*: between correctness and security, as well as between performance and security; and between correctness and effectiveness.

Finally, concerning the group of CES projects, the trade-offs occur mostly between critical QAs and non-critical QAs, which implies that, in the CES domain, non-critical QAs are more often sacrificed in favor of critical QAs.

5. DISCUSSION

In this section we present a discussion of the results, by providing possible interpretations and a comparison against related work (when applicable). We first discuss the findings from the CES trade-off analysis, and then the comparison between CES and non-CES. At the end of this section, we discuss possible implications to researchers and practitioners.

5.1 Trade-offs in CES Domain

By exploring the trade-offs in CES, the following observations can be made:

- *extendibility is negatively affected by reusability.* This is intuitive for CES. In general, embedded systems provide

specific functionalities that are not designed to facilitate future extensions in an object-oriented way (e.g., adding subclasses, polymorphic methods, etc.). Therefore, the addition of new functionality is expected to be performed by adding methods in existing classes, making existing methods larger in size, or adding new concrete classes, which in turn lead to even more decreased extendibility. On the other hand, according to [7], such classes (which offer large amount of functionalities) are considered more probable to be reused, since they provide more reuse opportunities, regarding offered functionalities.

- *performance negatively affects reusability*. One possible explanation is that, in order to improve the system performance, some solutions (e.g. refactoring of class into inner class[7]) lead to deterioration of aspects that support reusability, such as cohesion, coupling, and size. Coupling and cohesion are important assessors of reusability in the sense that they are related to the adaptation time needed for reusing a specific piece of code. A similar finding can also be drawn based on the work of Oliveira et al. [21], who suggest that *cohesion* and *coupling* metrics, that are assessments of reusability, are compromised in favor of metrics for *performance* [7].

Although the results of Figure 2 suggest that *extendibility is negatively affected by all critical QAs*, we believe that this result needs further investigation. Intuitively, extendibility is compromised by source code growth [4], and embedded system development style, as already mentioned. Therefore extendibility deteriorates during the evolution of CES, but we do not have evidence regarding the extent that this is connected to bug solving (i.e. improvement of critical QAs). However, a similar finding is reported in [21], where metrics for *extendibility are compromised in favor of performance.*

The rest of the findings are discussed in the next subsection, as they are also observed in the non-CES group.

5.2 Comparison of the Two Groups

On the one hand, in both CES and non-CES, we were able to observe the following:

- *non-critical QAs do not affect critical QAs*. Improvements on object-oriented properties (i.e., enhancement in design-time QAs – all non-critical QAs investigated in this study are design-time) are not likely to result in additional source code vulnerabilities (rule violations – assessment of critical QAs). A possible explanation is that there is no tension between non-critical and critical QAs. However, another possibility is that when improving design-time quality attributes, the developers are refactoring the code without changing its external behavior (extract class, extract method, etc.), which only "moves" rule violations to other parts of the system, without introducing new ones. In addition to that, especially concerning CES, this finding is intuitive in the sense that it was not expected from development teams to compromise a critical QA in a critical system, in favor of a non-critical one.

- *security negatively affects correctness.* Fixing security vulnerabilities can lead to additional errors in the code. For example, in order to fix a vulnerability related to a field that is not well implemented and should be moved[8], one might do the refactoring as suggested, but forget to initialize the field[9].

- *correctness negatively affects performance*. Coding mistakes are common during development (e.g., accessing an already freed reference[10]). Therefore, in order to solve these bugs, one might use inefficient coding styles (in terms of performance) in order to ensure that the output is the expected (e.g., introducing extra parameters in a method that ends up being of limited utility[11]).

On the other hand, by comparing the differences between the two groups, we identified the following findings:

- Although specific evidence of trade-off was discussed already (in the previous subsection), we note that the higher frequency of trade-offs between the critical QAs with non-critical QAs might reflect a higher importance of critical QAs over non-critical QAs in CES.

- *In non-CES, correctness* negatively affects *effectiveness*. While maintaining parts of source code, it might be the case that more non-object-oriented approaches are employed, leading to a reduction in system's effectiveness. We note that effectiveness is quantified by assessing how well the object-orientation paradigm is employed in the source code [7]. For example, in order to solve missed locks[12], one might centralize the responsibilities to avoid forgetting the lock.

- *In non-CES, correctness affects all other critical QAs*. This might be an indication that functionalities are not optimally implemented (i.e., implying less attention to errors or less knowledge on the topic) in other domains, possibly due to a lack on developers' skills.

- *In non-CES, security is affected by all other critical QAs*. Similarly to the previous finding, code exploitation vulnerabilities appear to be common in other domains, and could also be explained by lack of skills regarding this issue.

Finally, although the results of Figure 4 suggest that *understandability is negatively affected by all other QAs* and vice-versa, we believe that this result needs further investigation. Specifically, we observed that understandability is a QA that continually deteriorates during systems evolution [22], because based on the way it is calculated [7] it is inversely proportional to the growth of properties such as complexity (measured by number of methods) and design size (measured by number of classes). On the other hand, in the cases when understandability is increasing, we observe a negative relationship with the rest of the non-critical QAs, again because of the way that both understandability and non-critical-QAs are calculated. Concluding, we believe that the decrease of understandability in our study is not the result of

[7] http://findbugs.sourceforge.net/bugDescriptions.html#SIC_INNER_SHOULD_BE_STATIC

[8] http://findbugs.sourceforge.net/bugDescriptions.html#MS_OOI_PKGPROTECT

[9] http://findbugs.sourceforge.net/bugDescriptions.html#UR_UNINIT_READ

[10] http://findbugs.sourceforge.net/bugDescriptions.html#NP_NULL_ON_ SOME_PATH

[11] http://findbugs.sourceforge.net/bugDescriptions.html#BX_UNBOXING_ IMMEDIATELY_REBOXED

[12] http://findbugs.sourceforge.net/bugDescriptions.html#SWL_SLEEP_WITH_LOCK_HELD

explicit trade-offs, but simply, the natural effect of system growth.

5.3 Implications for Practitioners and Researchers

Firstly, by investigating our results, architects and software engineers can become aware of the most probable side-effects that the enhancement of one QA might have to another, in the sense that some trade-offs may be performed unintentionally. For example, by making developers aware of the fact that when fixing bugs related to *security,* they usually introduce additional bugs related to *correctness*, would make them consider possible ways to avoid such side-effects. Similarly, architects can also benefit from the identified trade-offs, as a source of potential threats to QAs, enabling them to: (a) monitor the potentially harmed QAs, and (b) identify concrete QA compromises earlier, so as to employ the necessary countermeasures.

Secondly, the results of the study suggest that CES differ from other application domains in terms of the actual trade-offs, and in terms of trade-offs between QAs not being bi-directional. Therefore, we strongly advise both researchers and practitioners to: (a) reflect on the direction of trade-offs, when reasoning about the interplay of QAs (e.g., the improvement of one QA affects another QA negatively, but not vice-versa), and (b) to take into account the application domain when investigating trade-offs.

Finally, the results of the study suggest that the outcome of investigating trade-offs at the level of the implemented architecture are intuitively correct, as they align with architecting principles. Therefore, we induce researchers to consider source code artifacts, when exploring trade-offs between QAs.

6. THREATS TO VALIDITY

In this section we present and discuss construct validity, reliability, as well as external validity for this study. Internal validity is not applicable, as the study does not examine causal relations. Construct validity reflects how connected are the object of study and the research questions. The reliability is related to whether the case study is conducted and presented in such way that others can replicate it with the same results. Finally, external validity deals with possible threats when generalizing the findings derived from sample to the entire population.

Concerning construct validity, one threat concerns the correctness of the formulae, proposed by Bansiya and Davis [11], for assessing the non-critical QAs. However, as described in the data collection section, the calculation had been validated through an empirical study involving experienced practitioners. Additionally, regarding FindBugs, we acknowledge that the list of bug patterns are by no means exhaustive, and additional bugs related to the investigated QAs could be used. However, to the best of our knowledge the used tool is among the most reputed in the community, and has adequate performance (see Section 3.3.1). Another threat is that effect size is not considered during the data analysis (Section 3.4), i.e., any positive or negative change in an attribute is considered the same, regardless of its magnitude. This measure was taken in order to avoid bias from specific projects to the entire domain.

In order to mitigate reliability, two different researchers were involved in the data collection, having all outputs double-checked. Furthermore, the same double-checking procedure happened during the data analysis. Finally, all primitive data can be reproduced by using the same bug detection tool (FindBugs, v3.0.0), for estimating critical QAs, and the QMOOD quality model calculations [11], for estimating non-critical QAs.

Finally, concerning external validity, we have identified four possible threats to the validity of our results. Firstly, we investigated a limited number of CESs, due to unavailability of critical embedded OSS implemented in Java. Thus, the inclusion of more CESs may differentiate the reported results. Additionally, modifications on the type and/or number of non-CES may slightly differentiate the results as well. Secondly, all software systems that were investigated are written in Java, while C/C++ is a more popular language for implementing CES; thus, there is a possibility that results are different for other object-oriented languages, as well as for other paradigms. Thirdly, due to the use of FindBugs and QMOOD, the reported results concern three critical and six non-critical QAs. Therefore, all discussions on the existence of possible trade-offs between critical and non-critical QAs, cannot be generalized to other QAs (e.g., reliability, changeability, etc.) without further investigation. Finally, our results cannot be generalized "as is" to trade-offs in the intended architecture, because we have analyzed trade-offs from the perspective of the implemented architecture, i.e. source code (including both intentional and unintentional trade-offs). In order to draw safe conclusions on the intentional trade-offs the architectural design of a system should be explored. For example, considering other points of view, such as risk analysis in the intended architecture [11].

7. CONCLUSION

One of the greatest challenges in engineering CESs is to guarantee critical QAs, which may pose hard constraints. This entails that complex trade-offs need to be made, either intentionally or unintentionally. In our study, we aimed at empirically investigating the interplay of QA and existence of quality trade-offs by analyzing source code through software evolution. For that we explored 9 QAs, measured from a total of 622 versions, obtained from 21 open source software projects.

Concerning CES, the results of the study imply the existence of possible trade-offs between critical QAs (correctness, security, and performance), as well as the fact that non-critical QAs (e.g., reusability, understandability, etc.) are usually compromised in favor of critical QAs. However, we have not observed critical QAs compromised in favor of non-critical QAs, for either CES or other application domains. Finally, we provide evidence on the fact that non-critical QAs are more often compromised than critical QAs.

As interesting pointers for future work, the results of our study highlight the relevance of investigating trade-offs involving critical QAs from a more detailed perspective, exploring the bug patterns entangled in the trade-offs, therefore, identifying possible connections between patterns and specific trade-offs. Furthermore, in order to support the high level findings presented in this study, it is interesting to consider additional QAs, especially critical ones. Finally, as an interesting replication we consider the execution of this study with projects written in other languages often used for developing embedded systems, such as C and C++.

8. ACKNOWLEDGMENTS

This work is financially supported by Brazilian funding agencies CAPES/Nuffic (Grant N.: 034/12), CNPq (Grant N.: 204607/2013-2), as well as the INCT-SEC (Grant N.: 573963/2008-8 and 2008/57870-9).

9. REFERENCES

[1] Abran, A., Moore, J.W., Bourque, P., Dupuis, R. and Tripp, L.L. 2014. *Guide to the Software Engineering Body of Knowledge*.

[2] Aguiar, A., Filho, S.J., Magalhães, F.G., Casagrande, T.D. and Hessel, F. 2010. Hellfire: A design framework for critical embedded systems' applications. In *Proceedings of the 11th International Symposium on Quality Electronic Design* (2010). ISQED'10. 730–737.

[3] Alhusain, S., Coupland, S., John, R. and Kavanagh, M. 2013. Towards machine learning based design pattern recognition. In *13th UK Workshop on Computational Intelligence* (2013). UKCI'13. 244–251.

[4] Alshammari, B., Fidge, C. and Corney, D. 2010. Security Metrics for Object-Oriented Designs. In *Proceedings of the 21st Australian Software Engineering Conference* (2010). ASWEC'10. 55–64.

[5] Ampatzoglou, A., Gkortzis, A., Charalampidou, S. and Avgeriou, P. 2013. An Embedded Multiple-Case Study on OSS Design Quality Assessment across Domains. In *Proceedings of the ACM / IEEE International Symposium on Empirical Software Engineering and Measurement* (2013). ESEM'13. 255–258.

[6] Ampatzoglou, A., Michou, O. and Stamelos, I. 2013. Building and mining a repository of design pattern instances: Practical and research benefits. *Entertainment Computing*. 4, 2 (Apr. 2013), 131–142.

[7] Bansiya, J. and Davis, C.G. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*. 28, 1 (2002), 4–17.

[8] Barney, S., Petersen, K., Svahnberg, M., Aurum, A. and Barney, H. 2012. Software quality trade-offs: A systematic map. *Information and Software Technology*. 54, 7 (Jul. 2012), 651–662.

[9] Barros, M. de O., Farzat, F. de A. and Travassos, G.H. 2014. Learning from optimization: A case study with Apache Ant. *Information and Software Technology*. 57, 1 (Aug. 2014), 684–704.

[10] Basili, V.R., Caldiera, G. and Rombach, H.D. 1994. Goal Question Metric paradigm. *Encyclopedia of Software Engineering*. Wiley & Sons. 528–532.

[11] Bass, L., Nord, R., Wood, W., Zubrow, D. and Ozkaya, I. 2008. Analysis of architecture evaluation data. *Journal of Systems and Software*. 81, 9 (Sep. 2008), 1443–1455.

[12] Bate, I. 2008. Systematic approaches to understanding and evaluating design trade-offs. *Journal of Systems and Software*. 81, 8 (Aug. 2008), 1253–1271.

[13] Buyens, K., Scandariato, R. and Joosen, W. 2009. Measuring the interplay of security principles in software architectures. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement* (2009). ESEM'09. 554–563.

[14] Chatzigeorgiou, A. and Stiakakis, E. 2013. Combining metrics for software evolution assessment by means of Data Envelopment Analysis. *Journal of Software: Evolution and Process*. 25, 3 (Mar. 2013), 303–324.

[15] Griffith, I. and Izurieta, C. 2014. Design pattern decay: the case for class grime. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2014). ESEM'14. 1–4.

[16] Hovemeyer, D. and Pugh, W. 2004. Finding bugs is easy. *ACM SIGPLAN Notices*. 39, 12 (2004), 92–106.

[17] Kitchenham, B. and Pfleeger, S.L. 1996. Software quality: the elusive target [special issues section]. *IEEE Software*. 13, 1 (1996), 12–21.

[18] Linares-Vásquez, M., Klock, S., McMillan, C., Sabané, A., Poshyvanyk, D. and Guéhéneuc, Y.-G. 2014. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps. In *Proceedings of the 22nd International Conference on Program Comprehension* (2014). ICPC'14. 232–243.

[19] Marwedel, P. 2010. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer Netherlands.

[20] Misra, S.C. and Bhavsar, V.C. 2003. Relationships Between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality. In *Proceedings of the Computational Science and Its Applications (2003)*. ICCSA'03. 724–732.

[21] Oliveira, M.F.S., Redin, R.M., Carro, L., Lamb, L. and Wagner, F. 2008. Software Quality Metrics and their Impact on Embedded Software. *5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software* (2008). MOMPES'08. 68–77.

[22] Penta, M. Di, Cerulo, L. and Aversano, L. 2009. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology*. 51, 10 (Oct. 2009), 1469–1484.

[23] Perry, D.E. and Wolf, A.L. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*. 17, 4 (Oct. 1992), 40–52.

[24] Del Rosso, C. 2008. Software performance tuning of software product family architectures: Two case studies in the real-time embedded systems domain. *Journal of Systems and Software*. 81, 1 (Jan. 2008), 1–19.

[25] Runeson, P., Host, M., Rainer, A. and Regnell, B. 2012. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Blackwell.

[26] Zaman, S., Adams, B. and Hassan, A.E. 2011. Security versus performance bugs. *Proceeding of the 8th working conference on Mining software repositories* (2011). MSR'11. 93–102.

[27] Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J.P. and Vouk, M.A.S.E.I.T. on 2006. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*. 32, 4 (2006), 240–253.