Applying patterns in embedded systems design for managing quality attributes and their trade-offs

Feitosa, Daniel

university of
groningen

Applying Patterns in Embedded Systems Design for managing Quality Attributes and their Trade-offs

PhD thesis

to obtain the degree of PhD at the
University of Groningen
on the authority of the
Rector Magnificus Prof. E. Sterken
and in accordance with
the decision by the College of Deans.

This thesis will be defended in public on

Friday 25 January 2019 at 11.00 hours

by

Daniel Feitosa
born on 6 January 1988
in Salvador, Brazil

Supervisors
Prof. P. Avgeriou
Prof. E. Y. Nakagawa

Co-supervisor
Dr. A. Ampatzoglou

Assessment committee
Prof. Y.G. Guéhéneuc
Prof. C. Izurieta
Prof. A.C. Telea

The research reported in this thesis has been conducted in the Software Engineering and Architecture group of the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence of the University of Groningen, The Netherlands. The research work was financially supported by the Brazilian and Dutch agencies Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and Netherlands Universities' Foundation for International Cooperation (NUFFIC), under grant number 034/12, Conselho Nacional de Desenvolviment Científico e Tecnológico (CNPq), under grant number 204607/2013-2, and Instituto Nacional de Ciência e Tecnologia em Sistemas Embarcados Críticos (INCT-SEC), under grant numbers 573963/2008-8 and 2008/57870-9.

Cover details: The geometric forms on the front and back side of the cover refer to two different implementations of similar features of a robotic face. This is an analogy to the possibility of having different instantiations of one same design pattern, which may express different levels of one or more quality attributes. The curve delineated by a dark shade of red is an analogy to measurements quality levels in a time series. Finally, the font used in the cover is named *Roboto*, by Christian Robertson.

Apllying patterns in embedded systems design for managing quality attributes and their trade-offs
Daniel Feitosa

ISBN: 978-94-034-1375-4 (printed version)
ISBN: 978-94-034-1374-7 (electronic version)

Abstract

Embedded systems comprise one of the most important types of software-intensive systems, as they are pervasive and used in daily life more than any other type, e.g., in cars or in electrical appliances. When these systems operate under hard constraints, the violation of which can lead to catastrophic events, the system is classified as a **critical embedded system** (CES). The quality attributes related to these hard constraints are named **critical quality attributes**. For example, the performance and security of the software for cruise-control, automatic braking, or self-driving in a car are critical as they can potentially relate to harming human lives.

Despite the growing body of knowledge on engineering CESs, there is still a lack of approaches that can support the design of CES, while managing critical quality attributes and their trade-offs with noncritical ones. To address this gap, this dissertation explored the state of research and practice on designing CES and managing quality trade-offs, identified approaches to improve the design of CES with regards to managing quality attributes and their trade-offs, and empirically investigated the merit of these approaches.

To investigate the state-of-practice, we explored the actual trade-offs between quality attributes (both critical and noncritical) in real systems. The results showed that trade-offs favor certain critical quality attributes against noncritical ones (e.g., security for extendibility) or other critical quality attributes (e.g., correctness for performance). In addition, these trade-offs between critical and noncritical quality attributes are more recurrent in the domain of CESs. These observations suggest that certain trade-offs are systemic (i.e., recurrent in the system) and may have great impact on the level of quality attributes. Therefore, it is of paramount importance to investigate approaches that can tackle both critical and noncritical qualities and support managing the trade-offs between them.

To explore the state-of-research, we conducted a systematic mapping study

(SMS) to explore approaches that had been proposed and used for CES design. Results of the SMS showed that multiple approaches have been proposed, focusing on a variety of specific challenges posed by different types of CESs. Evidence suggests that CESs have been growing both in terms of size and complexity. In addition, noncritical features, such as GPS and infotainment subsystems, are also being integrated, sometimes sharing resources (e.g., communication medium) with critical features, which leads to a new class of challenges related to these mixed levels of criticality. To tackle issues arising from this growth, several solutions (e.g., component-based approaches and software patterns) focus on improving design-time quality attributes, such as reusability and maintainability, while guaranteeing critical quality attributes. These findings provide further evidence on the necessity of addressing both critical and noncritical quality attributes during CES design.

After understanding the problem through the state of research and practice, the goal was to identify potential solutions that would support the management of trade-offs between QAs. Among the approaches identified during the SMS, some studies suggest using software patterns during CES design. Although literature shows that software patterns may affect different quality attributes, these effects have not been extensively explored empirically, especially regarding the correlation between critical and noncritical qualities. Due to their potential to support managing quality attributes, we decided to focus the PhD project to the use of software patterns, in particular GoF design patterns. For that, we conducted two empirical studies (case studies) to investigate how GoF design pattern affects three of the most common critical quality attributes, namely security, correctness, and performance.

In the first case study, five popular and non-trivial open-source software projects were considered to investigate the correlation between the application of GoF design patterns and the three quality attributes, which are assessed through static analysis. The results suggest that classes not participating in any pattern are correlated with lower quality levels. However, classes participating in patterns with more complex structure and pattern roles that are more change-prone are also more likely to be associated with lower quality levels. In the second study, dynamic analysis was exploited to assess and investigate one aspect of performance, namely energy efficiency, which has gained notorious attention from both practitioners and researchers in the last years. The results suggest that although a pattern solution tends to consume more energy than a non-pattern solution, certain design-time properties of a pattern instance (e.g., number of message calls, or method size) have considerable impact on their effect. In particular, results showed that large methods and/or methods with high number of method invocations were correlated with higher energy efficiency.

The results of both case studies suggest that design patterns are potential solutions for managing quality attributes. However, their impact on quality attributes is not uniform and, therefore, it is highly important to understand parameters that may affect it. One dominant parameter is the pattern instantiation. Similarly to how the design of a system may decay in comparison to its original architecture, design pattern instances can drift from their original implementation as the software evolves and additional functionality is added. This phenomenon is known as pattern grime, and understanding its consequences to the results observed in the aforementioned studies is vital for getting a comprehensive picture of the benefits and impairments of applying GoF patterns in CES development. For that, we conducted two case studies to investigate how pattern grime evolves and the relationship between its accumulation and levels of the three critical quality attributes addressed in this dissertation.

The first case study investigated the extent of the relationships between the accumulation of grime in pattern instances and various related factors: (a) projects, (b) pattern types, (c) developers, and (d) the structural characteristics of the pattern participating classes. The results suggest that pattern grime tends to increase linearly, and that it is likely independent of project but dependent of pattern type and developer. The second case study focused on examining the correlation between three forms of pattern grime (organizational, modular, and class) and the levels of performance, security and correctness. The results suggest that pattern grime is related to the depreciation of the three quality attributes in pattern instances. However, no strong evidence is observed on organizational grime. Furthermore, developers accumulate grime at different rates, and higher rates are mostly associated with lower quality levels. Finally, particular patterns, e.g., Factory Method, are associated with higher amounts of grime and lower quality levels.

Samenvatting

Geïntegreerde systemen zijn een van de meest belangrijke soorten software-intensieve systemen, omdat ze meer dan andere type systemen, gebruikt worden in vele aspecten van het dagelijks leven, bijvoorbeeld in auto's of elektrische apparaten. Wanneer deze systemen onder strenge restricties functioneren, d.w.z. restricties van dusdanig strenge aard dat schending van de restricties tot catastrofale gebeurtenissen kan leiden, wordt het geïntegreerde systeem geclassificeerd als een **kritieke toepassing** (ofwel *CES: Critical Embedded System*). De kwaliteitsattributen die gerelateerd zijn aan de restricties worden **kritieke kwaliteitsattributen** genoemd. Prestatie en veiligheid zijn bijvoorbeeld kritieke kwaliteitsattributen van software voor onder andere cruisecontrol, automatische remsystemen of zelfrijdende autosystemen aangezien het falen van deze systemen mensenlevens in gevaar kan brengen.

Ondanks de toenemende kennis op het gebied van CES engineering, is er nog een gebrek aan benaderingen die het ontwerp van CES ondersteunen en tegelijkertijd de kritieke kwaliteitsattributen en diens wisselwerking met niet-kritieke kwaliteitsattributen kunnen beheren. Om tegemoet te komen aan deze behoefte werden in dit proefschrift de stand van het onderzoek en de praktijk in CES-ontwerp, en het beheer van wisselwerkingen tussen kwaliteitsattributen bestudeerd. Voorts werden er benaderingen geïdentificeerd ter verbetering van het CES-ontwerp en het beheer van de wisselwerking tussen kwaliteitsattributen. De voordelen van de geïdentificeerde benaderingen werden onderzocht middels empirisch onderzoek.

Om de stand van de praktijk te onderzoeken, hebben we de daadwerkelijke wisselwerkingen tussen kwaliteitsattributen (zowel kritieke als niet-kritieke) in reële systemen bestudeerd. De resultaten toonden dat bepaalde kritieke kwaliteitsattributen in wisselwerkingen geprefreerd worden boven niet-kritieke kwaliteitsattributen (bijvoorbeeld veiligheid boven uitbreidbaarheid). Daarnaast zijn deze afwegingen tussen kritieke en niet-kritieke kwaliteitsattributen meer terugkerend in het domein

van kritieke toepassingen. Deze waarnemingen suggereren dat bepaalde compromissen stelselmatig zijn, d.w.z. dat ze wederkerend zijn in het systeem en grote invloed kunnen hebben op het niveau van kwaliteitsattributen. Daarom is het van het grootste belang om benaderingen te onderzoeken die zowel kritieke als niet-kritieke kwaliteitsattributen aankunnen en het beheer van de wisselwerking tussen beide ondersteunen.

Om de stand van onderzoek te bestuderen hebben we een *systematic mapping study (SMS)* uitgevoerd om voorgestelde en gebruikte benaderingen voor CES-ontwerp te onderzoeken. De resultaten van de SMS toonden dat meerdere benaderingen zijn voorgesteld, gericht op een verscheidenheid aan de specificieke uitdagingen die verschillende soorten CES bieden. Onderzoeksresultaten suggereren dat CES zijn gegroeid in termen van zowel grootte als complexiteit. Daarnaast worden niet-kritieke functies, zoals GPS en infotainment subsystemen, ook geïntegreerd, waarbij systeemelementen (bijvoorbeeld het communicatiemedium) gedeeld worden met kritieke functies. Dit leidt tot nieuwe uitdagingen op het gebied van de verschillen in kritieke niveaus van functies. Er zijn verscheidene oplossingen die uitkomst bieden aan de uitdagingen die voortkomen uit deze ontwikkeling. Deze oplossingen (zoals op componenten gebaseerde benaderingen en softwarepatronen) richten zich op verbetering van kwaliteitsattributen op het gebied van ontwerp en tijd, zoals herbruikbaarheid en onderhoud, terwijl ze kritieke kwaliteitsattributen garanderen. Deze bevindingen onderschrijven de noodzakelijkheid van het bestuderen van kritieke en niet-kritieke kwaliteitsattributen tijdens CES-ontwerp.

Na bestudering van de stand van praktijk en onderzoek omtrent het probleem, was het doel om potentiële oplossingen te identificeren die het beheer van wisselwerkingen tussen kwaliteitsattributen zouden kunnen ondersteunen. Uit de SMS waren benaderingen voortgekomen die het gebruik van softwarepatronen tijden CES-ontwerp voorschreven. Hoewel de literatuur aantoont dat softwarepatronen mogelijk verschillende kwaliteitsattributen beïnvloeden, zijn deze effecten, met name op het gebied van de correlatie tussen kritieke en niet-kritieke kwaliteitsattributen, niet extensief empirisch onderzocht. Omdat softwarepatronen potentie bieden in het ondersteunen van het beheer van kwakliteitsattributen, hebben wij besloten het proefschrift te richten op het gebruik van softwarepatronen, en GoF ontwerppatronen in het bijzonder. Hiervoor hebben wij twee empirische studies (casussen) uitgevoerd om te onderzoeken welke invloed GoF ontwerppatronen uitoefenen op drie veelvoorkomende kritieke kwaliteitsattributen: veiligheid, correctheid en prestatie.

In de eerste casus werden vijf populaire en non-triviale open-source software projecten geselecteerd om de correlatie te onderzoeken tussen de toepassing van GoF ontwerppatronen en de drie kwaliteitsattributen, die beoordeeld worden door statische analyse. De resultaten suggereren dat klassen die geen deel uitmaken van patronen gecorreleerd zijn met lagere kwaliteitsniveau's. In de tweede casus werd dynamische analyse gebruikt om een aspect van prestatie te bestuderen en beoordelen, namelijk energie-efficiëntie. Dit aspect heeft recentelijk veel aandacht gekregen van zowel onderzoekers als praktijkdeskundigen. Over het algemeeen verbruikt een oplossing op basis van patronen meer energie dan een oplossing die niet gebaseerd is op patronen. De resultaten suggereren echter dat bepaalde ontwerptijd eigenschappen van een patrooninstantie, bijvoorbeeld het aantal berichtoproepen of de omvang van de methode, een substantiële invloed hebben op het effect van een op patronen gebaseerde oplossing. De resultaten toonden met name aan dat grote methodes en/of methodes met een hoog aantal methode aanroepen gecorreleerd waren aan hogere energie-efficiëntie.

De resultaten van beide casussen suggereren dat ontwerppatronen uitkomst bieden voor het beheer van kwaliteitsattributen. De uitwerking van ontwerppatronen op kwaliteitsattributen is echter niet uniform, derhalve is begrip van de parameters die de uitwerking kunnen beïnvloeden uiterst belangrijk. Een belangrijke parameter is de instantiëring van het patroon. Evenzo een systeemontwerp mogelijk verslechtert in vergelijking met zijn originele architectuur, kunnen instanties van ontwerppatronen langzaam gaan afwijken van hun oorspronkelijke implementatie terwijl de software evolueert en aanvullende functionaliteiten toegevoegd worden. Dit fenomeen staat bekend 'pattern grime' en heeft consequenties voor de resultaten in de eerder genoemde studies. Derhalve is begrip van dit fenomeen essentieel om een volledig beeld van de voordelen en beperkingen van toepassing van GoF pattronen in CES ontwikkeling te schetsen.

In de eerste casus onderzochten we de relatie tussen de opbouw van *pattern grime* in patrooninstanties en verscheidene gerelateerde factoren: (a) projecten, (b) patroontypes, (c) ontwikkelaars, en (d) structurele kenmerken van de patroondeelnemende klassen. De resultaten suggereren dat *pattern grime* vaak lineair toeneemt en waarschijnlijk onafhankelijk is van de factor project, maar juist afhankelijk is van patroontype en ontwikkelaar. De tweede casus was gericht op het bestuderen van de correlatie tussen drie soorten *pattern grime* (organisatorisch, modulair en klasse) en de prestatie-, veiligheids-, en correctheidsniveaus. De resultaten suggereren dat *pattern grime* gerelateerd is aan de waardevermindering van de drie kwaliteitsattributen in patrooninstanties. Er is echter geen sterk bewijs waargenomen in organisatorische *pattern grime*. Bovendien verzamelen ontwikkelaars *pattern grime* in verschillende tempo's en hogere snelheden worden veelal geassocieerd met lagere kwaliteitsniveaus. Tot slot worden bepaalde patronen, zoals Factory Method, geassocieerd met grotere hoeveelheden pattern grime en lagere kwaliteitsniveaus.

Contents

List of Figures

List of Tables

Acknowledgements

Moving from Brazil to Groningen was an exciting moment, as so many things in my life were about to change. Throughout these past years, I got to work on challenging and interesting topics, meet new friends and colleagues, and share so many experiences. At the end of this journey, I am glad to realize that the excitement never really faded away.

First, I want to express my deepest gratitude to my supervisors Paris Avgeriou, Elisa Y. Nakagawa and Apostolos Ampatzoglou. I grew scientifically, academically and personally thanks to you. I read once that every PhD supervision is unique, as the different personalities of supervisor and student naturally leads to different styles of interaction and decision-making processes altogether. From a student perspective, I can say that my relationship with each one of you was unique, and equally nurturing nevertheless. It was a privilege to learn from you, with you, and to share so many experiences. Your wisdom and kindness will continue to be a source of inspiration to me.

I would like to thank the members of the assessment committee, the professors Alexander C. Telea, Yann-Gaël Guéhéneuc and Clemente Izurieta, for their valuable reviews. It was reassuring to read your comments and insightful suggestions.

It was a great pleasure to be part of the Software Engineering and Architecture group. The high level and diversity of the research conducted within the group led to countless, enriching discussions. Not to mention the great scientific effort put into brainstorming about "the next big thing" and on-the-fly protocols to investigate daily-life assumptions empirically; the gatherings, especially the BBQs, were priceless.

The work and academic environments were always pleasant and fueled with knowledge. For that, I want to first thank the great scientists and colleagues Nicolai Petkov, Alexander Chatzigeorgiou, Michael Wilkinson, Michael Biehl, George Azzopardi, Vasilios Andrikopoulos, Dimka Karastoyanova, Mircea Lungu, Tijs van der Storm, and Andrej Zwitter. I am also grateful for my collaborations with colleagues and students, which culminated in part of the work presented in this thesis. Thank you, Katia R. Felizardo, Frank J. Affonso, Hugo Andrade and Rutger Alders.

In a playful way, I often got asked: ”Why would you leave Brazil? Don’t you miss the weather, food, etc? Don’t you miss ‘home’?” Although I miss some things every now and then, I feel fortunate to recognize that I never really missed home; because so many places felt just as *gezellig*.

I want to thank my PhD colleagues and friends, my international family. Among Brazilians, Chinese, Cypriots, Dutch, Germans, Greeks, Italians, Iranians, Irish, Maltese, Mexicans, Romanians, Russians, and Spaniards, I experienced that nationality is quite a fun topic but hardly a descriptor of who one really is. Rafael and Danilo, thank you for hosting us during the first couple of weeks. The transition to living abroad would not had been as smooth nor fun without you guys. Bas, Marc and Mark, thank you for the great time while starting to learn and experience the Dutch life & culture. In between university, student house and nights out, thank you for sharing your knowledge, enthusiasm and silly thoughts: Sofia & Paschalis, Nicola & Arianna, Ugo & Tiziana, Laura (Robles and Fiorini), Manuel & Kitty, Astone & Jiapan, Estefan’ia, Sara, Niki & Amar, Dimitra, Emilia & Thomas, Octavio & Emilia, George & Charmaine, Andreas, Christian, Zengyang, Areti, Elvira, Chen, George, Darius, Anja, Héctor, Jie, Amaranta, Indira and Donna. In one way or another, my days were more lively because of you.

I want to express my gratitude to my friend, Kitty de Vries, for the great help creating the *samenvatting* of this thesis, and providing my much requested (and mirthful) receipt, which I almost added as an appendix. Most important, I want to thank you and Manuel for introducing those who would become my Dutch family: Ralf, Sander, Raymond, Frederique, Robert, Graz(iela), Rianne, Rafael and Rachel. Thank you all for the fellowship and for helping navigating the ‘intricacies’ of the Dutch culture.

Visiting home and having a good rest is important to re-energize and boost productivity, but traveling to Brazil for holidays and vacations was not so straightforward. In between summer breaks, Christmas and long weekends, I want to thank *Θεία* Nana & *Θείος* Kostis, *Θεία* Tania & *Θείος* Giorgos, *Γιαγιά* Sofia & *Παππού* Vasilis, Zia Margherita & Zio Ignazio, for having us as if we were part of your own family. For the great times in Greece, I also want to thank Sofia, Dafini, Hlektra, Nopi, Argyro, Thanasis, Pavlos, Nikos, Michalis, and Dimitris.

Despite the geographical distance, Brazil did not feel so far after all. Living together during the college time was precious (and hilarious), but seeing that distance did not change a thing is indescribable. Big (Diego), Chato (Guilherme) & Simone, Cirilo (Marcos), Garça (Luís) & Larissa, Gueixa (Mario) & Paula, Leza (Lucas), May(ara), Mi(chele) & Anton, Neve (Alfredo), Paquito (Mauricio), Piriguete

(Gabriel), Tia (Santiago), and Zero Dois (Eder), thank you for making the prologue of my scientific career one of the most memorable periods of my life.

To my paranymphs, Sabrina and Antonios, I want to thank you for being a constant throughout these past years, from scientific discussions to silly (though Copernican) chats. Sis, you were one of my first friends. Many of the traits that I appreciate in myself, I developed thanks to you. Antoni, you are one of the first friends that I made abroad and one of the nicest persons that I know.

To my wife, Renata, I leave one of the warmest thankful wishes. We got together not too long before we moved to Groningen and the PhD trajectory overlapped largely with the beginning of our journey. Living this moment with you, the way we did, was embellishing. Your kindness was encouraging, your endurance was inspiring, and your companionship was key to be were I am right now. I want to extended this gratitude to my in-laws, Sebastião, Elizabeth and Bruna, who were utterly lovable despite the distance.

To my *gezin*, Mom (Marciana), Leon Fagner, Sabrina and Bruno, I want to thank you for the unconditional love and support. I grew up seeing the world through your eyes, and it was breathtaking, troubles and all. For a long time, we lived spread around the country; now, around countries. Yet, somehow, meeting you or even talking to you is all it takes to transport me back, live all those memories again, and get back in time for some new ones. Mom, thank you for making everything possible, your strength is unmatched.

At last, I want to leave a testament of my love and gratitude for someone that, unfortunately, will not be able to read nor listen these words. I have shared so little time with you, but that was enough to ignite my passion and respect for learning and science. Most of my memories are of you explaining simple things, how and why they worked, how and why they were relevant. However, I would not realize it all until you passed away and Mom wisely said: “The most important thing that your father left to you was a lesson: to value knowledge and learning”. Thank you, Joaquim Duarte Feitosa. Thank you; Dad.

Daniel Feitosa
Groningen
January 4, 2018

Chapter 1

Introduction

This chapter elaborates on the main concepts of this PhD project, so as to describe the context of the study and the research design. Section 1.1 focus on the application domain, i.e., critical embedded systems, whereas Sections 1.2 and 1.3 focus on the main software engineering aspects that are being covered, i.e., software quality and patterns, respectively. Section 1.4 provides an overview of the problem statement that motivated this work, how the solution is decomposed into smaller parts, and the research methods that have been used in the project. Finally, Section 1.5 describes the organization of the remainder of this dissertation.

1.1 Critical Embedded Systems

Embedded systems (ES) have limited resources (e.g., processing power, memory) and are tailored to provide a particular functionality (Heath, 2002). However, the notion of a "limited resource" has changed drastically since ES emerged in the 1960s. More specifically, systems have become more powerful in terms of hardware, while the range of applications in which they can be used, has grown substantially.

Moreover, because ES are designed to be compact and efficient, they can be very attractive to industrial applications. Especially during the recent decades, we have witnessed the widespread industrial adoption of ES, which, on the one hand can reliably meet non-functional requirements (e.g., performance, reliability), while on the other hand demand less effort for their design and development. The range of applications increased and ES became ever more ubiquitous in daily life, to an extent that they are recognized as a strategic sector; as such both industrial and academic institutions devote great effort in developing expertise in this field (Helmerich et al., 2005; Ollila et al., 2004; Pétrissans et al., 2012; Thompson et al., 2017a). Currently, ES are widespread in all main industrial sectors (e.g., manufacturing, automotive, healthcare, rail and aerospace industries) and innovations such the Internet of Things is showing that ES are growing to become the main channel for information processing (Marwedel, 2011; Thompson et al., 2017b).

Critical Embedded Systems (CES) are a type of ES in which runtime errors can potentially be catastrophic, causing serious damage to the environment or to human lives, or non-recoverable material and financial losses (Aguiar et al., 2010; Medikonda and Panchumarthy, 2009). CES 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 and health appliances (Marwedel, 2011; Thompson et al., 2017b).

Engineering CES is particularly challenging since it needs to guarantee the satisfaction of various critical qualities such as security and safety. Although these challenges are being continuously addressed by research efforts, the problem space continuously broadens as the applicability of CES expands. Furthermore, with the continuous improvement of embedded hardware, the overall performance of the system also improves, which drives technological innovation around ES. The added complexity to CES gives way to new kinds of design challenges in CES projects, such as continuous integration and continuous delivery (Haghighatkhah et al., 2017; Pelliccione et al., 2017). Therefore achieving the desired qualities in CES becomes a moving target with an ever-expanding design space.

This dissertation aims at supporting the design process of CESs, by focusing on important quality requirements. The emphasis on software quality is essential to the success of any software-intensive system and the driver of the system design process (Bass et al., 2012). Achieving the correct functionality is imperative; but it does not guarantee that the system expresses the desired quality, which is often the reason for redesign activities (Bass et al., 2012). The next section elaborates further on software quality, and the types of quality attributes that are of paramount importance for the domain of CES.

1.2 Software Quality

During the design process, satisfying the functional requirements is not the most difficult part; one can get them right sooner or later. But meeting the non-functional requirements, i.e. the quality attributes, is much more challenging (Bass et al., 2012; Suryn, 2014). Quality is not only the determinant factor of the success of a system, but it often dictates its redesigning. However, stakeholders may perceive and explain differently what the quality of a product entails, while quality attributes are notoriously difficult to express in a SMART[1] way.

The previous section hints towards the fact that functionality and quality are orthogonal. For example, a mobile phone may be able to successfully make a call but users may be dissatisfied because of its performance or usability. The differ-

[1] Specific, Measurable, Assignable, Realistic, Time-related

ent aspects that qualify a product became known as quality attributes, and several models were devised to define and organize them, e.g., McCall's (McCall, 1977), Boehm's, (Boehm and In, 1996), Dromey's (Dromey, 1995), FURPS (Grady, 1992), and ISO/IEC 25010 (ISO/IEC, 2011); including efforts directed to particular domains, such as ES (Miyashiro et al., 2015; Oliveira et al., 2013; Tamrabet et al., 2018). These efforts have led to conceptualizing quality attributes through quality scenarios, which are key to estimating, monitoring and improving quality of the designed product (Bass et al., 2012).

In the domain of CES, the satisfaction of multiple quality constraints must be guaranteed, which may be achieved through the design of a sound architecture and its validation against the necessary quality attributes. However, this is far from trivial, as it entails complex trade-offs. To a large extent, it concerns safeguarding the levels of critical against other noncritical qualities (Ampatzoglou, Gkortzis, Charalampidou and Avgeriou, 2013; Linares-Vásquez et al., 2014). As mentioned in the previous section, if a system fails to satisfy critical quality attributes, it may lead to catastrophic failures. Typical examples of critical quality attributes are performance, security and reliability. But noncritical quality attributes are becoming more and more high-priority in CES, especially design-time qualities like maintainability, reusability and testability.

Literature reports on various approaches and practices in the domain of CES seeking to satisfy critical quality attributes while also supporting noncritical ones that are often found in more complex CES (Bass et al., 2012; Suryn, 2014). One such prominent approach is software patterns, a well-known design practice addressing functional requirements while also considering the impact on quality attributes. The next section elaborates on the most prevalent kind of software patterns, namely design patterns, which is fundamental to this dissertation.

1.3 Design Patterns

Software patterns compile the collective experience of skilled practitioners and researchers on designing software systems. They piece together best practices and common solutions to recurring problems (Buschmann et al., 2007; Gamma et al., 1995), and have proven to be timeless and invaluable to industry. Since their inception, communities were built around them (e.g., Pattern Languages of Programs conference series and transactions) and a variety of patterns and collections of patterns were devised on various fields and applications domains, e.g., object-orientated design (Gamma et al., 1995), architecture (Buschmann et al., 2007), security (Fernandez-Buglioni, 2013).

Design patterns is the most well-known type of software patterns that was introduced in 1995 by Gamma et al. (1995), known as the Gang of Four (GoF). Their seminal book describes a catalog of 23 patterns, which became to be known as the GoF patterns. These patterns address common problems on object-oriented (OO) design and can be categorized according to the scope of the solution (i.e., class vs. object) or the purpose of the pattern (i.e., creational, structural or behavioral). GoF patterns were originally intended to be problem-solving mechanisms. However, their usefulness extended far beyond that, also serving for knowledge sharing and communication, as well as quality assessment and improvement (Hsueh et al., 2008; Zhang and Budgen, 2012).

The influence of design patterns on quality attributes is of great interest to both researchers and practitioners (Ampatzoglou, Charalampidou and Stamelos, 2013b; Bafandeh Mayvan et al., 2017). This interest stems from the fact that GoF patterns are widespread in software development and their instances may comprise a significant part of the systems: from 15% to 65% of the classes (Ampatzoglou et al., 2015; Khomh et al., 2009). However, the state of research suggests that the effect of patterns on software quality is not uniform, and depends on a number of parameters (Ampatzoglou, Charalampidou and Stamelos, 2013b). Moreover, research on the effect of GoF patterns on quality attributes that are vital to CES, such as security and performance, is fairly limited (Bafandeh Mayvan et al., 2017). In other words, there is very little guidance for CES researchers in using GoF patterns regarding their effect on critical and noncritical quality attributes.

This dissertation explores design patterns as the main practice to improve the design of CES through investigating their relationship with quality attributes relevant to the design of CES. The next section elaborates on the research design of this dissertation, describing the problem statement and how it is addressed through the work carried out in the PhD project.

1.4 Research Design

The research project reported in this dissertation originates from a main problem, which is presented in Section 1.4.1. To identify and plan the research activities that were necessary to address this problem, we adopted the Design Science framework as defined by Wieringa (2014), which is described in Section 1.4.2. The breakdown of the research design and the research process are detailed in Section 1.4.3. The last section describes how the research activities are reported in the remainder of this dissertation.

1.4.1 Problem Statement

Design errors in CES can potentially be catastrophic, in terms of causing serious damage to the environment or to human lives, or non-recoverable material and financial losses (Aguiar et al., 2010; Bate, 2008). Due to the criticality of such systems, the satisfaction of multiple quality constraints must be guaranteed, which is far from trivial. Consequently, the activity of quality assurance in CES mostly focuses on safeguarding the levels of critical quality attributes (Ampatzoglou, Gkortzis, Charalampidou and Avgeriou, 2013; Linares-Vásquez et al., 2014).

Although critical quality attributes are of paramount importance for the development of such systems, the decision-making process during the design of CES is not limited to them. Noncritical quality attributes such as maintainability and reusability may also be of priority to the software architects and act as key drivers for CES development. In fact, due to pressing business goals, noncritical quality attributes are becoming as important as critical ones. For example, in the near future, the automotive industry expects to deliver software updates on a daily basis. In such situations, reusability and maintainability of the software may become as important as its security and safety in order to enable this frequent updating. This further aggravates the problem of designing CES: not only is guaranteeing critical qualities very challenging, but noncritical qualities also need to be accommodated during design, while trade-offs between critical and noncritical qualities are particularly problematic.

In light of the aforementioned challenges, the main problem addressed in this dissertation is stated as follows:

> *Despite the growing body of knowledge for engineering CESs, their design process is still challenging. This is especially true due to their complexity, and hard requirements regarding critical quality attributes. Furthermore, it usually involves complex trade-offs for both critical and noncritical quality attributes. However, we currently lack practices that can support the design of CES while managing quality attributes and their trade-offs.*

There are two main consequences if this problem is not addressed. First, there is a greater chance of creating a design solution that is not optimal for the system under design, as it might not reflect the desired levels of quality. An inappropriate solution at the design stage will lead to the introduction of more problems during implementation through a cascading effect, when it becomes considerably more expensive to solve those problems. Second, the evolution of the system becomes more challenging. In particular, it can be harder to control the effect of changes on the quality of the system, since mechanisms for facilitating software evolution (e.g., conforming to design principles or patterns, avoiding design smells) might have been ignored or overlooked during the initial phases of CES development. For example, local changes may cause ripple effects to the rest of the system, causing a kind of domino effect every time a change is initiated. In short, maintenance and evolution activities, which are essential in future CES applications, can suffer greatly from poor design. Consequently, tasks as feature addition and bug fixing can show to be very costly, and this cost will accumulate until it becomes prohibitive for the development organization.

1.4.2 Design Science Framework

Design Science is a framework for strategizing research on information systems and software engineering domains. It was introduced by March and Smith (1995) which focused on developing or improving technological solutions for the benefit of stakeholders, who in turn define the scope of the research project. The envisioned framework was proposed for the domain of information systems but was sufficiently general to be employed in other disciplines, as it revolves around the idea of "designing things for specific purposes". In particular, the framework was later refined by Wieringa (2014) to also suit software engineering research. The "thing" being designed is referred to by Wieringa as an artifact and can be anything used in software and information systems, such as architectures, methods, and algorithms. The "purpose" lies in the context of activities performed for software and information systems, such as design and maintenance.

The core elements of Wieringa's version of the Design Science framework are depicted in Figure 1.1. There are two sources of information for the project, namely *social context* and *knowledge context*. The purpose and constraints (e.g., budget) for the artifact to be designed are provided by the *social context*, which encompasses the stakeholders' concerns. The theories, scientific knowledge, and existing designs (e.g., measurement tools) that are relevant to the problem being solved are provided by the *knowledge context*. In short, the *social context* drives the design activities, whereas the *knowledge context* steers the activities, in the sense that the existing knowledge shows what has to be synthesized (e.g., questions to be answered and tools to be created). By the end of the research project, the main object of interest for the stakeholders, the artifact, is transferred to the *social context*, but also to the *knowledge context*. The former benefits from the tailor-made solution, and the latter benefits from the added knowledge. In addition, all other knowledge generated in the process (e.g., answers to research questions) is also transferred to the *knowledge context*.

To use the available information to design the requested artifact, the framework

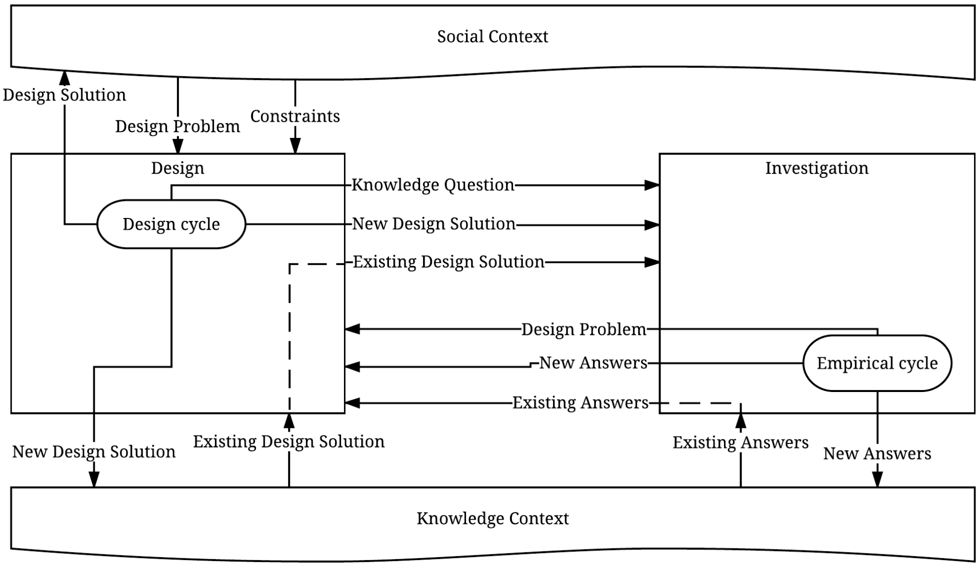


Figure 1.1: Design science framework, adapted from Wieringa (2014)

provides two main problem-solving mechanisms, namely *design cycle* and *empirical cycle*. The former aims at solving *design problems*, such as the main goal of the research project, and encompasses three activities: problem investigation, treatment design/identification, and treatment validation. To create solutions, not all information is explicitly available in the knowledge context and is queried in the form of *knowledge questions*, which are answered in an *empirical cycle.* In contrast to *design problems*, *knowledge questions* do not generate an artifact, but are essential nonetheless. The *empirical cycle* is proxy to well-known empirical methods, such as case studies, experiments and surveys, and thus its activities vary according to the question being asked.

Design science projects are iterative, alternating between two states: *design*, to solve design problems; and *investigation*, to answer knowledge questions. Therefore, design and empirical cycles are seamlessly nested to develop the artifact that was initially requested. Design cycles naturally raise knowledge questions (e.g., about available treatments or their validation), and for those without an answer in the knowledge context, an empirical cycle is performed. Conversely, such a cycle may require certain artifacts (e.g., measurement tools, classification schemes), which will lead to a design cycle if not available.

The workflow addressed in the framework is well-suited to strategize long-term research as PhD projects. In particular, it facilitates the decomposition of a problem into design problems that lead to new design problems or knowledge questions in an iterative manger; it thus supports the design of research questions and activities performed along the course of the PhD project. The initial design problem (i.e., stated in the problem statement — see Section 1.4.1) naturally leads to knowledge questions, which may in turn lead to new knowledge questions or new design problems.

1.4.3 Problem Decomposition

The decomposition of the problem addressed in this dissertation, is depicted in Figure 1.2. The problem statement describes the main challenge of the PhD, i.e., to identify or provide design practices that can support the design of CES while safeguarding quality attributes and managing their trade-offs. The gray boxes represent knowledge questions that were derived based on the Design Science Framework; these knowledge questions correspond to the research questions of this dissertation and are numbered as RQ_x (or $RQ_{x.y}$ in case of a decomposed research question). White boxes represent empirical cycles, i.e., scientific studies proposed to answer the knowledge questions and that are reported in the individual chapters of this dissertation. Moreover, thick, white arrows denote sequence, single-line arrows denote decomposition, and dashed arrows denote answers to research questions (obtained through an empirical cycle). Finally, dashed straight lines denote the separation between the three main parts of the work executed in the PhD: first, we explore the problem space through both state-of-the-practice and state-of-the-art studies; second, we study GoF patterns as one promising solution to support CES design; third, we extend the study of GoF patterns during software maintenance and evolution.

As a first step towards addressing the problem statement, we looked into the current *state-of-the-practice* by investigating the trade-offs between QAs (both critical and noncritical) in practice (RQ_1). Particularly we observed existing embedded systems through a case study, in order to investigate trade-offs in CES: (a) between critical QAs; (b) between noncritical QAs; and (c) between critical and noncritical QAs. In addition, we compared the results with quality trade-offs in other domains to explore potential similarities and differences. With this study we aim at characterizing the domain of CES with respect to quality attributes trade-offs, in order to explore relevant CES design practices that can support such trade-offs. The main outcome of this study is that indeed CES design places more emphasis on quality trade-offs compared to other application domains, and that the level of runtime qualities is non-negotiable, compared to design-time ones.

Having obtained knowledge of the state-of-the-practice, we moved on to RQ_2 where we explored the *state-of-the-art* on how CES are currently designed. In partic-

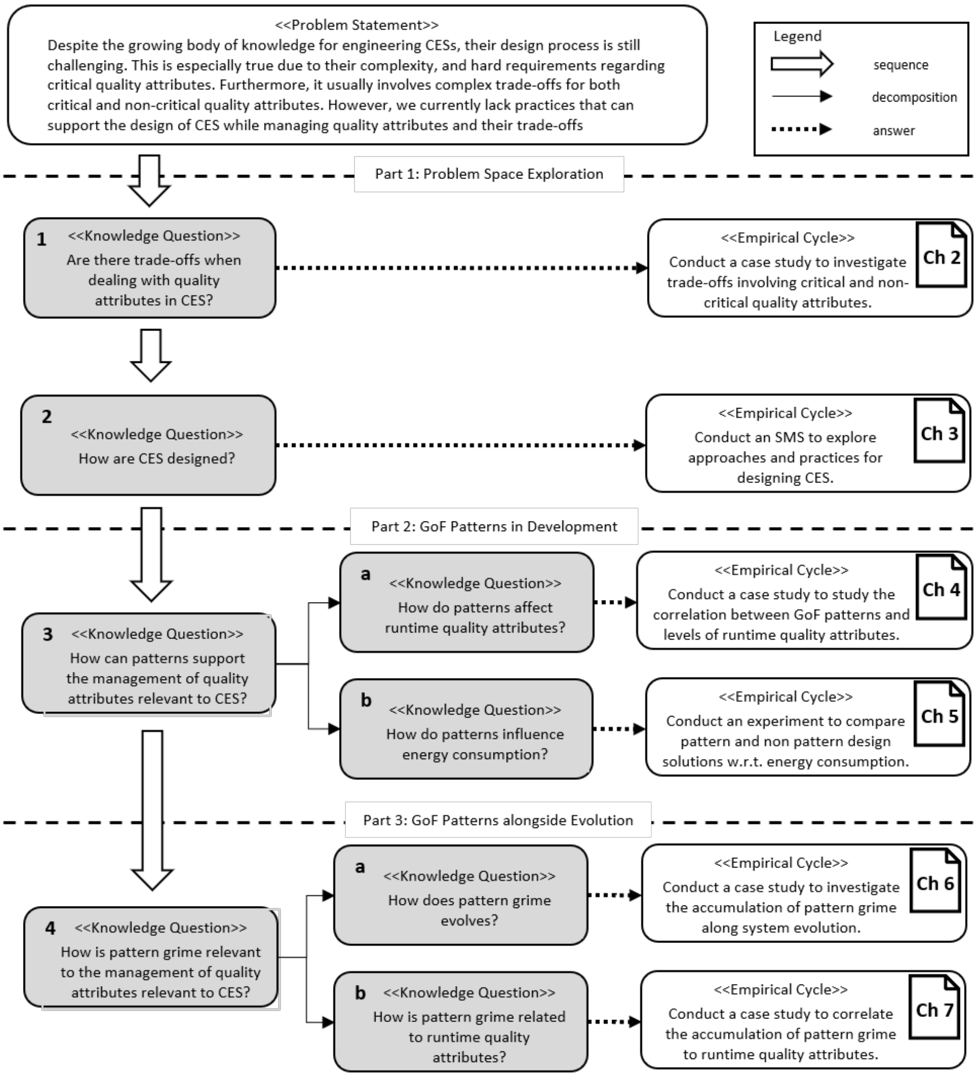

<<Problem Statement>>
Despite the growing body of knowledge for engineering CESs, their design process is still challenging. This is especially true due to their complexity, and hard requirements regarding critical quality attributes. Furthermore, it usually involves complex trade-offs for both critical and non-critical quality attributes. However, we currently lack practices that can support the design of CES while managing quality attributes and their trade-offs
Legend
sequence
decomposition
answer
Part 1: Problem Space Exploration
1 <<Knowledge Question>> Are there trade-offs when dealing with quality attributes in CES?
<<Empirical Cycle>> Conduct a case study to investigate trade-offs involving critical and non-critical quality attributes.
Ch 2
2 <<Knowledge Question>> How are CES designed?
<<Empirical Cycle>> Conduct an SMS to explore approaches and practices for designing CES.
Ch 3
Part 2: GoF Patterns in Development
3 <<Knowledge Question>> How can patterns support the management of quality attributes relevant to CES?
a <<Knowledge Question>> How do patterns affect runtime quality attributes?
<<Empirical Cycle>> Conduct a case study to study the correlation between GoF patterns and levels of runtime quality attributes.
Ch 4
b <<Knowledge Question>> How do patterns influence energy consumption?
<<Empirical Cycle>> Conduct an experiment to compare pattern and non pattern design solutions w.r.t. energy consumption.
Ch 5
Part 3: GoF Patterns alongside Evolution
4 <<Knowledge Question>> How is pattern grime relevant to the management of quality attributes relevant to CES?
a <<Knowledge Question>> How does pattern grime evolves?
<<Empirical Cycle>> Conduct a case study to investigate the accumulation of pattern grime along system evolution.
Ch 6
b <<Knowledge Question>> How is pattern grime related to runtime quality attributes?
<<Empirical Cycle>> Conduct a case study to correlate the accumulation of pattern grime to runtime quality attributes.
Ch 7

Figure 1.2: Problem decomposition overview

ular, we investigated systematically the quality attributes of interest in CES development and made an initial exploration on design approaches and practices that could be useful for safeguarding both runtime and design-time qualities. Understanding existing design approaches and practices is of paramount importance as they may, to some extent, be reusable, either in terms of ideas or even tools. The adoption of new design approaches and practices may also be facilitated if stakeholders are already acquainted with a related existing approach, thus reducing the learning curve. To answer this question, a systematic mapping study (SMS) was conducted to explore approaches and practices that have been proposed and used for CES design. Several design approaches and practices were analyzed in terms of their merits but also challenges, while we paid special attention to practices, principles and ideas that can be reused.

Among the design approaches and practices identified during the SMS, the practice of using software patterns during CES design seems to be one of the most promising in terms of managing quality attributes. The description of software patterns includes known consequences on the quality attributes (and well documented in the literature) and can be used to assess the overall impact of a design on quality attributes. Thus, we decided to explore the practice of applying software patterns in CES design, in particular GoF design patterns. On the one hand, current evidence shows that the effect of GoF pattern on *noncritical quality attributes* has been fairly investigated, and that it is not uniform. Various criteria (e.g., design characteristics) may influence the effect. On the other hand, there is a lack of evidence on how these patterns affect *critical quality attributes.* This leads to the next question to be answered (RQ_3), i.e. to investigate the influence of design patterns on critical QAs in order to identify the extent of the effects.

As a first step in investigating how GoF design patterns affect critical quality attributes, we selected three such common qualities, namely, performance, correctness, and security. Consequently, we conducted a case study ($RQ_{3.a}$) to assess the relationship between the presence of GoF design patterns and the level of these three quality attributes. In particular, we seek to explore how design pattern instances are correlated with violations of good coding practices associated to correctness, performance and security. The study considers approximately 13,000 classes retrieved from five nontrivial open-source projects, from which the violations are collected via static analysis. To fully investigate the underlying relationship, the classes are analyzed with regard to: (a) their participation in pattern occurrences, (b) the pattern category (c) the pattern in which they participate, and (d) their role within the pattern occurrence. This allows understanding in depth the details of the impact of patterns on critical qualities.

As a second step in our investigation, we conducted a second study ($RQ_{3.b}$), which aims at investigating the impact of GoF patterns on one performance indicator that has recently attracted the attention of both researchers and practitioners, i.e., energy consumption. This study considers pattern-participating methods (i.e., those that play a role within the pattern) and compares their energy consumption to the consumption of functionally equivalent alternative (non-pattern) solutions. The comparison is performed on 169 methods of two GoF patterns (namely State/Strategy and Template Method), retrieved from two well-known open source projects. This study also allowed us to use dynamic analysis as a complementary way to the static analysis performed for $RQ_{3.a}$, in order to explore whether the results would be aligned (triangulation of data collection methods).

The answers to RQ_3 suggest that the practice of applying design patterns is a promising solution to safeguard quality attributes in CES development as the effect of GoF patterns on critical quality attributes is controlled and deterministic; the right use of GoF patterns can help CES designers to strengthen the critical quality attributes in their systems. However, similarly to any other design artifact, design pattern instances tent to drift from their original implementation, as the software evolves and additional functionalities are added; this phenomenon is known as "pattern grime" (Izurieta and Bieman, 2013). Pattern grime results in degraded patterns instances that no longer hold the intended impact on critical quality attributes. Hence, the next question (RQ_4) focuses on the potential undesired effects of pattern grime. In particular, we explored the extent to which pattern grime can influence the impact of GoF patterns on critical quality attributes, thereby diminishing the benefits of applying the patterns in the first place (as observed for RQ_3). The investigation for this knowledge question is two-fold.

The first step ($RQ_{4.a}$) is to investigate the accumulation of pattern grime along system evolution. As pattern grime has been pointed out as one recurrent reason for the decay of GoF pattern instances, this study seeks to examine the existence of relations between the accumulation of grime in pattern instances and various related factors. In particular, it considers: (a) projects, (b) pattern types, (c) developers, and (d) the structural characteristics of the pattern-participating classes. For that, the study comprises the analysis of five industrial projects, implemented by 16 developers that provide a total of 2,349 pattern instances from eight different GoF design patterns.

The second step ($RQ_{4.b}$) is to investigate how the accumulation of pattern grime is related to levels of the three critical quality attributes that were studied for $RQ_{3.a}$, i.e. performance, security, correctness. To ease the analysis for answering this RQ, this study is a follow-up for the previous one. The same industrial software systems are considered in the investigation. The study seeks to correlate the accumulation of pattern grime with the accumulation of violations of coding practices (regarding each quality attribute) in pattern-participant classes. Moreover, it also seeks to analyze factors that might influence the observed correlations, in particular, projects, pattern types, and developers.

1.4.4 Empirical Research Methodology

The previous section broke down the problem statement addressed in this dissertation into knowledge questions. Each knowledge question is answered by following one or more empirical cycles, each corresponding to an empirical study conducted during the PhD. The empirical studies were designed based on the practices of evidence-based software engineering (EBSE), a paradigm advocated in the seminal work by Kitchenham et al. (2004). The approaches proposed in EBSE branch out from the more mature field of evidence-based medicine and have shown to be reliable research tools in improving software engineering research and practice.

Table 1.1 presents the research method used in each empirical study designed to answer the research questions posed in the PhD. The table also provides the reference to the section of the dissertation in which the design of the corresponding empirical study is presented. In the following, we describe these empirical methods and the context in which they were applied in the PhD.

Table 1.1: Overview of research methodology

Code	Knowledge Question	Empirical method	Described in
RQ_1	Are there trade-offs when dealing with quality attributes in CES?	Case study	Section 2.3
RQ_2	How are CES designed?	Systematic mapping study	Section 3.3
$RQ_{3.a}$	How do patterns deal with runtime quality attributes?	Case study	Section 4.3
$RQ_{3.b}$	How do patterns influence energy consumption?	Controlled experiment	Section 5.4
$RQ_{4.a}$	How does pattern grime evolves?	Case study	Section 6.3
$RQ_{4.b}$	How is pattern grime related to runtime quality attributes?	Case study	Section 7.3

Systematic Mapping Study (SMSs) and Systematic Literature Reviews (SLRs) have been broadly adopted as systematic research methods to aggregate knowledge (Kitchenham et al., 2004; Petersen et al., 2008). Both methods provide a systematic approach to reduce bias in reviewing a series of primary studies related to a common topic. Regarding the differences, SLRs are focused on the in-depth review of primary studies, allowing for synthesis of knowledge based on the findings in the investigated studies. SLRs demand considerable effort to review individual primary studies in depth and, thus, are more suitable to relatively narrow topics, where the amount of primary studies is manageable. Conversely, SMSs are focused on creating an overview of a certain topic, understanding how the knowledge is organized from the point of view of several facets. Such goals allow for reviews of a larger scale, e.g., an entire field of knowledge, in which the reviewing effort is distributed among a plethora of primary studies. *In the PhD project an SMS was applied to outline the state of the art on design approaches for CES in a broad sense, also characterizing the research effort in terms of application domain, addressed quality attributes, tooling, and maturity level.*

Controlled Experiment are empirical methods for studying phenomena under a controlled environment (Wohlin et al., 2012). This method requires isolation of the phenomenon under study but allows for precise manipulation of the subjects. By systematically reducing the confounding factors, experiments are suitable for investigating cause-effect relationships between different treatments, i.e., particular behavior of the isolated phenomenon. The assignment of treatments to subjects can vary depending on the number of factors (i.e., variables) and treatments (i.e., values) being studied. A common configuration has one factor with two treatments, for which the design can be completely randomized or crossover. In the former, the two treatments are randomly assigned to the subjects, whereas in the later, all subjects receive both treatments. *In the PhD project a controlled experiment was designed to investigate the extent to which GoF patterns can influence the energy consumption in nontrivial software. For that, a crossover design was selected to compare two treatments: a design solution using GoF patterns and an alternative (non-pattern) design solution.*

Case study is an empirical approach that provides the means to understanding a particular phenomenon in context (Runeson et al., 2012). Compared to other empirical approaches such as controlled experiments, surveys and action research, case studies allow for investigating a phenomenon in its environment (i.e., context) with considerably reduced to no interaction with the object of study. Case studies are more suitable to examining relationships and do not primarily aim at establishing causality. According to the actual purpose at hand, case studies can be *exploratory* if theory is induced by identifying patterns in the observations (i.e., inductive empirical research) or *explanatory* if a theory is confirmed or rejected through observations (i.e., deductive empirical research). They are also classified as *holistic* if the case is studied as a whole (i.e., unity of analysis is the case), or *embedded* if each case contains multiple units of analysis. *This empirical approach was broadly used in the PhD: to understand the problem space with regards to trade-offs between quality attributes in CES; to investigate in-depth the relationship between GoF patterns and critical quality attributes; and to study the limitations of GoF patterns with regards to pattern grime.*

1.5 Overview of the Dissertation

Chapters 2-7 are based on scientific work that has already been published in peer-reviewed venues. Each piece of work aims at answering one of the research questions presented in Section 1.4 and are described in the following paragraphs. Finally, Chapter 8 concludes this dissertation, summarizing the results obtained from all scientific work, recapping the answers to the research questions, and discussing opportunities of future work.

The scientific work presented in this dissertation is divided into three parts as presented in Table 1.2. The first part, comprised by Chapters 2 and 3, elaborates in the problem space exploration. Chapters 4 and 5 cover the second part, which focuses on exploring GoF patterns alongside development as a solution to support managing quality attributes in CES development. The last part, consists of Chapters 6 and 7, which investigate the benefits of applying GoF patterns alongside software evolution, and explore the limitations with respect to one key factor, namely pattern grime.

Table 1.2: Overview of dissertation

Research Question	**Chapter**
Part 1: Problem Space Exploration	
RQ_1: Are there trade-offs when dealing with quality attributes in CES?	Chapter 2
RQ_2: How are CES designed?	Chapter 3
Part 2: GoF Patterns in Development	
$RQ_{3.a}$: How do patterns deal with runtime quality attributes?	Chapter 4
$RQ_{3.b}$: How do patterns deal with energy consumption?	Chapter 5
Part 3: GoF Patterns alongside Evolution	
$RQ_{4.a}$: How does pattern grime evolves?	Chapter 6
$RQ_{4.b}$: How is pattern grime related to runtime quality attributes?	Chapter 7

Chapter 2 is based on a study that explores the interplay between quality attributes in CES. It is based on a peer-reviewed conference paper in the proceedings of the 11th International ACM SIGSOFT Conference on the Quality of Software Architectures (Feitosa et al., 2015). The study aimed at researching and discussing potential trade-offs between critical and noncritical quality attributes. I was the lead author, designing and executing the study. The co-authors participated in the data collection and analysis, as well as contributed to the revision of the published paper.

Chapter 3 reports on a peer-reviewed chapter published in the book “ENASE

2017: Evaluation of Novel Approaches to Software Engineering" (Feitosa, Ampatzoglou, Avgeriou, Affonso, Andrade, Felizardo and Nakagawa, 2018). This work comprises a systematic mapping study (SMS) investigating design approaches that are suitable for CES, analyzing them with regards to addressed critical quality attributes, application domain, provided tooling and type of evidence. I was the lead author in this publication, responsible for designing, orchestrating, and conducting the study. The co-authors contributed to the study design, supported one or more steps of the SMS, and reviewed the report.

Chapter 4 reports on a work accepted for peer-reviewed journal publication in Information and Software Technology (Feitosa, Ampatzoglou, Avgeriou, Chatzigeorgiou and Nakagawa, 2018). It describes a case study conducted to investigate the correlation between the presence of GoF patterns and critical quality attributes as assessed through static analysis of source code. I was the lead author, and the work division was similar to that in the aforementioned study. In addition, I was the designer and lead developer of the tool SSAP, used in the study.

Chapter 5 reports on a controlled experiment to investigate the influence of GoF design patterns on energy consumption. It is based on a peer-reviewed journal publication in the Journal Software: Evolution and Process (Feitosa, Alders, Ampatzoglou, Avgeriou and Nakagawa, 2017). The chapter provides evidence on the benefits of GoF patterns to energy consumption when used under appropriate circumstances. I was the lead author, designing the study and performing the analysis of the data, which was collected by a co-author under my supervision. The other co-authors assisted in the study execution and reviewing of the manuscript.

Chapter 6 is based on a peer-reviewed conference paper in the proceedings of the 18th International Conference on Product-Focused Software Process Improvement (Feitosa, Avgeriou, Ampatzoglou and Nakagawa, 2017b). It describes an industrial case study to investigate the limitations of the benefits provided by the use of GoF design patterns. In particular, this study aims at identifying how pattern grime (i.e., a form of design pattern deterioration) accumulates and how it correlates to levels of design-time quality attributes. I was the lead author, coordinating all activities from study design to reporting. I was also responsible for the development of spoon-pttgrime, a tool used in the study. The co-authors assisted all steps of the study and reviewed the manuscript.

Chapter 7 is the final piece of scientific work presented in this dissertation. It is based on a peer-reviewed journal publication in IEEE Access (Feitosa, Ampatzoglou, Avgeriou and Nakagawa, 2018) and presents another industrial case study, which complements the previous one by focusing on examining the correlation between the accumulation of pattern grime and levels of critical quality attributes as measured by static analysis of source code. I was the lead author and the work division is similar to that in the previous study. This chapter finalizes the collection of evidence to answer the research questions posed in the problem decomposition, leading to the final chapter of this dissertation.

Based on:

Daniel Feitosa, Apostolos Ampatzoglou, Paris Avgeriou, and Elisa Y. Nakagawa, (2015) *"Investigating Quality Trade-offs in Open Source Critical Embedded Systems,"* in Proceedings of the 11th International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA '15), Montreal, Canada, pp. 113–122, DOI:10.1145/2737182.2737190

Chapter 2

Investigating Quality Trade-offs in Open Source Critical Embedded Systems

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 noncritical qualities (e.g., reusability, understandability, etc.) may be compromised. In this chapter, 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, noncritical 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 noncritical ones in either CES or other application domains. Furthermore, quality trade-offs are more frequent among critical quality attributes, compared to trade-offs among noncritical quality attributes. Our study has implications for both practitioners when making trade-offs in practice, as well as researchers that investigate quality trade-offs.

2.1 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 (Marwedel, 2010). CESs are embedded systems in which design errors can potentially be catastrophic (Bate, 2008), in terms of causing serious damage to the environment or to human lives, or non-recoverable material and financial losses (Aguiar et al., 2010). 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 noncritical qualities (Ampatzoglou, Gkortzis, Charalampidou and Avgeriou, 2013; Linares-Vásquez et al., 2014). 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 (Barney et al., 2012). 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 (Barney et al., 2012) 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 (Perry and Wolf, 1992). 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 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 (Runeson et al., 2012). We considered three critical QAs namely, correctness, performance and security, as well as six noncritical QAs namely, reusability, understandability, functionality, extendibility, effectiveness and flexibility, which are all interpreted as defined in the SQuaRE quality model (ISO/IEC, 2011). We selected these QAs as they are relevant to both practitioners and researchers, and there is a lack of studies investigating trade-offs between them. 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 chapter is organized as follows: related work is presented in Section 2.2, along with a discussion of the main contributions of this study. In Section 2.3, we present the design of the case study. In Sections 2.4 and 2.5 we present the results and discuss the most important findings respectively. In Section 2.6, we report on the identified threats to validity and actions taken to mitigate them. Finally, in Section 2.7 we conclude the chapter.

2.2 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., Bourque and Fairley (2014) and Kitchenham and Pfleeger (1996)); however, since our work is focused on the domain of CES, we decided to simply classify them as either critical or noncritical. We organize this section by first presenting studies on the domain of embedded systems (Section 2.2.1) and next on the evolution of software qualities in general (Section 2.2.2)[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 2.2.3). Most of the related work discussed in this section is based

[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 chapter

on a mapping study on software quality trade-offs by Barney et al. (2012).

2.2.1 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*** (Del Rosso, 2008). 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 *runtime 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. (2008) investigate the relationship between noncritical quality attributes, measured by metrics obtained from source code, and ***performance***, measured by physical metrics (i.e., *memory*, *time*, and *energy*) obtained from runtime monitoring. 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.

2.2.2 Quality Analysis through Evolution

Concerning the investigation of quality attributes through source code evolution; Buyens et al. (2009) 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 Components*, 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. (2014) characterize the evolution of ***one open source project*** (Apache Ant) in terms of ***size***, ***changeability***, ***cohesion***, and ***coupling*** QAs, through an exploratory study. 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, Penta et al. (2009) 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" (Penta et al., 2009). 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.

2.2.3 Overview of Related Work

The main differences of this study compared to the related work are summarized in Table 2.1. Specifically, we present the 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 nine QAs. To the best of our knowledge, this is the most inclusive study of this type in terms of investigated QAs.

Table 2.1: Overview of related work

#ref	App. Domain		QA		#cases	
	CES	Others	#critical	#noncritical	CES	Others
(Del Rosso, 2008)	✓	✗	1	1	4	N/A
(Oliveira et al., 2008)	✓	✗	1	5	4	N/A
(Buyens et al., 2009)	✗	✓	1	1	N/A	3
(Barros et al., 2014)	✗	✓	0	4	N/A	1
(Penta et al., 2009)	✗	✓	1	1	N/A	3
this	✓	✓	3	6	4	17

2.3 Case Study Design

This section describes the case study protocol, which was designed according to the guidelines of Runeson et al. (2012), and is reported based on the Linear Analytic Structure (Runeson et al., 2012).

2.3.1 Objectives and Research Questions

The goal of this study is described using the Goal-Question-Metrics (GQM) approach (van Solingen et al., 2002), 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 noncritical quality attributes?

RQ$_{1.2}$: Are there trade-offs between critical quality attributes?

RQ$_{1.3}$: Are there trade-offs between critical and noncritical 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 noncritical 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 noncritical 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? ($RQ_{1.4}$). Such information is important to investigate scenarios in which QAs must be balanced.

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 noncritical 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 ($RQ_{2.2}$).

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 2.3.4), (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 2.2. 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 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.

[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.

Table 2.2: 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		

2.3.3 Variables

In order to answer the research questions, we extracted three sets of variables from each unit of analysis (see Section 2.3.4). 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 noncritical (V6 – V11) QAs. We clarify that in this chapter 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 noncritical QAs respectively, are presented in detail in the following sections.

Assessment of Critical Quality Attributes

Bugs have been extensively investigated as indicators of quality. More specifically, Misra and Bhavsar (2003) have explored bugs as indicators for correctness, and Zaman et al. (2011) 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. (2011) 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 (Hovemeyer and Pugh, 2004). In this case study, we have chosen to use FindBugs because it provides:

- adequate performance (with respect to precision) when compared to similar tools (Hovemeyer and Pugh, 2004; Zheng et al., 2006);
- 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.

Assessment of Noncritical QAs

Regarding the quantification of noncritical quality attributes, we selected to use the Quality Model for Object-Oriented Design (QMOOD) (Bansiya and Davis, 2002). 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 (Bansiya and Davis, 2002). 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 (Bansiya and Davis, 2002). 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 (2002). Although the QMOOD quality model seems rather simplistic in its calculations, weighted sum is the most

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

classical and, therefore, used approach for combining metrics (Chatzigeorgiou and Stiakakis, 2013). Additionally, Bansiya and Davis (2002) 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 (Ampatzoglou, Michou and Stamelos, 2013), 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 by Ampatzoglou, Gkortzis, Charalampidou and Avgeriou (2013), Griffith and Izurieta (2014), and Alhusain et al. (2013).

2.3.4 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*, *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

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

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 (δ_{variable} = variable$_{\text{v}}$ - variable$_{\text{v-1}}$), 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 2.3.

Table 2.3: 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 (2002).	Percerons Client
V7	Difference between two versions, in the flexibility score: the flexibility assessment computed as defined by Bansiya and Davis (2002).	
V8	Difference between two versions, in the Understandability score: the understandability assessment computed as defined by Bansiya and Davis (2002).	
V9	Difference between two versions, in the Functionality score: the functionality assessment computed as defined by Bansiya and Davis (2002).	
V10	Difference between two versions, in the Extendibility score: the extendibility assessment computed as defined by Bansiya and Davis (2002).	
V11	Difference between two versions, in the Effectiveness score: the effectiveness assessment computed as defined by Bansiya and Davis (2002).	

2.3.5 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 2.1 depicts the analysis of qualities V6 vs. V10, for CES. Therefore, for every pair of QAs, we proceed as follows:

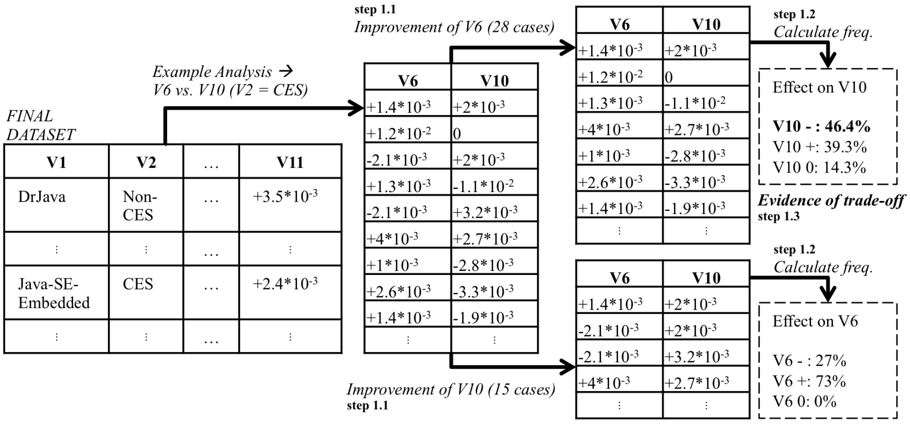


Figure 2.1: Example of trade-off analysis within the final dataset

(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 2.1, 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 2.1, 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 2.1, 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 2.1, 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. 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 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 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 2.4 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 2.4: Mapping of RQs to variables, steps, and presentation

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

2.4 Results

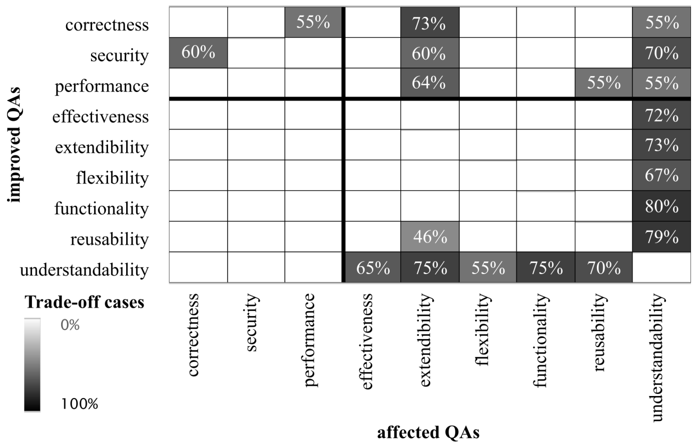
In this section we present the output of the analysis, and answer the research questions. To answer RQ_1 and its sub-questions, we explore the findings obtained from step 3. In order to visualize the interaction between the QAs, we compiled raw data[5]into a heat map (see Figure 2.2). In the heat map of Figure 2.2, 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 noncritical groups of QAs. Hence, the top-left quadrant represents the interactions between critical QAs, the bottom-right quadrant represents the interactions between noncritical QAs, while the other two represent the interaction between QAs of the two groups.

Based on Figure 2.2, we are able to answer all sub-questions of RQ_1, by confirming the existence of trade-offs between QAs (see Section 2.3.5, 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), noncritical 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 noncritical QAs (and vice-versa);
- *correctness* and *performance*, as well as between *security* and *correctness*;

[5] The raw data and other supplementary material on the collected data during the study is available at: `https://doi.org/10.5281/zenodo.2350387`

[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.



	correctness	security	performance	effectiveness	extendibility	flexibility	functionality	reusability	understandability
correctness			55%		73%				55%
security	60%				60%				70%
performance					64%			55%	55%
effectiveness									72%
extendibility									73%
flexibility									67%
functionality									80%
reusability					46%				79%
understandability				65%	75%	55%	75%	70%	

Figure 2.2: Trade-offs in CES domain

- 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 2.2 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 noncritical QAs: although the improvement of some critical QAs negatively affects noncritical QAs, the opposite phenomenon never appears, i.e., in this study we found no evidence of noncritical 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 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 2.2 (see Figure 2.3), but considering data from the non-CES projects, and another representing the difference of the two heat maps (see Figure 2.4). In Figure 2.4, three different filling patterns (with their respective colors) represent the possible classifications for the observed trade-offs:

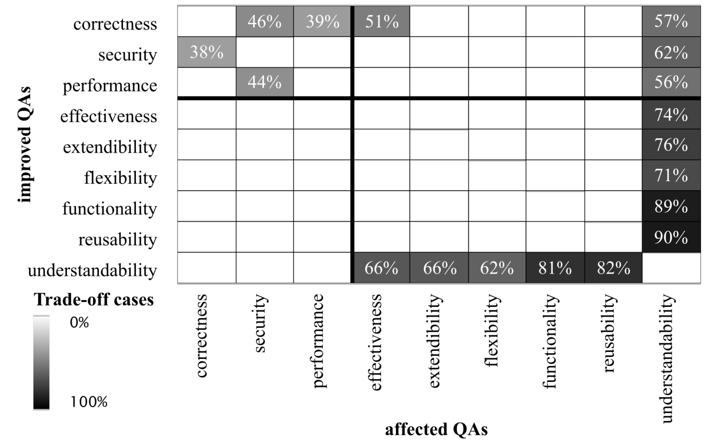
evident only in the group of CES projects (red background with circles); evident only in the group of non-CES projects (blue with slanted lines); and evident in both groups (green background with vertical lines). Based on this figure, we answer affirmatively RQ$_{2.1}$ - RQ$_{2.2}$, and, additionally, 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 noncritical 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.

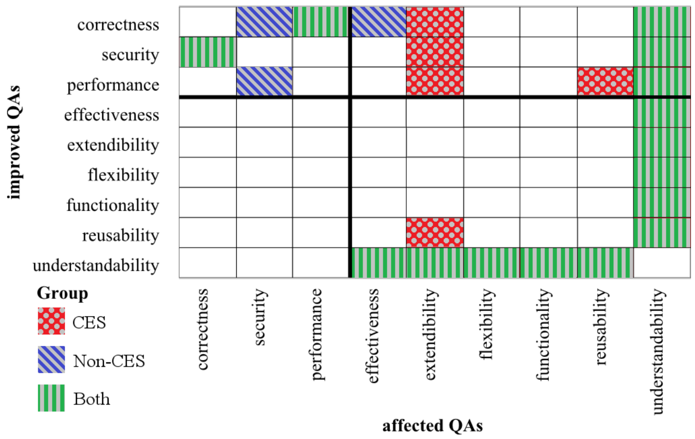
- *Trade-offs occurred only in the group of non-CES projects:* between correctness and security, as well as between performance and security; and between correctness and effectiveness.

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



	correctness	security	performance	effectiveness	extendibility	flexibility	functionality	reusability	understandability
correctness		46%	39%	51%					57%
security	38%								62%
performance		44%							56%
effectiveness									74%
extendibility									76%
flexibility									71%
functionality									89%
reusability									90%
understandability				66%	66%	62%	81%	82%	

Figure 2.3: Trade-offs in non-CES domain



improved QAs \ affected QAs	correctness	security	performance	effectiveness	extendibility	flexibility	functionality	reusability	understandability
correctness		▨	🟩	▨	🟥				🟩
security	🟩				🟥				🟩
performance		▨			🟥			🟥	🟩
effectiveness									🟩
extendibility									🟩
flexibility									🟩
functionality									🟩
reusability					🟥				🟩
understandability				🟩	🟩	🟩	🟩	🟩	

Figure 2.4: Comparison between CES and non-CES groups

2.5 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.

2.5.1 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 Bansiya and Davis (2002), 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. (2008), who suggest that cohesion and *coupling* metrics, that are assessments of reusability, are compromised in favor of metrics for *performance* (Bansiya and Davis, 2002).

Although the results of Figure 2.2 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 (Alshammari et al., 2010), 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 by Oliveira et al. (2008), 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.

2.5.2 Comparison of the Two Groups

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

- *noncritical QAs do not affect critical QAs*. Improvements on object-oriented properties (i.e., enhancement in design-time QAs – all noncritical 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 noncritical 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 noncritical one.

- *security negatively affects correctness*. Fixing security vulnerabilities can lead to additional errors in the code. For example, in order to fix a vulnerability

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

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 noncritical QAs might reflect a higher importance of critical QAs over noncritical 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 (Bansiya and Davis, 2002). 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 2.4 suggest that *understandability is negatively affected by all other QAs and vice-versa*, we believe that this result needs further

[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

investigation. Specifically, we observed that understandability is a QA that continually deteriorates during systems evolution (Penta et al., 2009), because based on the way it is calculated (Bansiya and Davis, 2002) 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 noncritical QAs, again because of the way that both understandability and noncritical-QAs are calculated. Concluding, we believe that the decrease of understandability in our study is not the result of explicit trade-offs, but simply, the natural effect of system growth.

2.5.3 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.

2.6 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 (2002), for assessing the noncritical 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 2.3.3). Another threat is that effect size is not considered during the data analysis (Section 2.3.5), 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 (Bansiya and Davis, 2002), for estimating noncritical 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 noncritical QAs. Therefore, all discussions on the existence of possible trade-offs between critical and noncritical 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 (Bass et al., 2008).

2.7 Conclusions

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 noncritical 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 noncritical QAs, for either CES or other application domains. Finally, we provide evidence on the fact that noncritical QAs are more often compromised than critical QAs.

This chapter focused on characterizing the domain of CES with respect to QAs trade-offs, obtaining knowledge on the *state-of-the-practice*. This study is the first step into exploring the problem space of the PhD project. To complete the problem exploration, the next chapter reports on the *state-of-the-art* on how CES are designed.

Based on:

Daniel Feitosa, Apostolos Ampatzoglou, Paris Avgeriou, Frank J. Affonso, Hugo Andrade, Katia R. Felizardo and Elisa Y. Nakagawa, (2017) *"Design Approaches for Critical Embedded Systems: A Systematic Mapping Study,"* in Evaluation of Novel Approaches to Software Engineering, Springer International Publishing, pp. 243–274, DOI:10.1007/978-3-319-94135-6_12

Chapter 3

Design Approaches for Critical Embedded System: A Systematic Mapping Study

Abstract

In the last years the amount of software accommodated within CES has considerably changed. This change means that software design for these systems is also bounded to hard constraints (e.g., high security and performance). Along the evolution of CES, the approaches for designing them are also changing rapidly, so as to fit the specialized needs of CES. Thus, a broad understanding of such approaches is missing. Therefore, this chapter aims at establishing a fair overview on CESs design approaches. For that, we conducted a Systematic Mapping Study (SMS), in which we collected 1,673 papers from five digital libraries, filtered 269 primary studies, and analyzed five facets: design approaches, applications domains, critical quality attributes, tools, and type of evidence. Our findings show that the body of knowledge is vast and overlaps with other types of systems (e.g., real-time or cyber-physical systems). In addition, we have observed that some critical quality attributes are common among various application domains, as well as approaches and tools are oftentimes generic to CES.

3.1 Introduction

Engineering CES is particularly challenging, since it needs to guarantee the satisfaction of various critical qualities. One of the key solutions to alleviate this challenge is to design a sound architecture and validate it against the critical quality attributes. To this end, multiple approaches have been proposed, solving a variety of specific design problems. However, the plethora and diversity of available solutions has led to a difficulty on understanding, applying or even extending and combining such approaches. Thus, in order to support researchers and practitioners on CES design, it is important to have a comprehensive understanding of this field.

To contribute towards a better understanding of design approaches for CES, we have conducted a systematic mapping study; this is a commonly used approach for assessing and describing the state of the art in a specific domain or problem (see Section 3.3 for more details). The contributions of this study are the following: (a) a classification of the existing approaches to design CES; (b) a list of tools for supporting existing approaches; (c) a list of domains for which approaches have been developed and used; (d) a list of the most commonly identified CQAs in the CES design; and (e) a classification of these approaches, based on the level of their empirical evidence.

The remainder of this chapter is organized as follows: related work is presented in Section 3.2, along with a discussion of the main contributions of this study. In Section 3.3, we present the design of the systematic mapping study. In Sections 3.4 and 3.5 we present the results and discuss the most important findings respectively. In Section 3.6, we report on the identified threats to validity and actions taken to mitigate them. Finally, in Section 3.7 we conclude the chapter.

3.2 Related Work

This section describes related Systematic Literature Reviews (SLRs) or Systematic Mapping Studies (SMSs), also known as secondary studies. To the best of our knowledge, there are no studies that focus on exactly the same topic as ours, i.e., designing of CESs. Thus, we searched for related work such as SMSs and SLRs that cover the entire software development process of CES, or a specific phase.

3.2.1 Development Processes

We identified two studies that discuss software development processes and are related to CESs (Cawley et al., 2010; Eklund and Bosch, 2013). Although such processes do not focus or limit themselves to the design phase, they do have impact on the design phase. Cawley et al. (2010) investigated Lean/Agile development processes on safety-critical systems, focusing on medical devices. For this purpose, an SLR based on the guidelines of Kitchenham and Charters (2007) was performed. The results of the SLR suggest that Lean/Agile methodologies are appropriate for the development of safety-critical systems, as they support several practices for regulated safety-critical domains (e.g., traceability and testing). However, the results also suggest a lack of adoption of Lean/Agile methods in these domains. This is not surprising as regulated environments typically involve activities that are not commonly used in these processes. Eklund and Bosch (2013) investigated a holistic model for aligning software development processes with the architecture of embedded software. As part of this study, an SMS on development approaches for embedded systems was performed (based on the guidelines of Kitchenham and Charters (2007)). The results of the study suggest that there is no single most common approach (or set of approaches) but, approaches are tailored for specific domains or products and may have different characteristics (e.g., incorporating agile practices). Despite the high customization of processes, the authors have been able to identify some similarities, e.g. activities are often executed sequentially and follow a V-model- (Karlström and Runeson, 2006) or stage-gate-like (Selim et al., 2012) process. In addition, the architectures created from these processes are often focused on supporting specific quality attributes, which are typically domain-specific (e.g., dependability for the space domain). Based on the identified approaches, the authors derived five archetypical developments processes, with their respective characteristics, aiming to support selection or migration between concrete archetypal development approaches.

3.2.2 Verification and Validation

Not all activities in the verification and validation of critical embedded software (V&V) are related to its design. However, a significant part concerns the verification and validation of design and are, therefore, relevant to the design phase. We identified two secondary studies that discuss aspects of V&V and are related to CES (Barbosa et al., 2011), (Elberzhager et al., 2013). Barbosa et al. (2011) investigated software testing of CESs, checking the compliance level with the standard DO-178B, for the aviation industry. The aim was to identify primary studies that could be used to create a methodology for testing of CES. For this purpose, a SLR, based on Dybå and Dingsøyr (2008), was performed to identify studies that implemented or applied V&V techniques in the context of CES. The results suggest that four techniques (functional, structural, mutation and model-based testing) are widely applied for testing of CES, from which the most recurrent technique is functional testing. In addition, all testing requirements of DO-178B have been investigated, with "structural coverage analysis" (e.g., dead code and deactivated code) being the most addressed requirement, likely due to its inherent complexity. Elberzhager et al. (2013) investigated quality assurance techniques (i.e., analysis or test approaches) applied to Matlab Simulink models. These models are used in embedded software design, especially in critical domains. The aim was to develop an approach able to integrate different quality assurance techniques. For this purpose, an SMS was performed based on the guidelines of Petersen et al. (2008), which presented different analysis and test techniques as well as some combined approaches. The results of the study suggest that formal methods, properties checking (e.g., rule-based analysis) and automatic test generation are the most common approaches for performing quality assurance for embedded systems. The results also suggest a lack of research on combining analysis techniques with testing techniques for such models.

3.2.3 Software Architecture

The activity of architecture design for embedded systems was investigated by Antonio et al. (2012), which aimed at establishing the state of the art on the topic by analyzing proposed architectures, available on the literature. For that, a SMS based on the guidelines of Petersen et al. (2008) was performed. To understand the activity, various characteristics were collected from the architectures, and used for classifying them. Firstly, the architectures were grouped according to the type of modeling technique used to design them, namely formal, semi-formal and informal. Next, further classes were identified based on recurrent characteristics, e.g., level of abstraction and whether it is domain-specific. The results of the SMS suggest that the Architecture Analysis and Design Language (AADL) is the most used formal modeling approach, whereas UML stands out among the semi-formal and informal approaches. In addition, the most recurrent characteristic of these architectures is that they are designed to specific application domains.

Similar to the previous study, Guessi et al. (2012) investigate the modeling of software architectures for embedded systems. However, this study focuses on architecture description languages (ADLs), as well as the concerns (e.g., quality attributes) being addressed and information (e.g., components, events) being represented in the designed architectures. The investigation was performed via a SLR based on the guidelines of Kitchenham and Charters (2007). The results suggest that UML is the most common language, while safety is the concern that is more often addressed. Despite the variety of approaches that currently exist, the results also suggest that more attention should be placed on the description of embedded system architectures. Among the reasons, Guessi et al. (2012) argue that there is a lack of consensus about the most adequate approach(es) for describing architecture, as well as whether existing approaches are sufficient for representing the variety of embedded systems.

Nakagawa et al. (2013) present the state of the art on architecting approaches for systems of systems[1] (SoS), of which CES are among the most common examples. For that, an SLR based on the guidelines of Kitchenham and Charters (2007) was performed, investigating the creation, representation, evaluation and evolution

[1] SoS are integrated solutions comprising operationally independent (nontrivial) systems, which are orchestrated in order to provide a more complex functionality.

of these architectures. The results suggest the existence of several approaches, although most of them lack maturity and are neither adequately adapted nor widely adopted. In addition, several application domains (e.g., avionics and space) and quality attributes (e.g., security, reliability and performance) are common between SoS and CES.

3.2.4 Comparative Analysis

After presenting related work, it is important to highlight the differences between these studies and our work. To illustrate these differences, we compare them w.r.t. six characteristics (Table 3.1): review type; number of included primary studies; whether the study focuses on CES or is only indirectly related (i.e., with partial applicability to CES); whether it considered quality attributes (QA) in the investigation; whether it considered application domains in the investigation; and the main topic of the investigation. The review type is an indication of whether the study presents an overview or a detailed analysis over the main topic (SMS) or it examines more in-depth research questions (SLR). As presented in Table 3.1, three other SMSs were performed, although they were focused in different, yet related, topics. However, these three studies were not focused on CESs, which reinforces the purpose of our study, as it complements existing knowledge. Other important aspects of our study include the larger body of knowledge that has been investigated (due to the broader topic of research), as well as the consideration of quality attributes and application domains in the investigation. CESs are used in a variety of application domains and multiple factors affect the decision-making to select or reuse a design approach. Quality constraints are among the most relevant factors, as also suggested by related work (Eklund and Bosch, 2013; Guessi et al., 2012; Nakagawa et al., 2013). Application domains may also play an important role, as each domain groups a set of common requirements, that are in turn related to specific quality attributes (Eklund and Bosch, 2013).

3.3 Review Methodology

Systematic Mapping Studies (SMSs) and Systematic Literature Reviews (SLRs) have been broadly adopted as systematic research methods to aggregate knowledge. As this study aims to outline the state-of-the-art on design approaches for CES in a broad sense, we decided to perform an SMS (Petersen et al., 2008). The rest of this section describes the protocol of our SMS, based on the guidelines of Petersen et al. (2008).

Table 3.1: Comparison between related work and our study

Study	**Review Type**	**Number of studies**	**Focus on CES?**	**Investigated QAs?**	**Focus on Domains**	**Main topic**
(Cawley et al., 2010)	SLR	19	Yes	No	No	DV
(Eklund and Bosch, 2013)	SMS	23	No	Yes	Yes	DV
(Barbosa et al., 2011)	SLR	97	Yes	No	No	V&V
(Elberzhager et al., 2013)	SMS	44	No	No	No	V&V
(Antonio et al., 2012)	SMS	104	No	No	No	SA
(Guessi et al., 2012)	SLR	24	No	Yes	No	SA
(Nakagawa et al., 2013)	SLR	60	No	Yes	Yes	SA
Ours	SMS	258	Yes	Yes	Yes	design

DP = development process
V&V = verification and validation
SA = software architecture

3.3.1 Research Scope

The goal of this SMS is described using the Goal-Question-Metrics (GQM) approach (van Solingen et al., 2002), as follows: "**analyze** existing software engineering literature **for the purpose of** characterizing the state of the art **with respect to** approaches (e.g., processes, methods and tools) for designing critical embedded systems **from the point of view of** researchers and practitioners **in the context of** software-intensive systems engineering". Based on the goal we defined the following research questions (RQs):

RQ_1: What are the proposed approaches for designing CES?

$\text{RQ}_{1.1}$: Is the nature of these approaches industrial, academic or mixed?

$\text{RQ}_{1.2}$: What is the purpose of the approach?

RQ_2: What are the application domains where these approaches are applied?

RQ_3: What are the most common critical quality attributes identified in CES design?

RQ_4: What tools have been used to support CES design?

RQ_5: What are the types of evidence provided in CES design research?

To achieve the aforementioned goal, we must analyze and present the existing body of knowledge from different perspectives. The most important outcome of this SMS is the identification and characterization of the approaches that were created and/or used to design CES (RQ_1). As a first step in characterizing the approaches, we consider their nature and purpose. Next, we look at the application domain

(RQ_2) which influences CES design as it often imposes a number of constraints. For example, several application domains are bounded by international standards (e.g., DO-178B for aviation). In addition, these constraints commonly aim at defining critical quality values (e.g., safety); thus, design approaches are often targeting those values (e.g., fault tree analysis). Therefore, investigating the addressed quality attributes (RQ_3) is of paramount importance. Furthermore, multiple tools have been proposed or tailored to support the design of CES. As the number of CES grows, it is interesting to investigate how this reflects on the tooling (RQ_4), e.g., leading to news tools and adaptation of existing ones. Finally, it is important to not only classify the approaches, but also assess their maturity level to inform researchers and practitioners. For that, we analyze the types of evidence provided within the literature (RQ_5).

3.3.2 Search Strategy

Considering the research questions, we defined the search strategy, which comprises the selection of sources for collecting primary studies, as well as the definition of the scope for the collection.

Sources Selection

We decided to perform an automated search, as a manual search would be very time-consuming, thus not allowing us to search as many venues. In addition, by considering digital libraries (through an automated search) we might also include venues that otherwise we would not be aware of. The following criteria were adopted to select search sources (i.e., digital libraries): content update (publications are regularly updated); availability (full text of the papers is available); quality of results (accuracy of the results returned by the search); and versatility export (since a lot of information is returned through the search, a mechanism to export the results is required). These criteria are also discussed by Dieste et al. (2009). The selected sources for our SMS are: ACM Digital Library, IEEE Xplore, Science Direct, Springer Link and Scopus. According to Dybå et al. (2007), the first four digital libraries are sufficient to conduct SMSs in the context of software engineering. Furthermore, Scopus was added, since it is considered to be the largest database of abstracts and citations (Kitchenham and Charters, 2007).

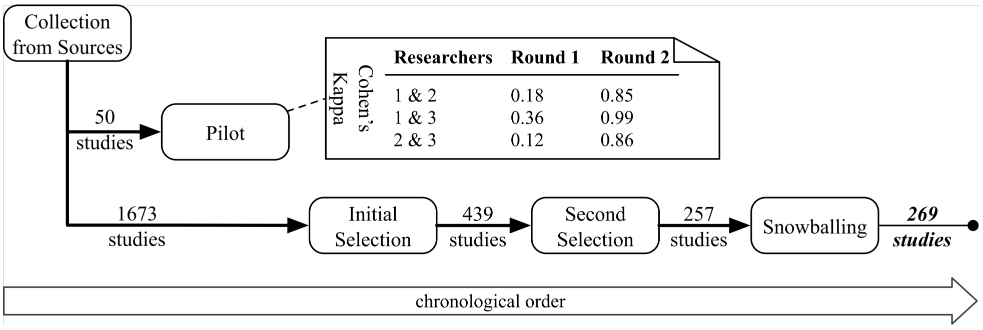
Search Scope

As CESs have been the subject of research for a long time, we decided to not limit the start of the search period based on date of publication. However, we limit the end date of the search period in order to measure influence of the primary studies (see Section 3.3.5), considering primary studies published up to two years before the date of collection. We performed the data collection on March of 2015 and, thus, collected primary studies published up to March of 2013. Moreover, only primary studies written in English will be processed in this SMS. Due to automated search, we also defined a search string for filtering the studies to those that can be potentially included in the SMS. As we are interested in approaches for CES design, we selected two main keywords, "Critical Embedded System" and "Approach", with the respective related terms. The keywords were chosen to be simple enough to yield a large number of results and, at the same time, rigorous to cover only the desired research topic. The final search string is: ("Critical Embedded System" OR "Critical Embedded Systems" OR "Critical Embedded Software") AND ("Approach" OR "Approaches" OR "Method" OR "Methods" OR "Framework" OR "Frameworks" OR "Technique" OR "Techniques" OR "Process" OR "Processes" OR "Tool" OR "Tooling" OR "Guideline" OR "Guidelines").

We clarify that we do not include terms such as "real-time", "hard real-time" or "cyber-physical systems", as they describe a broader range of systems, which extrapolates the scope of this SMS, and would make the paper selection process impractical. To validate the search string and, consequently, the papers collected by the automated search, we performed a manual search in a small number of venues, similarly to determining a quasi-gold standard as proposed by Zhang and Babar (2010). We selected the venues for the manual search based on their likelihood to publish studies on CES design: Real-time Systems journal, Digital Avionics Systems Conference (DASC), and International Conference on Computer Safety, Reliability, and Security (SAFECOMP). To filter the primary studies for the quasi-gold standard, we considered the metadata (i.e., title, keywords and abstract) and full text (when necessary), resulting in the collection of 23 primary studies. Based on the quasi-gold standard, we adapted the search string to ensure that all 23 primary studies were included.

3.3.3 Study Selection

Based on the previously mentioned search strategy, we defined the procedure for filtering the results of the automated search, selecting the primary studies to be analyzed in the SMS. The study selection comprises the definition of the criteria for filtering the papers, both inclusion and exclusion criteria, as well as steps for applying them. We include a primary study if it: (a) proposes an approach to design CESs; (b) reports on the use of an approach to design CESs; (c) evaluates an approach to design CESs; or (d) discusses approach(es) to design CESs. A primary study is excluded if it is an editorial, position paper, keynote, opinion paper, tutorial, poster or panel. To promote a common understanding of the selection criteria among the three involved researchers, we performed a pilot selection on a small subset (50) of the papers collected from the sources. In this pilot, during a first review round, all researchers analyzed title, keywords and abstract of all papers and Cohen's Kappa was calculated between every pair of researchers (see Figure 3.1). We clarify that no previous discussion was performed in order to evaluate the inclusion and exclusion criteria. Next, all researchers and authors discussed the criteria and their interpretation. Main points of this discussion included the boundaries of the design phase, hardware design and the inclusion of papers that do not propose approaches (e.g., use or discussion). Finally, in a second review round, the papers are analyzed again, but this time also considering introduction and conclusion sections (if necessary), and a new calculation of Cohen's Kappa was performed (see Figure 3.1).



Researchers	Round 1	Round 2
1 & 2	0.18	0.85
1 & 3	0.36	0.99
2 & 3	0.12	0.86

Figure 3.1: Study selection

To select the primary studies, we defined a three-step procedure. In every step, the papers were divided into three sets and three researchers were responsible for reviewing the papers of two sets. By doing this, we guarantee that every paper was reviewed by two different people while avoiding all three having to read all papers. When an inclusion/exclusion decision was conflicting or dubious (e.g., one or both reviewers were not confident), the case was discussed among all authors. The selection steps were the following: (1) Initial selection: the search string was customized and applied to each publication source listed in Section 3.3.2. The string terms were searched in the title, abstract and keywords of all primary studies available in each database and search engine. As a result, a set of primary studies possibly related to the research topic was obtained. Based on this set, the title and the abstract of each primary study were read and evaluated based on the inclusion and exclusion criteria. The introduction and the conclusion may also be considered when necessary; (2) Second selection: each of the previously selected primary studies were read in full-text and analyzed according to inclusion and exclusion criteria. This step also included the data extraction, which is discussed in Section 3.3.5; and (3) Snowballing: the references of the studies selected in step 2 were used to identify extra literature, for which steps 1 and 2 are repeated.

3.3.4 Keywording

During the first two steps of the selection procedure (see Section 3.3.3), a set of keywords was collected from each primary study. As proposed by Petersen et al. (2008), the keywording process occurs in two steps:

Identification of context: While reading the paper, the reviewer identifies any keywords and concepts that are relevant to describe that particular study. For example, words that describe the purpose of the approach, code of standards and names of quality attributes or tools were collected. During this step, reviewers share topics of keywords (e.g., code of standards) to maintain consistency and optimize the collection. Differently from Petersen et al. (2008), we extended the searching of keywords to the whole paper, as some relevant keywords have been identified within the full text at early stages of the study.

Summarization: The keywords are combined in order to create abstractions that support understanding the body of knowledge under investigation. Examples of such abstractions are the topics mentioned in the previous step (e.g., standards). The abstractions also support identifying categories and create a classification scheme for the primary studies.

We applied keywording not only to classify the primary studies but also to identify relevant concepts for all research questions, e.g., purpose of tools, application domains standards and safety integrity levels (SILs). As a result, we collected 164 keywords, which were used to create a classification scheme (see Section 3.4.2).

3.3.5 Data Extraction and Mapping

During the second selection procedure (see Section 3.3.3), a set of variables were collected from each primary study to answer the research questions. Similar to selection procedure, the data collection of every paper involved two researchers and conflicts were discussed among all authors. The extracted variables are described in Table 3.2.

The mapping between variables and research questions is provided in Table 3.3, accompanied by the analysis method used on the data. The type of evidence (V14) evaluates the level of evidence of the proposed approach. For that, we adopted the classification proposed by Alves et al. (2010) in order to make the assessment more practical. From weakest to strongest, the classes are: (i) no evidence; (ii) evidence obtained from demonstration or working out toy examples; (iii) evidence obtained from expert opinions or observations; (iv) evidence obtained from academic studies (e.g., controlled lab experiments); (v) evidence obtained from industrial studies (i.e., studies are done in industrial environments, e.g., causal case studies); and (vi) evidence obtained from industrial application (i.e., actual use in industry).

Table 3.2: Extracted variables

Variable	Description
V1	Author(s)
V2	Year
V3	Title
V4	Source
V5	Venue
V6	Author(s) keywords
V7	Number of citations per year
V8	Type of paper (conference / journal / book)
V9	SMS keywords
V10	Approaches to design CES
V11	Application domain(s)
V12	Critical quality attributes
V13	Nature of the approaches (industrial/academic/mixed)
V14	Tools to support the approaches
V15	Type of evidence used to develop the approach

3.4 Results

In this section, we present the results of the mapping study, highlighting the most important observations. We note that the complete information from data extraction is publicly available as part of the supplementary material for this chapter (Feitosa, Ampatzoglou, Avgeriou, Affonso, Andrade, Felizardo and Nakagawa, 2017). We clarify that, when necessary, we cite specific primary studies using an "S" (e.g., [S134]). The list the primary studies can be found in Appendix A, and we have made it also available as part of the supplementary material for this study (Feitosa, Ampatzoglou, Avgeriou, Affonso, Andrade, Felizardo and Nakagawa, 2017).

Table 3.3: Mapping of variables to RQs

Research Question	**Used Variables**	**Analysis Method**
RQ$_1$ (Approaches)	V1-V3, V6, V7, V9-V10	Descriptive Statistics (sum, average, frequency analyses, etc.). Classification based on keywording. Heatmap based on classification and year. Crosstabs on classification vs. nature.
RQ$_2$ (Application domains)	V1-V3, V10, V11	Descriptive Statistics (sum, average, frequency analyses, etc.). Heatmap based on application domain and year. Crosstabs on application domain vs. approaches (classification).
RQ$_3$ (Critical quality attributes)	V1-V3, V9-V12	Descriptive Statistics (sum, average, frequency analyses, etc.). Heatmap based on critical quality attribute and year. Bubble chart on critical quality attribute vs. approaches (classification) vs. application domain. Spearman correlation between critical quality attribute and approaches (classification), and application domain.
RQ$_4$ (Tools)	V1-V3, V9, V10, V14	Descriptive Statistics (sum, average, frequency analyses, etc.). Classification based on keywording.
RQ$_5$ (Evidence type)	V1-V3, V9, V10, V15	Descriptive Statistics (sum, average, frequency analyses, etc.). Heatmap based on type of evidence and year. Bubble chart on type of evidence vs. approaches (classification) vs. application domain. Spearman correlation between type of evidence and approaches (classification), and application domain.

3.4.1 Demographic Overview

The distribution of studies, per year, among the different types of publication (conference, journal and book) is depicted in Figure 3.2. We clarify that we collected studies published up to March of 2013 (see Section 3.3.2), resulting on the observed smaller number in that year. We notice a linear growth in the number of conference papers. The number of journal articles experiences a growth as well, but not as high. We note that conference proceedings published as books were counted as conferences, explaining the small number of book chapters in the chart.

To investigate further potential reasons for the aforementioned growth, we looked at the venues and checked whether they focus on CES alone, or have a broader scope (e.g., embedded systems) and only include CES as one of the topics of interest. We observed that, although a few venues do focus on CES (e.g., Brazilian symposium on CES), most of the studies were published in other venues, suggesting a shift or growing interest of the respective (broad) communities towards CES. In addition, we can try to identify the most relevant venues, by looking at their distribution according to two metrics: number of included studies (Figure 3.3a), and number of citations (Figure 3.3b). We chose these metrics, because they reflect distinct features that may draw the attention of researchers to venues: the size of the CES community within the venue, and the potential visibility of the study. To investigate the venues, we analyzed how they are distributed statistically, identifying the high outliers, which in this case indicate popular venues for CES. We used the software IBM SPSS Statistics to create the box-plots as well as to identify the outliers, using the stem-and-leaf diagram.

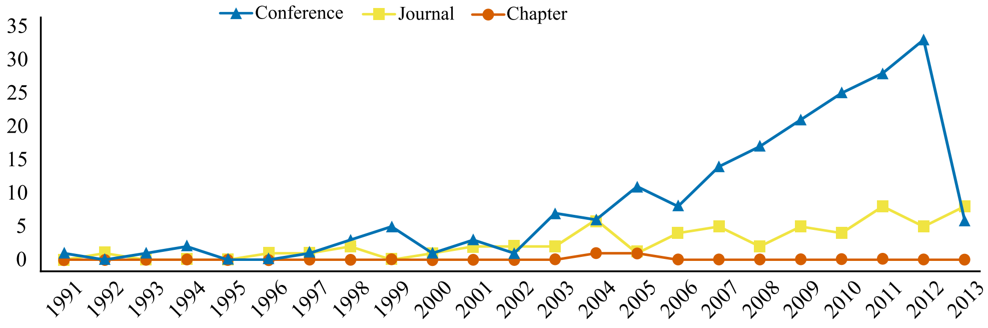


Figure 3.2: Number of filtered studies per year, per type of paper

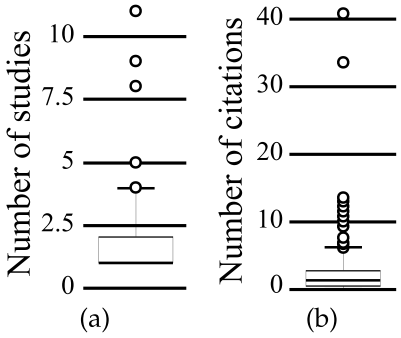


Figure 3.3: Box-plot of venues based on (a) number of studies and (b) citations per paper per year

On the one hand, Figure 3.3a shows that the vast majority of venues contributed with one or two papers only, respectively 111 (approx. 70%) and 28 (17.5%). The analysis suggests that venues that contributed with four papers or more (nine venues) are exceptional in our dataset. On the other hand, Figure 3.3b shows that most venues (85%) exhibit a maximum average of four citations per paper per year.

The analysis of this metric suggests that venues with an average citation rate of 6.2 or more (15 venues in total) are also exceptional. Thus, we identified a set of 22 exceptional venues, which is presented in the supplementary material (Feitosa, Ampatzoglou, Avgeriou, Affonso, Andrade, Felizardo and Nakagawa, 2017).

3.4.2 Design Approaches

As shown in the previous section we were able to collect a large number of studies. Therefore, it is infeasible to present all collected approaches here. For that reason, we decided to present the results as a summary based on the types of approaches that were found, which are based on a classification scheme (presented below). In addition, we present some details on the most relevant approaches, i.e., those with the most citations, identified by using the number of citations according to Google Scholar. To avoid omitting relatively new papers (i.e. those that did not have enough time to receive citations), we considered the number of citations per year. In the next subsections, we elaborate on this classification scheme and results.

Classification Scheme

The design phase in a development lifecycle is often elusive, in the sense that it is typically hard to determine the boundaries of design with respect to the other lifecycle phases. In embedded systems development, including systems with harder constraints such as CES, this is no exception. However, in order to classify the design approaches, it is necessary to identify the parts of the development lifecycle that approaches belong to, i.e., their purpose. It is widely accepted that the design phase includes activities that translate requirements into software/hardware elements, with their respective responsibilities, excluding the actual implementation of these elements (source code) (Bass et al., 2012; Marwedel, 2010; Sommerville, 2000). To initialize our classification scheme, we collected the keywords obtained from the keywording process (see Section 3.3.4) and filtered those that regard the purpose of approaches. Next, we grouped the keywords by similarity, trying to organize them in a hierarchical fashion, also creating a generic design flow[2]. However, it was not possible to derive such hierarchical organization, as we were not able to identify or define a flow that was sufficiently generic to accommodate the extracted approaches. This is due to the high heterogeneity of domains, requirements, and platforms for which CES are designed (Marwedel, 2010). Therefore, we decided to organize our keywords based on a simplified design flow proposed by Marwedel (2010), which is meant to generically represent the design activities of an ES.

[2] A design flow is the sequence of specific activities (with respective approaches) to design a system.

To create our classification scheme, we successfully mapped the identified keywords into some elements of the design flow proposed by Marwedel (2010), and assessed whether or not the relationship between the keywords were consistent with the description of the simplified design flow. By the end of the keyword mapping, we were able to derive five types of activity representing general purposes, as well as scope them and their relationships. The final classification scheme is presented in Figure 3.4, in which rectangles represent each general purpose, and arrows show the flow of design artifacts. Moreover, smaller rectangles (i.e., Optimization and Test) represent auxiliary purposes that are special for the design of embedded systems. The approaches are grouped according to how they modify the system's design, rather than based on a logical sequence of activities. In addition, common activities in embedded system design are also clearly placed within the classification (e.g., scheduling is placed within Application mapping). The main characteristic of this kind of classification is that it is artifact-centric, i.e., the artifacts dictate what activities may be performed (i.e., what purposes they serve), rather than the other way around (Marwedel, 2010). The five general purposes are described as follows:

Specification: these activities formalize constraints (e.g., safety requirements) in the design. They define the scope/boundaries of the design. To draw a parallel, this type of activity is similar to the analysis in a software architecture design flow (Hofmeister et al., 2007). Common examples are formal specification languages, such as Z.

Application mapping: these activities generate new (partial) design information. A series of mappings are applied in order to refine the design from a more abstract representation to platform-specific design. In a software architecture design flow, this type of activity is similar to architecture synthesis (Hofmeister et al., 2007). Common approaches encompass: mapping of operations to concurrent tasks; mapping of operations to HW/SW; compilation; or scheduling

Evaluation & Validation: similarly to the evaluation in a software architecture design flow (Hofmeister et al., 2007), these activities evaluate design elements w.r.t. the objectives (e.g. provide a proper scheduling of tasks) and validate a design description against other descriptions. Examples of approaches are algorithms or analysis frameworks for comparing models that tackle different quality attributes, as well as simulations.

Optimization: these activities perform design tuning according to stated objectives. Examples of approaches are HL transformation and energy optimizations.

Test: these activities include test generation and testability evaluation. They are included in design iterations if testability issues are already considered during the design steps. Tests are run after the design phase.

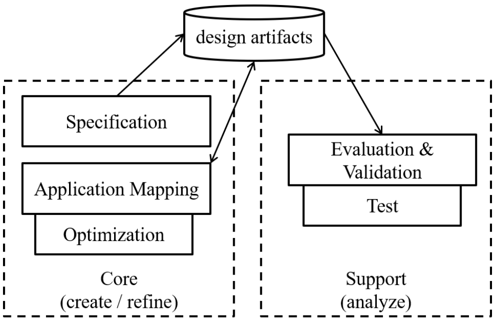


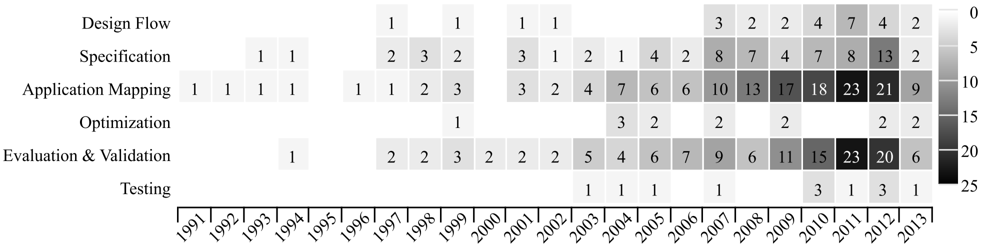
Figure 3.4: Classification scheme

This classification is sufficiently robust for expressing different software, hardware and SW/HW design flows, including prominent ones such as the V-Model (Bartelt et al., 2010) and the design flow provided with SpecC (Gajski et al., 2000). Finally, it is important to clarify that approaches may serve several purposes. For example, some architecture modeling languages are able to perform both application mapping and specification.

Summary of Design Approaches

To analyze the extracted approaches, we classified each of them into one or more of the aforementioned general purposes. In addition, some studies presented entire design flows and, therefore, we also considered it as a category for the classification. Figure 3.5 depicts a heat map that shows the number of studies, per year, discussing approaches from each category.

In this heat map, darker shades of grey represent bigger numbers, which are presented as well. For example, in 2011, 23 studies that contain approaches for application mapping were published. One can notice that most attention has been given to approaches for Application Mapping and Evaluation & Validation, which is understandable because approaches that serve this purpose encompass most of the design flow of an embedded system. Approaches for Specification of CES design were also presented in a considerable number of studies. Such interest is explained by the necessity of unambiguously representing the different aspects of CES (e.g., safety, components, security) in a variety of platforms (e.g., time/event-triggered and mixed architectures, and communication protocols). Table 3.4 presents the number of studies



	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013
Design Flow							1		1		1	1					3	2	2	4	7	4	2
Specification			1	1			2	3	2		3	1	2	1	4	2	8	7	4	7	8	13	2
Application Mapping	1	1	1	1		1	1	2	3		3	2	4	7	6	6	10	13	17	18	23	21	9
Optimization									1					3	2		2		2			2	2
Evaluation & Validation				1			2	2	3	2	2	2	5	4	6	7	9	6	11	15	23	20	6
Testing													1	1	1		1			3	1	3	1

Figure 3.5: Number of studies, per year, containing approaches from each category

in each category, grouped by nature (i.e., academic, industrial or mixed). The table also presents the number of citations per year, for the entire set of studies. By exploring this table, one can notice that most of the studies were performed in an academic setting, followed by mixed and industrial settings, respectively; this is understandable as the included venues are more academic than industrial. In addition, solutions are normally proposed and explored in academic studies before they are applied in industry. However, there is one interesting observation to highlight. The mixed setting does not follow the same trend of the academic and industrial settings (which are in accordance to Figure 3.5): studies performed in collaboration between academia and industry were mostly focused on Evaluation & Validation approaches, rather than Application Mapping, suggesting that the main interest of academic-industrial collaborations may be for evaluation & validation approaches. This finding may be partially explained by analyzing the number of citations per year. This number tends to follow the number of studies in the categories (i.e., more studies would result in more citations). However, there is one exception to that: industrial studies have more citations than mixed studies, w.r.t. approaches for Application Mapping, possibly due to increased industrial interest. By investigating the approaches we observed that: (a) almost all studies propose or consider formal approaches; (b) model-driven and component-based approaches are preferred for tackling CES problems, specially due to the facilitation of (semi-) automatic verification and code generation; and (c) one of the most prominent challenges in designing CES, is the design of systems with mixed-criticality (i.e., critical and noncritical elements co-existing within the same system). In the following, we present the most important observations regarding each of the categories.

Multiple design flows have been proposed so far, which is in accordance to the high heterogeneity of CES. Each design flow aims at tackling specific problems, such as multi-tasking in multi-periodic synchronization [S206] or reliability-driven design in CES with mixed criticality [S257]. The most important observation is that the majority of the design flows didn't provide a complete lifecycle. They rather

Table 3.4: Classification of included studies by type of activity and nature

Type of Activity	Metric	Nature			Total
		Academic	Industrial	Mixed	
Design Flow	Number of studies	16	6	6	28
	Citations/year	*65.05*	*8.71*	*18.48*	*92.25*
Specification	Number of studies	44	11	16	71
	Citations/year	*181.84*	*31.30*	*39.50*	*252.64*
Application Mapping	Number of studies	97	21	32	150
	Citations/year	*298.42*	*85.97*	*72.33*	*456.72*
Evaluation & Validation	Number of studies	74	17	36	127
	Citations/year	*232.66*	*22.33*	*73.50*	*328.49*
Optimization	Number of studies	11	1	2	14
	Citations/year	*28.81*	*0.12*	*3.19*	*32.11*
Testing	Number of studies	7	2	4	13
	Citations/year	*31.96*	*2.40*	*6.83*	*41.19*

described how to tackle the specific issue within the system design. These incomplete flows are not surprising because every single CES entails a rather unique set of requirements that are tackled by combining different approaches. The most relevant studies are a generic design flow (from 1997) that served as inspiration to other flows [S16] and a safety-oriented and component-based design flow for vehicular systems [S102]. Approaches for design specification consist mostly of (semi-)formal languages or notations for representing different types of problems, such as specific forms of scheduling [S117, S225], or classes of constraints (commonly related to quality attributes such as safety or reliability) [S87, S244]. We highlight that most studies presenting specification approaches (approx. 80%) also presented approaches with other purposes (e.g., application mapping or evaluation & validation). The most relevant studies include the specification of time constraints in systems with mixed criticality [S225] and formal specification of safety constraints on higher-level design [S180].

The majority of the studies involve a variety of approaches for Application Mapping. Among these studies, approx. 30% proposed architectural approaches, i.e., architectures [S35, S94] or approaches for designing architectures (e.g., styles or patterns) [S121, S166]. We highlight that in the context of CES, communication architecture (e.g., time-triggered architecture [S35]) is a more relevant kind of architecture, due to its relevance on evaluating the hard constraints CES are subject to. In fact, this relevance is also evident by another common topic: scheduling of tasks/components, which corresponds to approx. 21% of the studies. Scheduling poses several challenges, from guaranteeing of time allocation to specific components, to integration with other models (e.g., fault-tolerance) to provide more accurate scheduling. Another common topic is software patterns, corresponding to approx. 9% of the studies, among which, design patterns were the most investigated [S105, S106, S137, S160, S259], followed by architectural [S121, S201], fault-tolerance [S191] and process patterns [S240]. As for the remainder of the studies, other scattered topics can be observed, from which the most recent/recurrent encompass approaches for modeling components w.r.t. various critical constraints (e.g., safety) and integration of models. The most relevant studies include the time-triggered architecture [S35], remote agent architecture [S13], a component-based approach for modeling safety [S102] and an approach for scheduling of mixed-criticality workload [S164].

Approaches for Evaluation & Validation comprise mostly formal methods for evaluating specific aspects of the design, such as scheduling of tasks [S51, S140, S225], fault-tolerance [S151, S192] and safety requirements [S74, S102]. In addition, there is a growing interest on model-driven approaches and object-oriented design. Classical approaches for verifying safety and reliability (e.g., fault-tree analysis—FTA—and failure mode and effects analysis) have been adapted to new design paradigms. For example, a component-based FTA was proposed in [S128] aiming at facilitating the certification of systems by reusing certified components. In addition, exploratory-based evaluation approaches (e.g., prototyping and simulation) are also broadly explored in order to evaluate designs [S21, S102, S168, and S216]. The most relevant studies present formal approaches for evaluating reliability and safety [S8, S225], as well as safety evaluation based on simulation [S102].

Finally, regarding Optimization and Testing approaches, the approaches are used for the same reason: improving the evaluation & validation of the designed systems [S51, S186, and S261]. Most of the approaches, including the most relevant approaches, tackle time constraints [S51, S248] and fault-tolerance issues [S48, S151].

3.4.3 Application Domains

The results on application domains suggest that the most studies (approx. 57%) report generic approaches, from which approx. 9% showed examples on specific application domains, e.g., automotive [S149, S257] and avionics [S225, S166]. Figure 3.6 presents the distribution of the studies, per year, according to the application domains. For comparison purposes, we plot the amount of studies reporting generic approaches. We note that studies that report approaches for specific domains often refer to more than one domain, e.g., support the design of avionic and space systems [S161].

By observing Figure 3.6, we notice that, besides constituting the majority, the number of studies reporting generic approaches is growing more than for any specific domain. This may suggest a trend or intention to work on unified technologies for developing CES. However, we also notice that the combined number of studies that focus on specific domains comprise almost half (approx. 48%) of the papers. Among the specific domains: avionics and automotive present the biggest growth. On the one hand, avionics is historically among the first application domains of CES and contains special regulations, which make the interchange of approaches more difficult. On the other hand, the automotive industry has been going through a series of technological innovations to provide several new features such as autonomous driving.

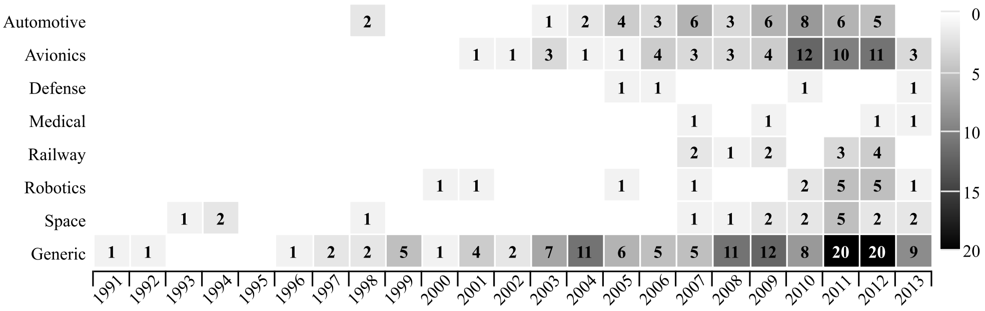


Figure 3.6: Number of studies per application domain, per year

To further analyze the influence of application domains on design approaches, we classified the primary studies according to their purpose. Table 3.5 presents the distribution of studies in each application domain among the five general purposes. We note that approaches serving more than one general purpose are counted for each of them. Based on Table 3.5, we observe that the distribution of studies on the application domains tend to be similar to the general distribution (Table 3.4). However, there is an exception for the medical and defense domains, as most studies report approaches for evaluation & validation rather than for application mapping. This may be either related to the low number of studies, or suggests a focus on this type of activity, perhaps motivated by specific industry standards or requirements of these domains. Another exception is that in the robotics domain the number of approaches for application mapping is quite higher (almost double) compared to evaluation & validation. Such disparity may be related to a larger variety of potential systems designs (large design space), which could result in more possibilities for mapping elements of the system. The disparity may also be related to a less regulated application domain that could in turn facilitate new design ideas to be implemented or experimented with.

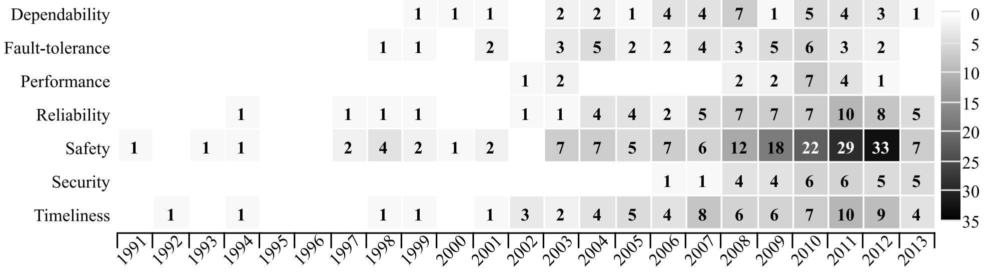
Table 3.5: Classification of primary studies by domain and purpose

Domain	Purpose					
	Design Flow	Specification	Application Mapping	Evaluation & Validation	Optimization	Test
Automotive	7	11	31	22	2	2
Avionics	7	20	32	30	0	4
Defense	0	1	1	4	0	1
Medical	0	1	1	3	0	0
Railway	3	5	7	7	0	2
Robotics	2	3	13	6	0	1
Space	5	8	13	12	0	3
Generic	13	36	77	61	13	5

3.4.4 Quality Attributes

CES are subject to constraints on critical quality attributes (CQA). In this section, we report on the CQAs that are tackled within each primary study, using the original terms of CQAs that are used in the studies (i.e. those terms used by the authors). Even though some qualities are similar (e.g. dependability, fault-tolerance and reliability) we have not tried to merge them. Our goal is not to create a new quality model, but to simply present how authors express the hard-constraints of CES. However, we checked whether each term has the same or similar definition among the authors (e.g., if security is always used to convey the same concerns). We further discuss the relationship between CQAs and their definition in Section 3.5.1. We note that each study may tackle one or more CQAs. In Figure 3.7, we present the number of studies, per year, tackling each critical quality attribute. We excluded two CQAs from this chart (power constraints and correctness) due to low number of papers (6 and 7, respectively).

By observing Figure 3.7, one can notice that the interest in the different CQAs has grown in a similar fashion, except for safety, which shows higher growth. Such interest is not surprising, as safety is a very common and challenging concern among CQAs. In addition, the emergence and/or growth of application domains such as automotive, home automation, unmanned vehicles (e.g., drones) that are intrinsically centered on safety, have likely contributed to the observed growth. It is also relevant to point out that, although less intense, the interest in timeliness and reliability has also grown more than the remaining CQAs. The aforementioned arguments regarding safety, may also explain this observation. For example, the interest in multi-core platforms, as well as systems with mixed-criticality requires careful



Timeliness includes *timing*, and *time-behavior*
Fault-tolerance includes *error-tolerance*

Figure 3.7: Number of studies tackling quality attributes, per year

scheduling of tasks, and assurance that no interference between system parts with different criticality.

To further characterize the primary studies, we investigate them with respect to purpose and application domain. In Figure 3.8, we present a bubble chart that depicts the distribution of the studies, based on CQAs (Y axis), with regards to the general purpose (X axis—left side) and the application domain (X axis—right side). The size of the bubble represents the number of studies, which is shown inside the bubble. On the one hand, the distribution of studies among purposes, for each CQA, is similar compared to each other as well as compared to the general data (see Section 3.4.2). To confirm that, we calculated Spearman's correlation between every pair of CQA and against the general data. All results were statistically significant and showed strong correlation (minimum coefficient of 0.899). This suggests that the distribution of research effort among different purposes is independent of CQAs. On the other hand, it is possible to observe a variation in the distribution of studies among application domains. For example, we notice that dependability displays a higher interest on the automotive domain (i.e., approx. 20% of the papers tackle this CQA), when compared against the average number of papers on dependability across domains (9%). We further investigated this observation by calculating the correlation between every pair of CQA, which showed that dependability has a weaker correlation with other CQAs (e.g., 0.667 with performance). This may suggest that these application domains are characterized by different constraints for the respective CQAs.

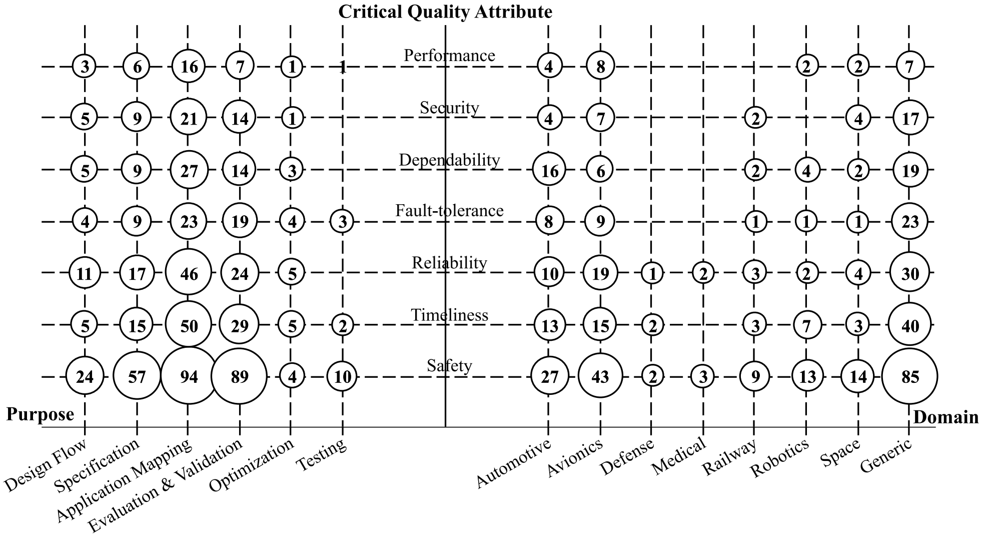


Figure 3.8: Classification of studies based on quality attribute, purpose, and application domain

3.4.5 Tools

During the data extraction, we observed that approx. 53% of the papers either proposed or explicitly mentioned the use of specific tools. We also identified several Reference Technology Platforms (RTPs) (Kacimi et al., 2014), which consist of a set of approaches (e.g., methods, workflows) and tools providing a generic solution that can be tailored to various applications. The RTPs extracted in our study are all part of large projects involving multiple partners from both academia and industry. In total, we identified 186 tools of different kinds (e.g., CAD, tool suites, etc.) and with various purposes (e.g., specification, application mapping, etc.). In addition, we noticed that some specification and/or modeling languages are an important part for many of these tools, e.g., serving as input format and base of the tool, or as exchange format between different tools. Therefore, we considered it relevant to include these languages in the results. Due to the number of identified tools, we summarized the results based on the general purposes presented in Section 3.4.2.

Table 3.6 shows the number of tools identified for each category (i.e., purpose). Within each category, we were able to define certain subcategories of tools representing specific purposes. We note that we include RTPs and IDEs (Integrated Development Environment), into the Design Flow category, as they support entire sets of activities. We also note that similar to approaches every tool may be classified in

Table 3.6: Summary of identified tools

Purpose	**Number of Tools**
Design Flow	12
IDE	*6*
RTP	*6*
Specification	15
Notation/Specification Language	*12*
Programming Language	*3*
Application Mapping	35
CAD	*14*
Model Transformation	*5*
Evaluation & Validation	32
Simulation	*9*
Model checking	*9*
Optimization	1
Testing	2

more than one category, e.g., a modeling tool that can import and export different models (i.e., Application Mapping category) as well as analyze them (i.e., Evaluation & Verification category). Furthermore, we note that the number of tools for subcategories do not necessarily add up to the number of the parent category. On the one hand, we only present subcategories with at least 3 tools (i.e. there were more subcategories with only 1 or 2 tools). On the other hand, tools may serve more than one purpose, which also affects subcategories. For example, SPIN is a verification tool with model checking and simulation capabilities, thus, counting for two subcategories. In the following we provide a brief description and the purpose of some relevant tools/languages, which we identified based on the number of studies referring to the tool/language, as well as on the amount of citations these studies have. More information the tools and languages can be found in the supplementary material (Feitosa, Ampatzoglou, Avgeriou, Affonso, Andrade, Felizardo and Nakagawa, 2017).

Summary of Languages

In Table 3.7, we list the top five recurrent languages within the primary studies, i.e., those discussed by three or more papers. We consider these languages relevant also due to the amount of citations obtained by the studies that refer to them. We observed that most languages are mentioned indirectly, i.e. not being the focus of the paper. For example, the Promela language is recurrent because researchers are interested in the SPIN verification tool, which defines models in Promela. In addi-

Table 3.7: Highlighted languages

Language	Number of Studies	Number of citations	CES specific
AADL	20	294	Yes
Promela	7	162	No
SystemC	7	51	No
Z	5	153	No
EAST-ADL	3	19	Yes

Table 3.8: Highlighted tools

Tool	Number of Studies	Number of citations	CES specific
Simulink	15	132	No
IPPAAL	8	79	No
DECOS	7	164	Yes
SPIN	7	162	No
NuSMV	4	112	No

tion, most languages are also not specific to CES, although they are heavily used for this class of systems. Languages (e.g., Z) were created to enable representation of formal/mathematical constraints, which are common to CES.

Summary of Tools

The top five tools according to the number of studies and citations are presented in Table 3.8. We observe that most tools are not specific to designing CES. We believe this is related to the fact that most tools in this list have Evaluation & Validation purposes. Tools from this category, are mainly focused on ensuring the hard constrains imposed w.r.t. meeting critical quality attributes; such CQAs are not particular to CES only. Finally, we notice that the tools focused on CES are mostly (a) from the Application Mapping category (e.g., modeling tools and schedulers), which are specialized for one or a group of application domains; and (b) RTPs and IDEs, which are tailored for this class of systems, and normally include some tools that are not specific to CES (e.g., verification tools).

3.4.6 Evidence Type

To investigate the maturity of the primary studies, we considered the type of evidence they provide. For that, we use the classification proposed by Alves et al.

(2010), as mentioned in Section 3.3.5. At the lowest level, the primary study does not provide any evidence, whereas at the highest level, the study provides evidence from actual use of the approach within an industrial application. In Figure 3.9, we present the distribution of the primary studies, per year, according to the evidence type. By observing Figure 3.9, one can notice that the amount of studies that provide evidence from academic studies has been growing considerably, exhibiting the highest growth among the six types of evidence. This also reflects the fact that most primary studies (approx. 55%) are supported by such type of evidence. This result is understandable, as studies performed in academic settings usually have a lower threshold to conduct than those performed in industrial settings. In addition, considering the hard constraints of CES, multiple studies may need to take place before a mature technology emerges and industrial studies can be performed. Interestingly, the second most common type of evidence is industrial studies (approx. 20%), which is one step further according to the classification of Alves et al. (2010), and may suggest successful transition of a fair number of technologies to industrial maturity level.

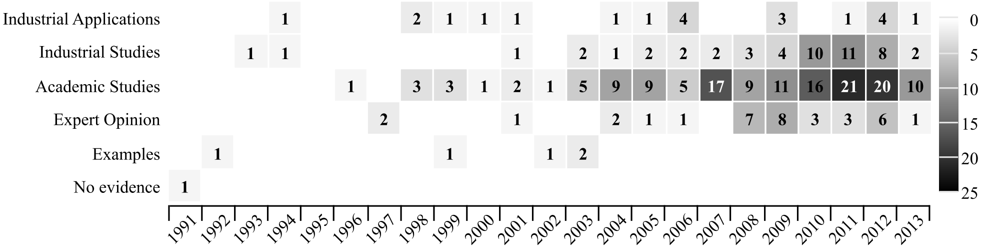


Figure 3.9: Number of studies per type of evidence, per year

Another interesting observation is that most studies are distributed among higher levels of evidence (academic studies, industrial studies and industrial applications). This may be, again, a consequence of the hard constraints imposed to CES, as tackling them would require stronger evidence to support the reported results. Another complimentary reason may be that embedded systems have been extensively investigated already, and management of hard constraints is not a new research topic for this class of system. Therefore, much of the exploratory research that has been done for embedded systems is now reused to investigate CES. To further investigate the evidence type, we classified the studies according to the purpose that their approaches serve, as well as the application domain. Similar to Figure 3.8, Figure 3.10 depicts the distribution of the studies, based on evidence type (Y axis), with respect to the purpose (X axis—left side) and the application domain (X axis—right side).

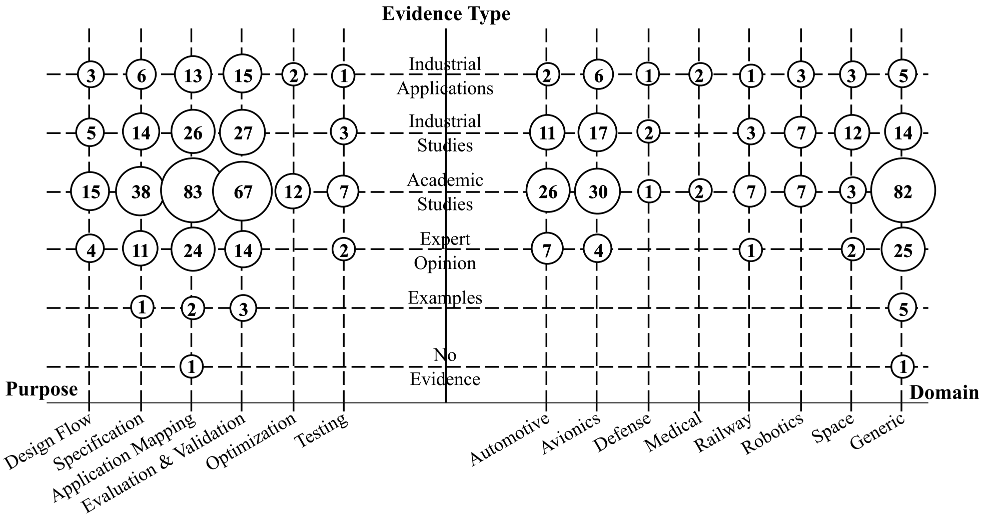


Figure 3.10: Classification of studies based on evidence type, purpose, and application domain

When verifying the distribution according to purpose, we observe that it follows a similar trend to that of the general data (presented in Section 3.4.2). We checked this hypothesis by calculating the correlation between each pair of evidence type, which showed a minimum correlation coefficient of 0.900. Conversely, while a visual inspection of the distribution according to domain suggests similarities between evidence types, the statistical correlation reveals minor differences between types of evidence, with coefficients varying from 0.500 to 0.927. These minor differences suggest that the application domain may affect what kind of research is performed.

3.5 Discussion

3.5.1 Relationship between Quality Attributes

The approaches investigated in this mapping study tackle various CQAs, as presented in Section 3.4.4. While investigating this research question (RQ_3), we recorded the CQAs as used by the authors, i.e., we neither grouped nor merged any quality attributes, based on the definition used or implied in the primary studies. However, it is undeniable that some CQAs are related and, therefore, the identified quality attributes should be further investigated / synthesized. In this subsection, we group CQAs that have a similar or related meaning and map them to a quality model. For this purpose, we consider: (a) the SQuaRE quality model (ISO/IEC, 2011) which is a well-known quality model adopted by both researchers and practitioners; and (b) the ISO/IEC/IEEE vocabulary for system and software engineering (ISO/IEC/IEEE, 2010), which is used within SQuaRE and provides additional definitions. We note that other quality models could be used to map the CQAs and that we do not assume that SQuaRE is the best model. We selected this model due to our experience with it and the possibility to fit all our recorded CQAs and observed terminologies. In Table 3.9 we present the CQAs identified in this study (presented in Section 3.4.4) on the right, and the characteristic (i.e., quality attribute) from SQuaRE to which they are mapped on the left. We note that SQuaRE presents a set of characteristics (left column of Table 3.9) and sub-characteristics (e.g. sub-characteristics of Performance Efficiency are Time Behavior, Resource Utilization and Capacity), which were both used to map CQAs. In addition, a CQA can be directly related if the terms are equivalent (e.g., safety maps to freedom from risk), or indirectly related if it is one of the aspects of the main quality attribute (e.g., correctness is a sub-characteristic of Functional suitability) or if it is related to one of them (e.g., energy efficiency regards Resource utilization, i.e., sub-characteristic of performance).

Table 3.9: Grouping and mapping of critical quality attributes

CQA from SQuaRE	**Identified CQA**
Functional suitability	correctness
Security	security
Performance efficiency	performance energy efficiency timeliness
Reliability	reliability fault tolerance dependability
Freedom from risk	safety

Correctness and security are directly mapped, since they similarly referred in the primary studies. However, the grouping of the remainder CQAs is not as straightforward. Performance efficiency is defined as the degree to which functionalities are delivered within given constraints (ISO/IEC, 2011), i.e., how well the system uses its resources to accomplish the designed functions. This definition encompasses the interpretations of performance, energy efficiency, and timeliness among the primary studies. Fault tolerance is a well-known aspect of reliability and the interpretations of the authors meet the definition of the sub-characteristic in SQuaRE (also named Fault tolerance). Although dependability is commonly addressed as a separate quality attribute, we decided to map it to Reliability. Dependability is not part of SQuaRE but it is explained within the description of reliability. It comprises a more subjective definition, which is not easily quantifiable, and reflects whether or not a system can be trusted (ISO/IEC/IEEE, 2010). Due to its subjective definition, dependability is commonly improved through addressing other, more objective, quality attributes that can contribute to the trustworthiness of the system, in particular, reliability, maintainability, and availability. By observing the primary studies of our mapping, it is also clear that dependability is commonly used as proxy to other quality attributes, in particular, aspects of reliability, such as fault tolerance. Therefore, since the primary studies exploit dependability mostly as a proxy to reliability, we decided to group them together. Safety is another subjective CQAs, which is mentioned within SQuaRE's model for quality in use, i.e., how well the product can be used by specific users (ISO/IEC, 2011). Similar to dependability, safety is commonly used as a proxy to other quality attributes, although not always the same ones. Particularly, safety is related to the avoidance of hazardous situations (i.e., that lead to endangerment of humans, environment or properties), which can originate from various sources, depending on the system. In our study, we identified connections between safety and various aspects: security [S215], performance, correctness [S50, S198] and fault-tolerance [S50, S84]. For example, when using a Time-Triggered Architecture (TTA) for communication (instead of an event-triggered one), timeliness become a safety threat.

In summary, CQAs as defined in primary studies are uniformly understood (i.e. their definitions are the same or similar across the studies) and that some can be grouped based on similarity. This culminated into the identification of five attributes: Functional Suitability, Security, Performance efficiency, Reliability, and Safety (Freedom from risk). We acknowledge that other CQAs may exist in individual cases depending on application-specific constraints. However, these five QA are by far the most recurrent ones. We also noticed that Safety is more abstract, since it depends on other CQAs. Therefore, is achieved by meeting requirements related to other CQAs. Furthermore, we note that identifying these CQAs is not always a trivial task as different components in the same systems may pose different constraints, i.e., may be subject to different kinds of hazards. A common approach to handle this mixed criticality is the use of integrity levels (ISO/IEC, 2015), which reflect the degree of compliance within a certain characteristic. Components with different integrity levels will be subject to different safety checks, which may also reflect the different concerns of that level. For example, the drive-by-wire feature is subject to hard reliability checks, while GPS navigation should only be assured to not interfere with the critical components. Therefore, it is important to identify and monitor the CQAs that are tightly related to safety.

3.5.2 Domain-Specific Research for CES

In Sections 4.3 through 4.6, we presented an overview of the primary studies with respect to application domains, as well as how other facets (e.g., evidence type) related to domains. In summary, we did not notice major differences across application domains regarding which CQAs are the most relevant. This observation might be an indication that CQA-related challenges in CES are common to all application domains and have similar relevance. The only difference we observed was that studies focused on the automotive domain seem more concerned about dependability rather than reliability. However, these two fall under the umbrella quality of reliability in the SQuaRE model (see Section 3.5.1). Furthermore, we also notice that domains may influence the kind of research that is performed; for example, most studies on medical and defense domains focused on approaches for evaluation & validation rather than application mapping (as the general trend).

The difference between domains becomes clearer when looking at the type of evidence that studies provide (see Section 3.4.6). We separated the studies into three groups and verified their distribution among the different types of evidence (see Figure 3.11). The three groups consist of studies that: (a) focus on a specific domain; (b) do not focus on any domain but present an example of application on a specific domain; and (c) neither focus nor present an example on specific domains. We notice that application domains become more relevant when a technology is being transferred to industry, as the two rightmost types of evidence (Industrial Study and Industrial Application) account mostly for studies that focus on application domains.

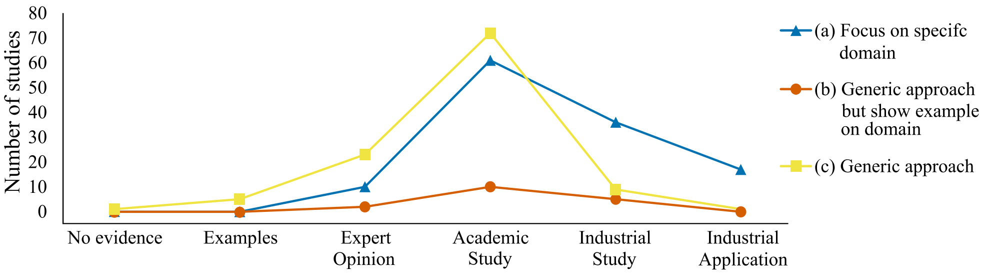


Figure 3.11: Distribution of studies according to type of evidence and application domain

It is understandable that studies conducted with industrial partners or in an industrial setting are focused on specific domains, as companies are by and large interested into applying approaches on certain products, which in turn fall under specific domains. As expected, generic approaches that solve domain-independent problems are first validated in academic settings, and subsequently find applications in industry that in turn customize and validate them in specific application domains. The opposite is also possible: there are also technologies that initially emerge as domain-specific solutions and are later applied to other domains. For example, the Architecture Analysis and Design Language (AADL) was standardized by the Society of Automotive Engineers (SAE) with focus on the avionics domain[3] and is currently being applied in other CES domains.

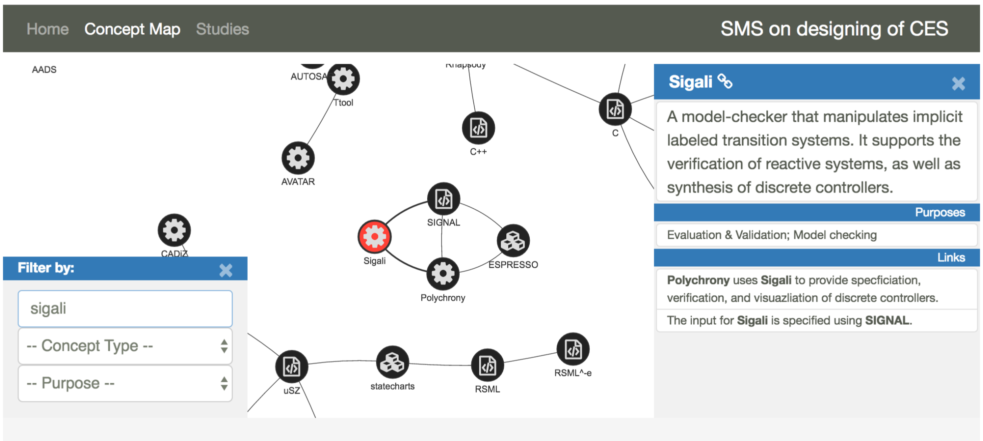
3.5.3 Relationships among Approaches, Tools, and Languages

The data analysis in this SMS resulted in the identification of many concepts related to the research questions, namely approaches, tools, languages, critical quality attributes, and application domains, as well as relationships between them. While we were able to present and discuss all CQAs and application domains found in the primary studies (see Sections 4.3, 4.4, 5.1 and 5.2), the amounts of approaches, tools and languages was too large to present and discuss all concepts and relationships. To tackle this issue, we created a concept map to help us visualize these approaches, tools, and languages and identify relevant findings.

The concept map was created as a webpage that features an interactive interface, which is available[4]. To avoid loss of information, we also created a text version of the concept map. The text version and source code of the web version are available within the supplementary material (Feitosa, Ampatzoglou, Avgeriou, Affonso, Andrade, Felizardo and Nakagawa, 2017). In Figure 3.12, we show a screenshot of the concept map and its interface. The concept map consists of a network in which nodes represent concepts and edges relationships. Each type of concept (i.e., approach, tool or language) is represented by an icon for easy identification. The mouse's button and wheel can be used to pan and zoom in and out on the network. Upon clicking on a concept, an information panel is prompted on the right side, showing: (a) name of the concept, which is a link if a URL (Uniform Resource Locator) is available (shown by the chain icon next to the name); (b) a brief description of the concept; (c) the list of purposes, according to our classification scheme; and (d) a list of relationships (i.e., links) attributed to the concept. The relationship between concepts can be of two types: "use / is used" (e.g., "Polychrony uses Sigale to provide specification ... of discrete controllers"), or "is kind of" (e.g., "SystemC is a subset of C++").

[3] Note that SAE does not limit itself to the automotive domain

[4] http://feitosa-daniel.github.io/sms-ces-design



sigali

-- Concept Type --

-- Purpose --

A model-checker that manipulates implicit labeled transition systems. It supports the verification of reactive systems, as well as synthesis of discrete controllers.

Evaluation & Validation; Model checking

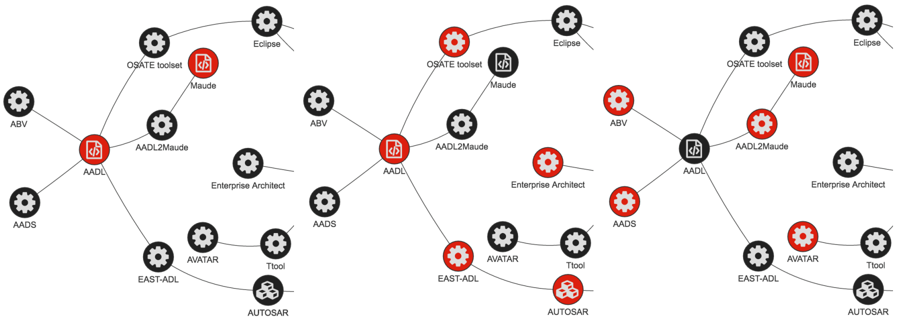
Polychrony uses **Sigali** to provide specficiation, verification, and visuazliation of discrete controllers.

The input for **Sigali** is specified using **SIGNAL**.

Figure 3.12: Screenshot of the concept map interactive interface

The interface also provides a feature to filter concepts based on name, type of concepts, or purpose. Upon typing on the name field or selecting type of concept or purpose, the filtered items are highlighted in red (see Figure 3.12). For example, in the screenshot we typed “sigali” and the tool “Sigali” was automatically highlighted (the search looks for partial matches and is not case sensitive). After that, we clicked on the node, which prompted the information panel on the right. Finally, the interface is responsive, i.e., it adapts to different screen sizes (e.g., smartphones), which improves the usability of the concepts map.

Based on the concept map, we can make several observations. In the following, we provide one such observation as an example, also explaining how we identified it. We note that the main purpose of the concept map is to support the investigation of its concepts by third-parties and, therefore, we encourage the reader to further analyze it. The Architecture Analysis and Design Language (AADL) appears to be a rather mature technology. The results of the study showed that AADL is cited in multiple papers (see Section 3.4.5). In addition, by looking at the concept map we notice a fair number of related concepts (see Figure 3.13) when compared against the average of 2.13 edges per node, and we notice that there are related concepts that serve different purposes: (a) specification, (b) application mapping, and (c) evaluation & validation. In particular, there is a toolset that is able to read AADL models, tools to evaluate AADL models and a language (EAST-ADL) that is partially derived from AADL.



(a) (b) (c)

Figure 3.13: Part of the concept map surrounding AADL

3.5.4 Implications to Researchers and Practitioners

The results and discussion presented in this SMS have potential value for both researchers and practitioners. The information compiled in this study may support readers that want to get acquainted with the design process of CES or may be interested in specific outcomes, e.g., identified CQAs and how they are tackled by primary studies. Researchers can use the information in this SMS to identify work that is related or that can contribute to theirs, as well as identify opportunities for future work. For example, researchers interested in a specific application domain have access pointers to the existing literature, as well as how studies are distributed within the domain. We envisage similar learning opportunities to practitioners, through a more practical perspective. For example, practitioners can investigate a tool that is being considered for the designing of a new system or investigate the ecosystem around an approach, i.e., tools and related approaches.

In addition, we specifically aimed at the reuse of the information collected during our SMS when we created the concept map, which contains the complete set of approaches, tools and languages. Based on the information and features provided by the user interface, we believe that the concept map is valuable to both practitioners and researchers. Regarding practitioners, it can be used to support the exploration of problem and solution spaces while designing CESs. For example, using filters, one is able to search for approaches and or tools that fit the requirements of the systems (e.g., model-checking of models specified in SIGNAL). Also, if one has decided for a specific approach or tool, she can also explore related concepts and identify alternatives or tools that support the approach (e.g., tools that evaluate Binary Decision Diagrams). Regarding researchers, the concept map helps identifying potential links between different research results. For example, researchers interested into investigating a certain approach can use the concept map to easily visualize some of the involved approaches and tools that support it. We note that despite our great effort on collecting and analyzing the selected studies, the concepts and relationships presented in this map do not present the entire set of approaches, tools and languages available to design CES. Therefore, we hope that by providing access to the concept map, we can support others on developing it even further.

3.6 Threats to Validity

Concerning studies identification, the main threat is that the automatic search may not have been able to collect all relevant primary studies, i.e., the search string was not as inclusive as necessary or the considered digital libraries did not include all relevant venues. To mitigate this risk, we defined a gold standard and ensured that the automatic search returned all papers in the gold standard. In addition, we included digital libraries of the main publishers in the topic, and Scopus, which indexes papers from additional venues. Another potential threat is that the inclusion and exclusion criteria may have left relevant studies out of the final set of primary studies. This was mitigated not only by the usage of the gold standard but also by having key points of our protocol (e.g., inclusion and exclusion criteria) inspected by other external researchers with experience in CES. To mitigate risks related to data collection and analysis, we considered several strategies. The filtering of papers and data extraction involved at least two researchers on every step, while there were extensive discussions on topics such as selection criteria and understanding of CES terminology. In addition, the alignment of researchers involved in these steps where verified by calculating the Cohen's kappa coefficient between them. For data analysis, we applied frequency analysis, cross-tabulation and statistical tests, which are less prone to researcher bias. However, we acknowledge that our results are limited to the set of design approaches, CQAs, and application domains that were discussed in the collected primary studies. Although considering non-peer-reviewed literature was out of the scope of our SMS, we argue that the digital libraries we considered, do catalog most of the work relevant to the research of CES design.

Finally, to mitigate replicability threats, the steps of our study were clearly stated in our protocol and can be reproduced by other researchers. However, we acknowledge that the reproduction of the SMS by other researchers may lead to slight different sets of primary studies due to biases, e.g., when applying the inclusion and exclusion criteria. We mitigated this threat to some extent by comprehensively documenting faced challenges and decisions made upon them. Thus, despite some potential minor differences, we believe that the results and observations would be predominantly similar in replication studies.

3.7 Conclusions

In this chapter, we presented a Systematic Mapping Study (SMS) on designing Critical Embedded Systems (CES) that investigated five facets: (a) approaches for designing CES; (b) application domains for which these approaches are developed; (c) Critical Quality Attributes (CQAs) considered on these approaches; (d) tools used for designing CES; and (e) type of evidence provided by these approaches. We considered five digital libraries and collected an initial amount of 1,673 primary studies, which were then filtered, resulting in 269 selected primary studies. Subsequently, we extracted and analyzed all data necessary to answer our research questions.

The results of our SMS show that the body of knowledge on designing CES is vast, and this is partially due to the overlap of knowledge with other classes of systems such as hard real-time systems. Results also suggest that the CQAs that are relevant to the design of CES, are common for this whole class of systems, i.e. they are mostly independent of application domain. The main contributions of our work are the classification scheme for approaches and tooling, the provided collection of CQAs and approaches (with associated tools), as well as the webpage that supports exploring this information. We believe that both researchers and practitioners can benefit from these contributions, taking advantage of our provided overview of this vast body of knowledge; they can thus focus on more relevant tasks such as identification of related and future work, and exploration of problem and solution spaces.

Based on our results and observations we identified several approaches and practices that could be used to address the main problem of this PhD project and that have not been thoroughly explored so far. Among these approaches and practices, software patterns seems to be one of the most promising in terms of managing quality attributes. The description of software patterns includes known consequences on the QAs (and well documented in the literature) and can be used to assess the overall impact of a design on QAs. Thus, we decided to explore the practice of applying software patterns in CES design, in particular GoF design patterns. This leads to the next chapter, which reports on the relationship between design patterns and a subset of CQAs, namely correctness, performance and security. We selected these CQAs due to their relevance to both practitioners and researchers, and a lack of studies investigating the relationship between them and GoF design patterns. As the outcome of such investigation may have great value not only to CES but also to other domains, the next chapters do not focus only on CES. Moreover, the selected CQAs are now referred to as runtime quality attributes.

Based on:

Daniel Feitosa, Apostolos Ampatzoglou, Paris Avgeriou, Alexander Chatzigeorgiou and Elisa Y. Nakagawa, (in press) *"What can Violations of Good Practices tell about the Relationship between GoF Patterns and Run-time Quality Attributes?,"* in Information and Software Technology, DOI:10.1016/j.infsof.2018.07.014

Chapter 4

What can Violations of Good Practices tell about the Relationship between GoF Patterns and Runtime Quality Attributes?

Abstract

GoF patterns have been extensively studied with respect to the benefit they provide as problem-solving, communication and quality improvement mechanisms. The latter has been mostly investigated through empirical studies, but some aspects of quality (esp. runtime ones) are still under-investigated. In this chapter, we study if the presence of patterns enforces the conformance to good coding practices. To achieve this goal, we performed a case study on approximately 13,000 classes retrieved from five open-source projects. In particular, we explore the relationship between the presence of GoF design patterns and violations of good practices related to source code correctness, performance and security, via static analysis. The obtained results suggest that classes not participating in patterns are more probable to violate good coding practices for correctness, performance and security. In a more fine-grained level of analysis, by focusing on specific patterns, we observed that patterns with more complex structure (e.g., Decorator) and pattern roles that are more change-prone (e.g., Subclasses) are more likely to be associated with a higher number of violations (up to 50 times more violations). This finding implies that investing in a well-thought architecture based on best practices, such as patterns, is often accompanied with cleaner code with fewer violations.

4.1 Introduction

Design patterns have been introduced in the software engineering literature by Gamma et al. (1995) (known as the Gang of Four (GoF)—Gamma, Helm, Johnson, and Vlissides), aiming to provide common solutions to recurring problems, while designing object-oriented (OO) systems. The GoF catalog includes 23 patterns, organized into three categories (structural, behavioral, and creational), based on their purpose (Gamma et al., 1995). Since their inception, GoF patterns have been widely explored by both researchers and practitioners, and are currently considered as a common practice for software development. In addition to their original purpose of solving OO design problems (Riaz et al., 2015), their effect on quality attributes (QAs) has also been widely investigated, according to two mapping studies by Ampatzoglou, Charalampidou and Stamelos (2013b) and Bafandeh Mayvan et al. (2017). However, the current state of the research has two main limitations:

- ***limited number of studies related to runtime qualities***. In particular, on the one hand, several empirical studies have explored the impact of GoF design patterns on design-time QAs such as modifiability and reusability (see for example (Bafandeh Mayvan et al., 2017)). On the other hand, research on the effect of GoF patterns on runtime QAs, such as security and performance, is fairly limited (Bafandeh Mayvan et al., 2017). Although GoF patterns are not originally intended to serve any runtime QA in particular, some indirect effect, either positive or negative, is to be expected. For instance, developers often use GoF design patterns as communication mechanisms, which facilitate the understanding of each other's code. As a consequence, code smells, such as Message Chains and Middle Man, can be avoided and, thus, performance improved, since the number of method calls is decreased.

- ***runtime qualities have been explored mostly through dynamic analysis***. Until now, the limited work done on studying the effect of GoF patterns on runtime QAs was performed mostly by using dynamic analysis, i.e., by exploring the observed effects during the execution of a system. For example, researchers have used profilers for extracting memory usage or energy consumption data to investigate performance (e.g., Litke et al. (2007) and Sahin et al. (2012)). An alternative to dynamic analysis, for investigating the same phenomenon, is the employment of static analysis. Static analysis is an established method for performing code quality analysis (Ayewah et al., 2008; Ayewah and Pugh, 2010; Sadowski et al., 2015; Tripathi and Gupta, 2014; Zheng et al., 2006), mostly because it is based upon an artifact that is always available to quality engineers (i.e., source code). There are also a few research efforts that employed static analysis for assessing runtime qualities, e.g., (Schilling and Alam, 2008) for assessing reliability with GERT, (Khalid et al., 2016) for assessing performance with FindBugs, (AlBreiki and Mahmoud, 2014; Díaz and Bermejo, 2013; Goseva-Popstojanova and Perhinschi, 2015) for assessing security, etc. With static analysis, one would be able to explore the underlying relationship between GoF patterns and runtime QAs without executing the systems in which they are instantiated. We believe that statically detecting violations of good coding practices that affect runtime QAs is complementary to dynamic analysis, since it promotes the assessment of different artifacts (e.g., violations instead of profilers output) and by using a different approach (static instead of dynamic analysis). We note that, even if the introduction of design patterns might not directly aim at removing violations, it may lead to a 'cleaner' and well-designed architecture, accompanied by better coding practices. Nevertheless, not all parts of the architecture can benefit from the introduction of patterns, as the pattern goal might be irrelevant to the functionality/design of that part. Design patterns are not a panacea as they cannot solve all design problems and all design problems cannot be solved by a design pattern.

Motivated by the aforementioned limitations, in this study, we exploit static analysis to explore whether the application of GoF patterns can be associated to the existence of violations of good coding practices related to three runtime QAs, namely, correctness, performance and security, as defined in the SQuaRE quality model (ISO/IEC, 2011). We note that we consider correctness a runtime QA because, as performance and security, it is discernible at runtime (Bourque and Fairley, 2014). We selected these QAs as they are highly relevant for both practitioners and researchers, with considerable literature addressing them. However, there is a lack of studies investigating them from the proposed perspective, i.e., by using static analysis to examine the effect of GoF patterns on them.

To estimate the effect of GoF patterns on the aforementioned qualities, we adopted the same approach used by Sahin et al. (2012) and Gatrell and Counsell (2011), i.e., we compare pattern-participating (PP) parts of the system against non-pattern-participating (NPP) parts. Similar to Ampatzoglou et al. (2015) and Aversano et al. (2009), our investigation is performed at class-level to standardize data collection and source code analysis. Specifically, by working on class-level we can discriminate between: PP classes—that participate in pattern occurrences—and NPP classes. Additionally, we further classify PP classes into: single-pattern-participating (SPP), i.e., those that participate in exactly one pattern occurrence; and coupled-pattern-participating (CPP), i.e., those that participate in more than one pattern occurrences. According to the literature (e.g., (Ampatzoglou et al., 2015; Khomh et al., 2009; McNatt and Bieman, 2001)), these two types of pattern participation can lead to diverse effects on QAs; therefore, we treat them separately in this study.

To explore the relationship of GoF patterns with the aforementioned QAs, we compare quality levels of classes (quantified by the number of violations) measured from four different perspectives, serving the sub-goals of this study:

1. *Pattern participation*: by clustering classes according to their pattern participation, i.e., NPP, SPP and CPP. As previously mentioned, this perspective allows us to compare the number of violations concentrated in SPP and CPP elements against NPP elements.

2. *Pattern category*: by clustering classes according to the pattern category in which they participate, i.e., creational, behavioral and structural. This perspective allows us to investigate whether there are differences in the relationship of GoF patterns of different categories on QAs.

3. *Pattern*: by clustering classes according to the pattern in which they participate (e.g., Singleton, State, Strategy, etc.). This perspective allows finer grained observations of the relationship of applying GoF patterns and runtime QAs. It is rather common when investigating GoF patterns, as it represents the unit of the proposed solutions (i.e., the patterns) and can inform designers of both benefits and disadvantages of their usage.

4. *Pattern role*: by clustering classes according to the role they play in the pattern occurrence (e.g., Concrete State, Concrete Prototype, etc.). This perspective represents the finest-grained analysis that can be performed under the considered level (i.e., class-level). It allows us to investigate if the number of violations in classes is related to specific roles or to the joint effect of all roles.

We note that the last three perspectives involve SPP classes only. This decision is based on the fact that for coupled pattern occurrences it is not possible to separate the individual influence of each pattern. Moreover, based on literature, coupled design pattern occurrences have a different effect on QAs compared to single occurrences (Ampatzoglou et al., 2015; Khomh et al., 2009; McNatt and Bieman, 2001).

Summarizing the above, the main contribution of our work is that it explores the link between patterns and aspects of quality that are not evident as problems yet. For example, classes that do not participate in design pattern instances may be more prone to the existence of performance issues (e.g., unnecessary data boxing[1] and unboxing, allocation of an object only to get its class, or inefficient use of collections). If the number of concentrated violations becomes high, it may result in a perceivable decrease of quality regarding performance. Therefore, the early identification of such issues, and their potential link to some GoF patterns, is considered important.

[1] Boxing and unboxing refers to encapsulating data from one type into another, causing the value to be wrapped, leading in turn to an extra hop in order to access the value (by unboxing it).

Another contribution is that our work increases the validity of the empirical results on the subject in terms of data source and methodological triangulation (Patton, 2014). In other words, we can reach a safer conclusion by gathering data from different sources and using different methods. Approaching a problem from different perspectives is especially important from an empirical software engineering viewpoint in the sense that every method poses different threats to validity (e.g., the use of profilers provides an overhead to program execution that is difficult to filter out). Additionally, some use cases may never be executed during dynamic analysis, leading to the omission of the underlying violations, but they will show up in static analysis, as it covers the whole codebase. Therefore, if studies using different methods reach similar conclusions, the results can be more uniformly interpreted.

The remainder of this chapter is organized as follows: related work is presented in Section 4.2, along with a discussion of the main points of differentiation of this study. In Section 4.3, we present the case study design, whereas its results are presented in Section 4.4, followed by a discussion of the findings in Section 4.5. Finally, we report on threats to validity and actions taken to mitigate them in Section 4.6, and draw the conclusions in Section 4.7.

4.2 Related Work

In this section, we present related work that discusses the relationship between the application of design patterns and runtime QAs. We clarify that we only present studies that consider GoF design patterns; therefore, we excluded studies that use different patterns, e.g., architectural (Cohen, 2007) or security (Aleksy et al., 2006) patterns. This section is organized into four subsections: First, we present the related work, grouped by the QAs addressed in this study, i.e., correctness (see Section 4.2.1), performance (see Section 4.2.2) and security (see Section 4.2.3). Next, we summarize this section and present the main points of advancement of our work (Section 4.2.4). Most of the related work discussed in this section were retrieved from a mapping study on GoF design patterns, by Ampatzoglou, Charalampidou and Stamelos (2013b), and on a literature survey on the impact of patterns on quality, by Ali and Elish (2013).

4.2.1 Design Patterns and Correctness

Vokac (2004) analyzed the correlation of five design patterns and correctness in a large commercial product (written in C++, with approx. 500KLOC). The product was investigated over weekly snapshots of the source code, during a period of three years. For each snapshot, the correctness of pattern-participating classes was measured in terms of number of defects, as collected from the issue tracking system. The results of the study suggest that Factory, Observer, Singleton, and Template Method patterns are correlated to higher defect frequency in source code. Additionally, Singleton and Observer seem to be often used in complex parts of the project (i.e., with more code, and higher defect frequency).

Ampatzoglou, Kritikos, Arvanitou, Gortzis, Chatziasimidis and Stamelos (2011) investigated the correlation between 12 design patterns and correctness. For that, they performed a case study involving 94 software projects in the game application domain. In this study, information was collected regarding bug tracking and pattern instances from each version of every software. During the analysis, each pattern was analyzed separately in order to identify correlations between the number of defects and pattern instances. The results of the study suggest that specific design patterns are related to higher defect frequency, although the presence of pattern occurrences (without examining each pattern separately) seems not to be correlated with such a frequency.

Gatrell and Counsell (2011) investigated the effect of 11 design patterns on correctness by analyzing a commercial project written in C# (with approx. 266KLOC). For that, PP classes were manually collected and compared against NPP classes over a two-year period, correlating them with the fault history provided by the source control system, aiming at finding fault-prone classes. The results of the study suggest that PP classes are more fault-prone than NPP classes, as well as that this is related to both a higher number and the size of changes in NPP classes. Additionally, the authors characterized Adapter, Template Method and Singleton as the most fault-prone patterns.

Aversano et al. (2009) investigated the relationship between correctness of pattern participants and the scattering degree of concerns[2] that communicate with them. For that, occurrences of 12 design patterns were extracted from several snapshots of three open-source projects, and the correctness was measured in terms of code defects. The results of this study suggest that patterns that induce crosscutting concerns (i.e., implemented across several classes spread along the system (Aversano, Cerulo and Penta, 2007)) are correlated to a higher number of defects in their participants.

4.2.2 Design Patterns and Performance

Afacan (2011) investigated the effect of the State design pattern on performance of a Digital Signal Processor (DSP). The author compared three implementations for a

[2] According to Aversano et al., it is how spread, among classes, is the implementation of a concern.

state machine: in C, C++, and C++ using the State design pattern. For that, performance was measured in terms of execution time (in clock cycles, and μsec), and required memory (in 16-bit words). The results suggest that usage of the State design pattern has a negative effect on the performance of a system. However, the author also reports that the gain in architectural aspects is worth the expectedly small loss in performance.

Rudzki (2004) investigated the effect of design patterns on performance. For that, two design patterns (Facade and Command) were compared as alternative solutions to each other. These patterns were used for implementing two different solutions for accessing services of business layer from a sample Java application. Their performance was measured in different deployment configurations, using four metrics: throughput, response time, number of correctly served requests, and number of requests. The results of the study suggest that, in general, Facade provided a better performance than Command. However, some results were hard to interpret due to noise in the measured values.

Chantarasathaporn and Srisa-an (Chantarasathaporn and Srisa-an, 2006) proposed a pattern instantiation for the Factory pattern (Gamma et al., 1995). This variant consists of an energy conscious implementation of the pattern by using C# language. In order to create an instantiation for power limited systems, the authors evaluated several options that varied in terms of component structure (i.e., class or struct) and type (i.e., static or non-static). The energy consumption was measured using four metrics (obtained via profiler): User Processor Time (UPT), Privileged Processor Time (PPT), Total Processor Time (TPT) and Memory used by the specific software process. The results of the study suggest that the modified Factory Method consumes around 11% less CPU time than the regular implementation.

Sahin et al. (2012) investigated the energy consumption of design patterns. For that, they considered 15 design patterns, five from each of the categories proposed by Gamma et al. (1995), measuring the difference in energy consumption between two versions of the same software (before and after applying the pattern). For measuring the energy consumption, the authors used a tool created by them, which is also introduced in their work. The results of this study show that: (a) design patterns can increase or decrease the energy usage; (b) the impact in energy consumption is not necessarily similar for pattern within the same category; and (c) energy usage is unlikely to be predicted by considering design-level artifacts only.

Finally, Litke et al. (2007) investigated changes in the energy consumption due to the application of three different design patterns. For that, they used a profiler for measuring: memory accesses to the instruction memory; memory accesses to the data memory; and dissipated energy within the processor core. The results of the study show that the application of design patterns does not necessarily imply a change of energy consumption.

4.2.3 Design Patterns and Security

To the best of our knowledge, there is a lack of empirical studies investigating security aspects of GoF design patterns; however, we were able to identify one descriptive study. Ferraz et al. (2009) relate the 12 common types of security requirements proposed by Firesmith (2003) to the GoF pattern categories (Gamma et al., 1995). The authors suggest that using an initial set of GoF patterns might substantially reduce the effort required to fulfill security requirements in the future. However, no empirical analysis was performed to evaluate the proposal.

A possible explanation on the lack of related work on the relationship between security and GoF patterns is the fact that GoF patterns were not originally intended to serve security requirements (VanHilst and Fernandez, 2007); hence the existence of specialized solutions known as security patterns (Hafiz et al., 2007). Thus, we are not interested in investigating whether the use of GoF patterns promotes security, but on the contrary, if the application of GoF patterns leads to violations of security good practices.

4.2.4 Overview of Related Work

The main differences of our study compared to the related work are summarized in Table 4.1. In particular, we compare the studies with respect to three aspects:

(a) ***Objectives***: The conceptual elements of the work, i.e., studied QAs; studied GoF patterns; and type of approach measuring QAs.

(b) ***Empirical setting***: The empirical setup of the studies, i.e., type of validation; number of used projects; level of measurement (i.e., unit from which the QA was assessed); and the number of assessed classes (a dash indicates that it was not possible to find or estimate the number of classes).

(c) ***Ability to compare results***: The elements of the analysis that are comparable to our study, i.e., whether or not PP components are compared against NPP components; granularity of the pattern investigation—i.e., category, pattern and role—whether or not there is a distinction between SPP and CPP.

Based on this overview, the advancements of this study compared to the state-of-the-art research are:

- *It investigates three runtime QAs using* ***static analysis***, providing evidence on their potential relationship to the application of design patterns;

Table 4.1: Overview of related work

#ref	Objectives			Empirical setting				Ability to compare results		
	QA	patterns	approach	validation	projects	level	classes	PP vs. NPP	granularity	SPP ≠ CPP
[19]	correctness	11[a]	dynamic	case study	1	class	7,439	yes	pattern	no
[21]	correctness	12[b]	dynamic	case study	3	class	~10,000	no	pattern	no
[28]	correctness	5[c]	dynamic	case study	1	class	1,550	no	pattern	yes
[29]	correctness	12[b]	dynamic	case study	94	system	~85,000	no	pattern	no
[5]	performance	3[d]	dynamic	example	6	pattern instance	~30	yes	pattern	yes*
[6]	performance	15[e]	dynamic	example	15	pattern instance	~250	yes	category, pattern	yes*
[31]	performance	State	dynamic	case study	1	system	8	yes	pattern	no
[32]	performance	Facade, Command	dynamic	case study	1	pattern instance	-	no	pattern	no
[33]	performance	Factory Method	dynamic	example	1	pattern instance	~20	no	pattern	yes*
[34]	security	All	none	theoretical	0	none	0	no	category	no
This	correctness, performance and security	12[b]	static	case study	5	class	12,857	yes	category, pattern, role	yes

* Only SPP components are considered.

[a] Adapter, Builder, Command, Creator, Factory, Template Method, Proxy, Singleton, State, Strategy, Visitor.

[b] Abstract Factory, Singleton, Composite, Adapter, Command, Observer, State, Strategy, Template Method, Decorator, Prototype and Proxy.

[c] Singleton, Template Method, Decorator, Observer, Factory.

[d] Factory Method, Adapter, Observer.

[e] Abstract Factory, Builder, Factory Method, Prototype, Singleton, Bridge, Composite, Decorator, Flyweight, Proxy, Command, Mediator, Observer, Strategy, Visitor.

- *It identifies similarities and differences between the results obtained by static analysis and those obtained by **dynamic analysis**,* increasing the validity of evidence on the subject, as well as adding to the current state of the art on analysis of runtime QAs;
- It is, to the best of our knowledge, the first study that provides ***empirical evidence*** *on the relationship between the use of GoF patterns and security.*

4.3 Case Study Design

This section describes the case study protocol, which was designed according to the guidelines of Runeson et al. (2012) and is reported based on the Linear Analytic

Structure (Runeson et al., 2012).

4.3.1 Objectives and Research Questions

The goal of this study is described using the Goal-Question-Metrics (GQM) approach (van Solingen et al., 2002), as follows: *"**analyze** software projects **for the purpose of** evaluating GoF design patterns **with respect to** their potential relationship with runtime quality attributes, **from the point of view of** software developers **in the context of** open source systems"*. Based on the goal of this study, we defined the following research questions (RQs):

RQ$_1$: To what extent do runtime QAs differ between non-pattern-participating (NPP), single-pattern-participating (SPP), and coupled-pattern-participating (CPP) classes?

- **RQ$_{1.1}$:** To what extent do the aforementioned groups of classes differ regarding correctness?
- **RQ$_{1.2}$:** To what extent do the aforementioned groups of classes differ regarding performance?
- **RQ$_{1.3}$:** To what extent do the aforementioned groups of classes differ regarding security?

RQ$_1$ aims at exploring whether the application of GoF design patterns is related to the levels of runtime QAs. This question is important to investigate, in the sense that certain GoF patterns are using "expensive" or sometimes complex OO mechanisms, e.g., polymorphism or extensive message passing, that can potentially harm runtime QAs in favor of improving design-time ones. Additionally, while performing such an investigation, it is important to treat the two types of pattern participation (SPP and CPP) separately, because CPP classes are a special case of pattern-participation.

RQ$_2$: Is the relationship between GoF patterns and runtime QAs different across categories of design patterns?

- **RQ$_{2.1}$:** Is there a difference in the levels of correctness among classes participating in patterns of different categories?
- **RQ$_{2.2}$:** Is there a difference in the levels of performance among classes participating in patterns of different categories?
- **RQ$_{2.3}$:** Is there a difference in the levels of security among classes participating in patterns of different categories?

RQ$_2$ aims at investigating if the different purposes that patterns serve (i.e., create objects, handle system behavior, and organize source code structure) lead to a different relation between GoF pattern application and the levels of runtime qualities. A similar question was considered in other studies such as Sahin et al. (2012). To explore this research question, we focus only on SPP classes, clustering them by category (creational, behavioral and structural). We exclude CPP classes from this RQ, since the behavior of coupled pattern cannot be safely attributed to one of the patterns participating in it.

RQ$_3$: Is the relationship between GoF patterns and runtime QAs, different across design patterns?

- **RQ$_{3.1}$:** Is there a difference in the levels of correctness among classes participating in different patterns?
- **RQ$_{3.2}$:** Is there a difference in the levels of performance among classes participating in different patterns?
- **RQ$_{3.3}$:** Is there a difference in the levels of security among classes participating in different patterns?

RQ$_3$ aims at identifying design patterns whose classes might be more prone to violating good coding practices. Therefore, alternative solutions (non-pattern or non-GoF) may be preferred when possible (Ampatzoglou, Charalampidou and Stamelos, 2013a). Many studies (e.g., (Gatrell and Counsell, 2011), (Vokac, 2004) and (Aversano et al., 2009)), have explored similar RQs. To answer this research question, we focus only on SPP classes, clustering them by design pattern.

RQ$_4$: Is the relationship between GoF patterns and the levels of runtime QAs, different across design patterns roles?

- **RQ$_{4.1}$:** Is there a difference in the levels of correctness among classes playing different roles in GoF patterns?
- **RQ$_{4.2}$:** Is there a difference in the levels of performance among classes playing different roles in GoF patterns?
- **RQ$_{4.3}$:** Is there a difference in the levels of security among classes playing different roles in GoF patterns?

RQ$_4$ aims at investigating the different roles that classes can play within a certain pattern (e.g., the Subject in an Observer pattern instance) to identify those that are more prone to harm specific runtime QAs. Although roles have also been explored in other studies, e.g., (Ampatzoglou et al., 2015; Di Penta et al., 2008), we decided not to investigate the roles individually, but rather consider the similar purposes they have (e.g., Container, Containee and Client), namely meta-roles (see Section 4.3.4). We made this decision since meta-roles may lead to more exploitable results as they encompass responsibilities that may lead to more relevant investigation (Ampatzoglou et al., 2015). To explore this research question, we focus only on SPP classes, clustering them by design pattern meta-role.

4.3.2 Case Selection and Unit of Analysis

This study is a holistic multiple-case study, in which open-source software (OSS) projects are the subjects. As unit of analysis we refer to one class of a project over a certain period of time when the number and type of patterns, in which the class participates, is stable. Based on the above, as a stable pattern status period, we refer to a set of versions in which the class did not change its participation status (i.e., participating to a specific pattern, or not participating to any pattern). Therefore, it is important to note that, in order to avoid possible bias from outliers (e.g., an outstanding good release, or a very buggy version), we consider all available versions of each system, considering the measurement of each metric in a unit of analysis as the average of all versions in the stable pattern status period.

Concluding, we consider triplets of *<class_name, pattern_participation, versions-span>* as unit of analysis. For example, suppose that class C1 starts its lifespan as a participant in a Visitor pattern, until version 6; next, it does not participate in a pattern for three versions; and next it becomes a Strategy participant for three more versions. This class would provide us with three units of analysis, as follows:

<C1, Adapter, Ver. 1-Ver. 6>

<C1, no-pattern, Ver. 7-Ver. 9>

<C1, Strategy, Ver. 10-Ver. 12>

In order to select appropriate cases (i.e., subjects) for our study, we considered OSS projects from SourceForge[3]. The projects used in our analysis were required: (a) to be written in Java, due to limitations of the used tools (see Section 4.3.4); (b) to have an adequate number of versions for evolution analysis; and (c) not to be considered as “toy examples”. For that, we selected the five most popular projects from SourceForge that fit our requirements, and we collected all available versions for each one of them. The selected projects, accompanied by their size and duration information, are presented in Table 4.2.

[3] `https://sourceforge.net/`

4.3.3 Variables

In order to answer the research questions stated in Section 4.3.1, we extracted three sets of variables from each class of each version of the five selected projects (to later compute the units of analysis - see Section 4.3.4), as follows:

- project and class identification information;
- pattern participation information; and
- estimates on the levels of QAs.

Table 4.2: Projects considered in the case study

Project name	**Starting year**[a]	**Size**[b]	**NoC**[c]	**NoV**[d]
Bonita BPM	2009	138K	3,994	45
Convertigo	2011	79K	1,779	40
Eclipse Checkstyle	2003	9K	216	37
Hibernate	2001	162K	3,374	126
LogicalDOC	2008	46K	818	31

[a] Year of registration according to SourceForge.
[b] Size in lines of code (of the last version).
[c] NoC = Number of Classes (of the last version).
[d] NoV = Number of Versions.

Details on the pattern detection and runtime QAs assessment are presented in the following sections. We clarify that the third set represents assessments of the studied QAs, and, therefore, when referring to the attributes, we are in practice referring to their assessments. For that, we selected metrics that, to the best of our knowledge, are able to quantify aspects of their levels of quality. An overview and more details on each variable are presented in Section 4.3.4.

Detection of Design Patterns Occurrences

Regarding pattern detection in each version of the projects, we used a tool developed by Tsantalis et al. (2006). This tool[4] uses a Similarity Scoring Algorithm (SSA) for detecting instances of 12 patterns, namely, Adapter/Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State/Strategy, Template Method, and Visitor. By reverse engineering the system under study, this tool isolates subsystems and explores the relationship between elements in each one of

[4] http://users.encs.concordia.ca/~nikolaos/pattern_detection.html

them, applying the proposed SSA to detect occurrences of the aforementioned patterns (Tsantalis et al., 2006). The tool was already evaluated in independent studies (e.g., by Kniesel and Binun (2009), and Pettersson et al. (2010)), which reported positively on its performance, precision and recall rates. In short, the recall of the tool averages around 70%, varying between 25% and 100% in the reported benchmarks. The precision is reported to be close to 100%, mostly due to structure-based detection approach. Moreover, we manually verified the precision of the tool by checking 50 random pattern instances for each GoF pattern that is detected by this tool (i.e., over 500 instances in total—standing for 32% of the total dataset in terms of pattern instances), which were all true positives. We decided to use this tool for the following reasons:

- it covers a fair amount of design patterns that can be detected;
- it has adequate performance, as reported in Tsantalis et al. (2006), also when compared to similar tools (Kniesel and Binun, 2009; Pettersson et al., 2010); and
- it facilitates the pattern detection process.

Although this tool is able to detect the aforementioned patterns, it does not extract all PP classes. According to Aversano et al. (2009) classes are subdivided into two categories: (a) main PP classes, comprising those that provide the structure of the pattern solution (commonly abstract classes); and (b) extended PP classes, which are subclasses of the former that extend the functionality of the pattern solution. The SSA tool only detects the main PP classes. Therefore, as we require all PPs to be identified for our study, a second tool[5] named SSA+, developed by the authors, was used to identify and extract the extended PP classes. SSA+ takes as input the output of SSA, and is able to identify 10 extra roles, based on the information provided by SSA for each pattern occurrence. The extra roles are: Concrete Creator and Product, for Factory Method pattern; Concrete Prototype, for Prototype pattern; Leaf, for Composite pattern; Concrete Decorator and Concrete Component, for Decorator pattern; Concrete Observer, for Observer pattern; Concrete State/Strategy, for State/Strategy pattern; Concrete Class, for Template Method pattern; and Subject, for Proxy pattern.

The task performed by SSA+ is deterministic, as it identifies classes that comply with a set of rules (e.g., inherit from a main PP class), and, therefore, the final version of the tool should present no faulty results, as it was thoroughly tested. In order to further validate SSA+, we manually verified the output for numerous pattern

[5] `https://github.com/search-rug/ssap`

occurrences (randomly selected) of each pattern for which we detect extra roles. In all cases, SSA+ found only true positives, and no class appeared to be missing. The tool was also used in another study, in which a similar verification procedure was performed (Feitosa, Avgeriou, Ampatzoglou and Nakagawa, 2017b). Additionally, to validate our data collection, we verified the frequency of pattern occurrences in the entire dataset against the frequencies obtained by related work (Ampatzoglou et al., 2015; Khomh et al., 2009). In Table 4.3, we present the distribution of classes among all participation types (NPP, SPP and CPP), for the five projects, as well as the summary of all projects. All frequencies are presented considering main roles only (detected via SSA), as well as all roles (SSA+, i.e., main and extra roles). Related work has considered main roles and the frequency reported in Table 4.3 (SSA) is accordance to theirs (Ampatzoglou et al., 2015; Ampatzoglou, Kritikos, Arvanitou, Gortzis, Chatziasimidis and Stamelos, 2011; Khomh et al., 2009). Finally, one can notice that, as expected, the frequency of PP classes considering all roles (SSA+) is higher than considering main roles only (SSA).

Assessment of Runtime Quality Attributes

To evaluate software projects with respect to their runtime QAs, we performed static analysis by collecting the amount of several different types of violations of good coding practices. For that, we used the tool FindBugs[6], which detects such violations and provides warnings (Hovemeyer and Pugh, 2004). The tool was already evaluated in independent studies (e.g., by Hovemeyer and Pugh (2004) and Ayewah et al. (2007)), which reported an average precision of 66% and stated that the precision can be increased by measures such as filtering bug patterns and selecting confidence levels. We also evaluated the tool for the study reported in Chapter 2 and found that its precision for the three levels of confidence (i.e., low, medium, and high) are 26.67%, 60%, and 73.33%, respectively (Feitosa et al., 2015). Thus, we advised discarding violations with low level of confidence.

It is possible that FindBugs introduces noise (i.e., false positives) to the data collection. However, we also found in other studies (Ayewah and Pugh, 2010; Hovemeyer and Pugh, 2004; Khalid et al., 2016; Tripathi and Gupta, 2014; Zheng et al., 2006), as well as in our experiments, that the violations identified by FindBugs are valuable pointers to parts of the system that need to be maintained. Moreover, we note that although part of the violations detected by FindBugs may incur bugs in the system (therefore the nomenclature "bug pattern"), we do not make this assumption. We treat them simply as violations, which can be used as indicators of quality. Other studies explored this approach to estimate quality (Hora et al., 2012;

[6] `http://findbugs.sourceforge.net/`

Table 4.3: Frequency of pattern occurrences based on SSA and SSA+

Project name	Number of classes	Tool	NPP	SPP	CPP
Bonita BPM	3994	SSA	74.0%	19.2%	6.8%
		SSA+	54.0%	35.3%	10.8%
Convertigo	1779	SSA	89.0%	8.2%	2.8%
		SSA+	74.6%	16.1%	9.3%
Eclipse Checkstyle	216	SSA	71.8%	19.9%	8.3%
		SSA+	40.7%	44.9%	14.4%
Hibernate	3374	SSA	63.5%	25.2%	11.4%
		SSA+	51.0%	26.6%	22.4%
LogicalDOC	818	SSA	85.0%	9.5%	5.5%
		SSA+	80.6%	13.0%	6.5%
Total	10181	SSA	74.0%	18.5%	7.5%
		SSA+	58.5%	27.4%	14.1%

The frequency is w.r.t. the last version of each project

Khalid et al., 2016). In particular, Khalid et al. (2016) examined violations from FindBugs, correlating them to software quality as perceived by end-users. Their results suggest that violations can be used as quality indicators, as the two were closely related in the observed population. In short, we adopt the estimation of violations identified by static analysis not as an absolute number of real bugs or faults in the system at a given moment, but as a quality indicator that can warn developers and architects to investigate a given part of the system.

In this case study, we have chosen to use FindBugs because it provides:

- a collection of over 400 bug patterns;
- adequate precision when compared to similar tools (Ayewah et al., 2008, 2007; Hovemeyer and Pugh, 2004), which reflects on the relevance of the offered bug patterns; and
- a grouping of these bug patterns in nine high-level categories[7] that can in turn be mapped into QAs, as presented below.

In this study, to evaluate runtime QAs, we considered the first five categories (in total 246 bug patterns), as they can be mapped to the three studied QAs: *correctness*[8]

[7] The categories are: Security, Correctness, Multithreaded Correctness, Performance, Malicious Code, Bad Practice, Internationalization, Experimental and Dodgy Code

[8] An example of a correctness bug pattern is: Signature declares use of unhashable class in hashed construct (`http://findbugs.sourceforge.net/bugDescriptions.html#HE_SIGNATURE_DECLARES_HASHING_OF_UNHASHABLE_CLASS`)

(*Correctness* and *Multithreaded Correctness* categories), *performance*[9] (*Performance* category), and *security*[10] (*Security* and *Malicious Code* categories). The levels of quality, in each OSS version, for the three aforementioned QAs are estimated by the quantity of:

- ***New violations per lines of code (LoC)***: this metric partially expresses the likelihood of a class to harm the assessed QA;
- ***Removed violations (i.e., not detected in comparison to the previous version) per LoC***: this metric partially expresses the likelihood of a class to benefit the assessed QA. As this metric depends on the previous existence of violations, we represent it as the percentage of removed violations compared to the total amount; and
- ***Total violations (i.e., total number of detected violations) per LoC***: this metric takes into account the resulting effect of both adding and removing violations.

We clarify that, concerning correctness and security, the quantities are the sum of the two categories of bug patterns that each QA is comprised of. For example, security is measured by summing the numbers from both *Security* and *Malicious Code* categories. For all three QAs a lower number of *Total* and *New* violations reflect a higher level of quality, while it is the opposite for *Removed* violations (i.e., the level of quality is directly proportional to the number of *Removed* violations).

4.3.4 Collection Procedure and Pre-processing

The data collection phase was a two-step process. First, we collected raw data of the QAs assessment variables for every class of every version using FindBugs, as well as design pattern related variables using the SSA and SSA+ tools. All tools work on Java binary code, so we fed them with a set of .class files and recorded the outcome. For FindBugs, we used the command line version 3.0.0, for automation purposes. We configured the tool with maximum effort (i.e., enabling analysis that increases precision), and reported violations with medium or high confidence level (to improve precision, as reported in a previous study (Feitosa et al., 2015)) and from all urgency priorities (i.e., from least to most harmful to the system). For SSA tool, we used the command line version 4.5, also for automation purposes.

[9] An example of a performance bug pattern is: Method allocates an object, only to get the class object (`http://findbugs.sourceforge.net/bugDescriptions.html#DM_NEW_FOR_GETCLASS`)

[10] An example of a security bug pattern is: HTTP cookie formed from untrusted input (`http://findbugs.sourceforge.net/bugDescriptions.html#HRS_REQUEST_PARAMETER_TO_COOKIE`)

Next, we derived additional information that was needed to collect every variable for the units of analysis. For that, two tasks were performed:

1. We compiled FindBugs' information for each project. The *bug detection report of all versions* was compiled into a ***history of violations***, generated by FindBugs. The number of violations from each QA was obtained by counting the rule violations with medium and high confidence from FindBugs[11]. The three aforementioned metrics (*New*, *Removed*, and *Total* number of violations) were calculated based on these values.

2. We mapped pattern roles to their respective meta-roles, according to the map presented in Table 4.4. The meta-roles are assigned to roles based on the purpose of those roles. We considered the same seven meta-roles as in our earlier work (for more details see (Ampatzoglou et al., 2015)): Client, Container (a container or aggregate in a "whole-part" relationship, or the dependent class in a "simple association"), Containee (a containee or component in a "whole-part" relationship or the independent class in a "simple association"), Superclass (or abstract class), Subclass, Compound (playing two or more of the aforementioned roles), and Singleton. For example, the Subject acts as a container in the Observer pattern.

Finally, a dataset was created with the information of all variables for each unit of analysis. This dataset was recorded as a table into a spreadsheet, in which each line corresponded to one class of one project in a certain version range. Summarizing, the full list of variables, together with their description, is presented in Table 4.5. The final dataset is created as described above to facilitate the identification of coupled patterns (by detecting duplication of classes in different patterns on the same version), and merging of tuples to build data subsets for answering each RQ (e.g., merging tuples of same pattern category and version, to answer RQ$_2$).

[11] We had analyzed and validated FindBugs, in a previous work (Khalid et al., 2016), regarding its confidence levels, reporting that precision can be improved by excluding bugs with low confidence level.

Table 4.4: Mapping of pattern roles to meta-roles

Pattern Type	Pattern Role	Meta-role
Adapter/Command	Adaptee/Receiver	Containee
	Adapter/Concrete Command	Container
Composite	Component	Superclass
	Composite	Compound
	Leaf	Subclass
Decorator	Component	Superclass
	Concrete Component	Subclass
	Concrete Decorator	Subclass
	Decorator	Compound
Factory Method	Concrete Creator	Compound
	Creator	Superclass
	Product	Containee
Observer	Concrete Observer	Subclass
	Observer	Compound
	Subject	Container
Prototype	Client	Client
	Concrete Prototype	Subclass
	Prototype	Superclass
Proxy	Proxy	Compound
	RealSubject	Subclass
	Subject	Superclass
Singleton	Singleton	Singleton
State/Strategy	Concrete State/Strategy	Subclass
	Context	Client
	State/Strategy	Superclass
Template Method	Abstract Class	Superclass
	Concrete Class	Subclass

Table 4.5: List of collected variables

Variable	Description	Tool
V1	Source project of the class	-
V2	Class full name (package + class name)	-
V3	Versions of the project considered in the unit of analysis	-
V4	Class type (i.e., NPP, SPP, or CPP)	SSA & SSA+ tools
V5	Name of the category (i.e., behavioral, creational, structural) containing the pattern in which the class participated.	
V6	Name of the pattern in which the class participated	
V7	Name of the meta-role that the class played in the pattern	
V8-V10	Violation metrics for correctness	FindBugs
V11-V13	Violation metrics for performance	
V14-V16	Violation metrics for security	

4.3.5 Data Analysis

During this phase, we analyzed the previously described variables (V1–V16) to investigate the relationship between the use of design patterns and the level of runtime QAs. We clarify that, unless specified, when referring to a relationship with a certain QA, we imply the scores of all metrics of the QA. The analysis of the collected data is split in four steps, each aiming at answering each of the four RQs. In each step, a data subset was derived from the final dataset (see Section 4.3.4), and further analyzed as follows:

step 1 *Verify relationship between NPP, SPP, and CPP classes for each QA.* In this step, we derived a data subset consisting of classes that are related to only one of the three participation types during their entire version-span (within the collected data). By avoiding change of participation, we aimed at mitigating bias caused by the joint effect of multiple types of pattern participation. Therefore, independent sample t-tests are performed in order to investigate differences in the level of QAs among the types of participation. This step was divided into four sub-steps.

a. *Select relevant tuples.* We selected only classes that have been only NPP, SPP, or CPP during their lifetime (i.e., classes that did not change their participation).

b. *Remove duplicated entries.* As CPP classes appear in more than one pattern occurrence, it is necessary to avoid accounting for duplicates of classes. For example, a class that has initially been a Singleton and in a later version part of an Adapter occurrence would produce two units of analysis as a pattern participant, which have identical number of violations (because they regard the same class). These two units of analysis would be merged into a single CPP entry (to avoid duplicated entries).

c. *Compute units of analysis.* Tuples concerning the same class in the different versions are now united by calculating the average for each metric, resulting in the data subset to be analyzed.

d. *Calculate differences in levels of QA*s. We performed independent sample t-tests using pattern participation (V4) as a grouping variable, and the number of violations (V8-V16) as test variables. We clarify that, despite analyzing three groups (NPP, SPP, and CPP), we did not perform analysis of variance, since we intend to look into every pairwise comparison.

step 2 *Verify difference among design pattern categories for each metric of each QA.* In this step, we derived a data subset consisting of classes that were SPP during their entire version-span, for the same reasons described in step 1. In a similar four-sub-steps process, the units of analysis were initially selected and clustered into three clusters, one for each pattern category (structural, behavioral, and creational); afterwards, duplicates were removed, i.e., tuples concerning same class, version, and pattern category; then, the units of analysis were computed by merging tuples concerning same classes and different versions; finally, a one-way analysis of variance (ANOVA) was performed for each metric of each QA, i.e., V8–V16, using pattern category (V5) as grouping variable.

step 3 *Verify difference among design patterns for each metric of each QA*. In this step, we derived a data subset consisting of classes that were SPP during their entire version-span, for the same reasons described in step 1. Similar to step 2, we performed ANOVA for each metric of each QA, but using the pattern (V6) as grouping variable. Removal of duplicates and computation of units of analysis were also similarly performed (considering pattern in the procedure).

step 4 *Verify difference among design pattern meta-roles for each metric of each QA*. In this step we derived a data subset consisting of classes that were SPP during their entire version-span, for the same reasons described in step 1. Next, we clustered the dataset into seven groups, one for each meta-role (V7). Finally, we performed independent sample t-tests between pairs of meta-roles for each metric of each QA, i.e., V8–V16.

Summarizing the procedure for answering the RQs, Table 4.6 presents the mapping between each RQ, the used variables, as well as the step of the analysis in which each RQ is answered, along with the used presentation methods.

4.4 Results

In this section, we present the results of the case study, highlighting the most important observations based on the acquired data. Each RQ is addressed separately, presenting an overview of the considered data subset, as well as the statistical analysis over the data (see Section 4.3.5). Before presenting the results, we clarify that the metric on *Removed* violations was not statistically evaluated in all RQs due to the low number of classes that had removed violations. However, this is not an important issue since the metric on *Total* number of violations also reflects the effects of the removed ones (see Section 4.3.3).

Table 4.6: Mapping of RQs to variables, steps, and presentation

Research Question		Used Variables	Step	Presentation Method
RQ1 (relationship between PP and NPP classes)	**RQ1.1** (regarding correctness)	V2-V4: unit of analysis V8-V10: correctness metrics	1	Independent sample t-test
	RQ1.2 (regarding performance)	V2-V4: unit of analysis V11-V13: performance metrics		
	RQ1.3 (regarding security)	V2-V4: unit of analysis V14-V16: security metrics		
RQ2 (difference among design pattern categories)	**RQ2.1** (regarding correctness)	V5: pattern category V8-V10: correctness metrics	2	ANOVA
	RQ2.2 (regarding performance)	V5: pattern category V11-V13: performance metrics		
	RQ2.3 (regarding security)	V5: pattern category V14-V16: security metrics		
RQ3 (difference among design patterns)	**RQ3.1** (regarding correctness)	V6: pattern V8-V10: correctness metrics	3	ANOVA
	RQ3.2 (regarding performance)	V6: pattern V11-V13: performance metrics		
	RQ3.3 (regarding security)	V6: pattern V14-V16: security metrics		
RQ4 (difference among design pattern meta-roles)	**RQ4.1** (regarding correctness)	V7: meta-role V8-V10: correctness metrics	4	Independent sample t-test
	RQ4.2 (regarding performance)	V7: meta-role V11-V13: performance metrics		
	RQ4.3 (regarding security)	V7: meta-role V14-V16: security metrics		

4.4.1 Comparison between SPP, PPC, and NPP classes (RQ_1)

The descriptive statistics (i.e., number of units of analysis, mean number of violations per 10 KLOC, and standard deviation) of the data subset built for answering RQ_1 (see step 1 of Section 4.3.5) are presented in Table 4.7. We clarify that due to the nature of the data (i.e., violations) it is expected that most of the classes in the dataset do not present violations. Thus, metrics such as median and mode would not be descriptive for our dataset and, for that reason, are not included in Table 4.7. For each metric, we highlight the type of pattern participation with the lowest amount of *New* and *Total* violations, e.g., for *New* security violations SPP classes had the lowest average number of violations (i.e., 0.72).

Based on the descriptive statistics, we created the radar chart depicted on Figure 4.1to better visualize and compare the three types of pattern participation. To create the charts, we normalized the mean for each metric (*New* violations, *Removed* vio-

Table 4.7: Descriptive statistics of the data subset for RQ_1

VT	QA	Class Type	N	Mean	SD
New	Security	NPP	7,961	1.39	16.30
		SPP	3,249	0.72	8.58
		CPP	1,647	0.86	11.50
	Correctness	NPP	7,961	0.56	11.50
		SPP	3,249	0.18	2.82
		CPP	1,632	0.45	9.99
	Performance	NPP	7,961	0.74	10.90
		SPP	3,249	0.44	5.70
		CPP	1,647	0.24	2.26
Total	Security	NPP	7,961	19.90	137.00
		SPP	3,249	11.60	81.90
		CPP	1,647	11.20	90.80
	Correctness	NPP	7,961	5.35	50.20
		SPP	3249	2.92	35.10
		CPP	1,643	3.65	29.30
	Performance	NPP	7,961	7.44	57.90
		SPP	3,249	4.52	43.70
		CPP	1,647	8.79	55.60
Removed	Security	NPP	387	2.94	18.60
		SPP	131	19.10	147.00
		CPP	86	13.20	69.20
	Correctness	NPP	197	13.20	47.10
		SPP	59	9.22	37.70
		CPP	87	2.34	7.90
	Performance	NPP	259	10.30	53.20
		SPP	71	11.70	43.00
		CPP	103	4.84	18.00

lations, and *Total* number of violations) as the ratio over the best result. Therefore, the best result has score 1, and the other two scores are equal or less than 1. We note that for *New* and *Total* violations, the ratio is inverse, i.e., more violations are implied by scores closer to 0, and best results (i.e., fewer violations) are denoted with scores close to 1. Regarding *Removed* violations high scores imply best results (more violations are corrected), whereas low scores refer to cases in which only few violations are resolved. In each radar chart, we have created three lines: (a) a continuous red line for NPP classes; (b) a dotted blue line for SPP classes; and (c) a dashed green line for CPP classes. To interpret these charts one needs to check which type of pattern participation has a value equal to 1, and then compare to the rest. For instance, for *New* violations we can observe that SPP classes exhibit the best results for Security and Correctness, whereas CPP for Performance.

In order to verify the previously presented differences, we carried out statistical tests to compare all obtained means. For that, we performed independent t-tests

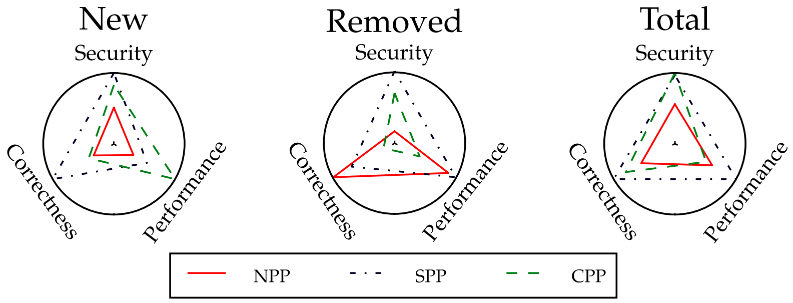


Figure 4.1: Relationship between pattern participation type and QAs

Table 4.8: Statistically significant results from the investigation of RQ_1

VT	QA	Test	Eq. of Variance		Independent T-test		
			F	Sig.	t	df	Sig(2-tailed)
New	Security	NPP vs. SPP	19.13	< 0.01	2.87	10,523.31	< 0.01
	Correctness	NPP vs. SPP	12.96	< 0.01	2.70	9,938.51	< 0.01
	Performance	NPP vs. SPP	8.72	< 0.01	1.93	10,576.89	0.05
		NPP vs. CPP	13.92	< 0.01	3.77	9,606.00	< 0.01
Total	Security	NPP vs. SPP	40.61	< 0.01	3.98	9,729.72	< 0.01
		NPP vs. CPP	23.97	< 0.01	3.20	3,404.78	< 0.01
	Correctness	NPP vs. SPP	24.81	< 0.01	2.91	8,510.88	< 0.01
	Performance	NPP vs. SPP	25.71	< 0.01	2.91	7,912.10	< 0.01
		SPP vs. CPP	30.02	< 0.01	-2.72	2,703.67	< 0.01
Removed	Correctness	NPP vs. SPP	23.97	< 0.01	3.15	219.77	< 0.01

between every two types of participation (i.e., SPP vs. NPP, SPP vs. CPP, and NPP vs. CPP) for each metric of each QA (V8–V16). In Table 4.8 we present the results of the tests that are statistically significant. For example, the difference between NPP and SPP for *New* security bugs presented in Figure 4.1 (top on the left radar chart) is statistically significant. From the results, the following observations can be highlighted.

Non-pattern-participating (NPP) classes are likely to underperform when compared against pattern-participating (PP) classes (both SPP and CPP). For nine out of the 18 possible comparisons (between NPP and the other two class types), NPP classes exhibit a statistically significant larger number of violations than classes participating in patterns. *At the same time, there is no strong statistical evidence of the difference between SPP and CPP classes.* However, it should be pointed out that one difference

Table 4.9: Descriptive statistics of the data subset for RQ_2

VT	QA	Pattern Category	N	Mean	SD
New	Security	Behavioral	1,830	0.87	7.37
		Creational	1,082	0.40	4.10
		Structural	50	8.08	49.60
	Correctness	Behavioral	1,830	0.25	3.40
		Creational	1,082	0.10	2.11
		Structural	50	0.09	0.61
	Performance	Behavioral	1,830	0.63	6.93
		Creational	1,082	0.23	3.70
		Structural	50	0.30	1.60
Total	Security	Behavioral	1,830	14.90	92.30
		Creational	1,082	7.96	67.70
		Structural	50	42.20	161.00
	Correctness	Behavioral	1,830	4.90	51.60
		Creational	1,082	1.17	15.80
		Structural	50	3.30	23.30
	Performance	Behavioral	1,829	7.12	56.50
		Creational	1,082	2.58	33.50
		Structural	50	5.66	34.10

between SPP and CPP classes has been found to be statistically significant (i.e., *Total* amount of performance violations), providing an indication in favor of SPP classes.

4.4.2 Comparison between pattern categories RQ_2

To compare the different pattern categories (i.e., Behavioral, Creational and Structural), we considered the data subset as described in step 2 of the data analysis (see Section 4.3.5), comprising units of analysis limited to SPP classes. The descriptive statistics for this dataset are presented in Table 4.9. Similar to RQ_1, we present the number of units of analysis, mean number of violations per 10 KLOC, and standard deviation for each category (i.e., in this case, of each pattern category). To better visualize the descriptive data, we created a radar chart to ease the comparison of mean values (see Figure 4.2). Similar to Figure 4.1, the best result has score 1, and the other two scores are equal or less than 1.

To verify the differences presented by the descriptive statistics, we performed a two-step statistical analysis for each metric of each QA. First, we carried out an analysis of variance (ANOVA) to identify the existence of differences between the three pattern categories for the given metric. If a difference was detected, then we applied a post-hoc test to recognize which pattern categories differentiate from each other. For the post-hoc test, we used the Bonferroni correction due to its ability to control Type I error (i.e., find a relationship that in fact does not exist), and its suitability for a small number of categories (Field, 2013). In Table 4.10, we present the tests that revealed a statistically significant difference. From the results, we highlight the

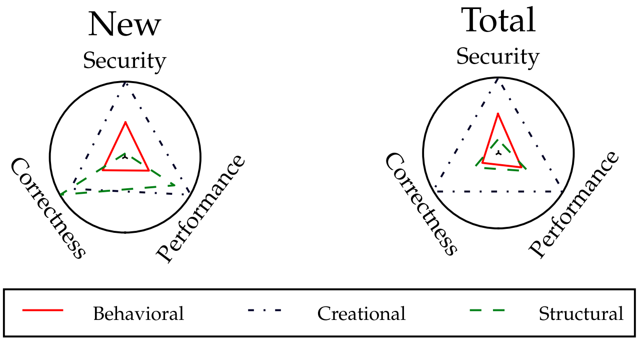


Figure 4.2: Relationship between pattern categories and QAs

Table 4.10: Statistically significant results from the investigation of RQ_2

VT	QA	ANOVA		Post-hoc test (Bonferroni)	
		F	Sig.	Test	Sig.
New	Security	17.56	< 0.01	Structural vs. Behavioral	< 0.01
				Structural vs. Creational	< 0.01
Total	Security	5.20	< 0.01	Structural vs. Creational	0.01
	Performance	2.92	0.05	Behavioral vs. Creational	0.05

following observations.

Classes that participate in Creational patterns are likely to have fewer violations of security and performance coding practices than those participating in Structural and Behavioral patterns. Based on the statistically significant differences presented in Table 4.10, we notice that comparisons of Creational patterns regarding security are mostly valid. With regards to performance violation, the only statistically significant different provides an indication that Creational patterns may be less prone to such violations. This is an expected result as Creational patterns tend to be simpler than patterns of the other two categories. Moreover, this finding corroborates with findings of related work.

4.4.3 Comparison between patterns (RQ_3)

For comparing the 12 patterns considered in this study, we focused on SPP classes as units of analysis. However, this time we clustered them by pattern, so as to be able to explore the differences between such types for each QA metric. Similar to the previous RQs we present the descriptive statistics of this data subset (see Table

4.11). This table shows the number of units of analysis for each pattern, as well as the mean and standard deviation for each QA metric. It is important to highlight that we excluded three patterns from the analysis (Composite, Observer and Proxy) due to the small number of available SPP classes (1, 15 and 11, respectively). Additionally, we did not consider results with mean violations equal to zero, although they are shown in Table 4.11 (for clarity purposes). Such a mean indicates that we identified no violations in the SPP classes and, although we expect this number to be small, we cannot predict it with enough confidence.

Table 4.11: Descriptive statistics of the data subset for RQ_3

VT	QA	Pattern	N	Mean	SD
New	Security	Adapter/Command	371	0.98	6.22
		Decorator	38	10.10	56.80
		Factory Method	602	0.20	3.30
		Prototype	45	0.00	0.00
		Singleton	435	0.72	5.17
		State/Strategy	1,020	0.67	7.31
		Template Method	797	1.13	7.51
	Correctness	Adapter/Command	371	0.17	1.24
		Decorator	38	0.11	0.70
		Factory Method	602	0.00	0.06
		Prototype	45	0.00	0.00
		Singleton	435	0.25	3.32
		State/Strategy	1,020	0.22	2.02
		Template Method	797	0.29	4.62
	Performance	Adapter/Command	371	0.23	3.31
		Decorator	38	0.40	1.83
		Factory Method	602	0.12	2.67
		Prototype	45	0.26	1.24
		Singleton	435	0.37	4.90
		State/Strategy	1,020	0.52	6.46
		Template Method	797	0.79	7.54
Total	Security	Adapter/Command	371	16.40	81.50
		Decorator	38	50.40	182.00
		Factory Method	602	1.94	30.10
		Prototype	45	3.28	16.00
		Singleton	435	16.80	100.00
		State/Strategy	1,020	12.20	90.00
		Template Method	797	18.60	95.80
	Correctness	Adapter/Command	371	3.20	20.80
		Decorator	38	4.34	26.80
		Factory Method	602	0.11	1.95
		Prototype	45	3.34	22.40
		Singleton	435	2.42	23.70
		State/Strategy	1,020	3.52	38.60
		Template Method	797	6.74	64.80
	Performance	Adapter/Command	371	2.75	19.30
		Decorator	38	7.45	39.10
		Factory Method	602	1.96	39.10
		Prototype	45	14.80	56.10
		Singleton	435	2.18	18.80
		State/Strategy	1,019	3.78	35.80
		Template Method	797	11.50	75.10

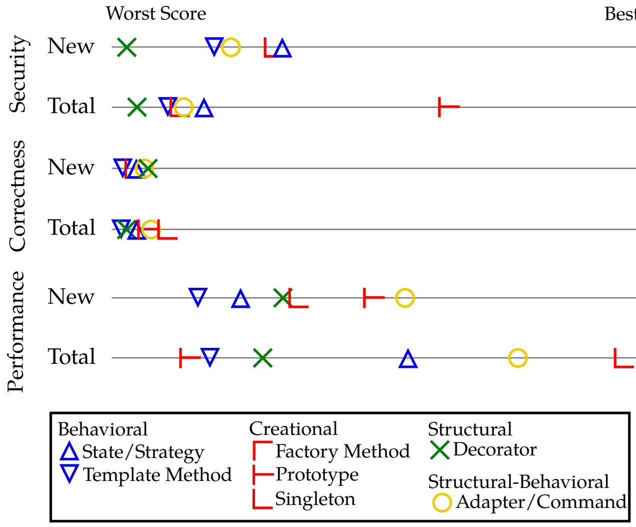


Figure 4.3: Relationship between patterns and QAs

To better visualize the difference between the means and, most importantly, how the patterns are ordered, we plotted each metric in different rows of Figure 4.3. Each row presents the mean number of violations, normalized by the best score, i.e., the best score received 1 (plotted on the rightmost side) and all others a ratio of this value factor (patterns with score 0 are not plotted). The patterns are identified by symbol (see legend of the figure), and each pattern category (e.g., Creational) has a different filling color and graphic pattern. It is important to highlight that the Adapter/Command patterns are detected jointly by SSA+ (due to design similarities), and that they are from different categories (Structural and Behavioral respectively). This is considered in both charts and analysis of the results.

Similar to the previous RQ, we used ANOVA to verify the differences among the means. However, for the post-hoc test, we selected Games-Howell because the number of groups (i.e., patterns) was inadequate for using the Bonferroni correction. Table 4.12 presents the statistically significant results. From the results, we highlight the following observations.

SPP classes of a Factory Method instance are less prone to violations. This finding is supported by the facts that six comparisons involving Factory Method are statistically significant (see Table 4.12). This result is not surprising because this is a Creational pattern, which has previously shown to be the least vulnerable one. *Ordering of patterns tends to reflect the ordering of pattern categories* (see Section 4.4.2), as it would be expected. By looking at Figure 4.3, one can see that Creational patterns

Table 4.12: Statistically significant results from the investigation of RQ_3

VT	QA	ANOVA		Post-hoc test (Games-Howell)	
		F	Sig.	Test	Sig.
New	Security	5.92	< 0.01	Adapter/Command vs. Prototype	0.04
				Factory Method vs. Template Method	0.03
				Singleton vs. State/Strategy	0.059
Total	Security	3.72	< 0.01	Adapter-Command vs. Factory Method	0.02
				Factory Method vs. Singleton	0.04
				Factory Method vs. State/Strategy	0.02
				Factory Method vs. Template Method	< 0.01
				Prototype vs. Template Method	< 0.01
	Performance	4.21	< 0.01	Adapter/Command vs. Template Method	0.04
				Factory Method vs. Template Method	0.04
				Singleton vs. Template Method	0.02

are predominantly ranked among those with best scores, whereas Behavioral and Structural patterns interchangeably rank among those with worst scores. For example, Factory Method, which is a Creational pattern, achieves the highest scores, while Template Method (a Behavioral pattern) and Decorator (a Structural pattern) have the worst scores. However, no clear ordering appears within the patterns of a category (expect for Factory Method). This might be evidence that the amount of violations might be more related to the type of responsibility (identified by a pattern category) rather than to specific patterns.

4.4.4 Comparison between pattern roles (RQ_4)

As shown in Table 4.4, there are 30 pattern roles distributed among the 12 patterns considered in this study. To study these roles, we decided to consider the meta-roles (see Table 4.4) rather than the roles themselves, as we expect the amount of violations to be related to the type of responsibility a role has. This reduces the number of groups to be analyzed to the seven meta-roles presented in Table 4.4.

By focusing the investigation to the meta-roles that classes play in a pattern instance, we analyze the relationship between the types of roles these classes have. Therefore, for RQ_4 we investigated only the combinations of roles that participate within the same pattern (i.e., meta-roles that collaborate to provide a specific pattern solution). For example, in the instance of a Prototype pattern, classes of the

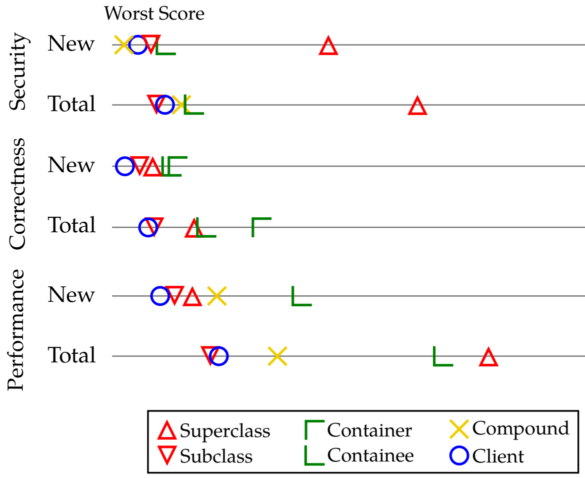


Figure 4.4: Relationship between meta-roles and QAs

following meta-roles are present: Client, Superclass and Subclass. The investigation of these meta-roles was performed in pairs, as follows: Client vs. Superclass; Client vs. Subclass; and Superclass vs. Subclass. Considering all patterns, we derived a total of 19 pairs; we excluded the Singleton meta-role because it involves one class only.

The descriptive statistics of the analyzed meta-roles are presented in Table 4.13, showing the number of units of analysis, mean violations per 10 KLOC and standard deviation for each QA metric. In contrast to the previous RQs, we do not highlight the best means, as the comparisons are at the level of pairs of meta-roles rather than overall. To better visualize the comparison between the means, we created eight plots, grouping the means by metric and type of violation. Figure 4.4 presents these charts; they are read similarly to Figure 4.3 and colors represent sets of related meta-roles (e.g., Subclass and Superclass).

For statistically analyzing the difference between meta-roles in each pair, we performed independent sample t-tests. The tests that showed statistically significant difference are presented in Table 4.14. Based on the results, the most important observation is that *more generic meta-roles (i.e., Container and Superclass) are less prone to violations than less generic meta-roles*. This is an intuitive result because more abstract elements of a design usually have less complex logic, being supposedly simpler to implement. Additionally, there are several statistically significant differences that show the clear difference between the meta-roles.

Table 4.13: Descriptive statistics of the data subset for RQ_4

VT	QA	Meta-role	N	Mean	SD
New	Security	Client	461	1.26	10.50
		Compound	203	2.43	25.30
		Containee	470	0.77	5.54
		Container	152	0.04	0.53
		Subclass	1,382	0.90	7.05
		Superclass	459	0.11	1.05
	Correctness	Client	461	0.34	2.56
		Compound	203	0.01	0.10
		Containee	470	0.10	0.98
		Container	152	0.09	0.88
		Subclass	1,382	0.19	3.51
		Superclass	459	0.13	1.71
	Performance	Client	461	0.68	6.43
		Compound	203	0.32	4.58
		Containee	470	0.17	2.94
		Container	152	0.06	0.54
		Subclass	1,382	0.56	5.93
		Superclass	459	0.41	7.13
Total	Security	Client	461	15.40	77.20
		Compound	203	13.00	93.70
		Containee	470	12.80	72.40
		Container	152	1.36	16.70
		Subclass	1,382	17.40	103.00
		Superclass	459	2.54	24.20
	Correctness	Client	461	5.29	41.70
		Compound	203	0.32	3.35
		Containee	470	2.13	17.80
		Container	152	1.22	9.28
		Subclass	1,382	4.40	49.90
		Superclass	459	2.16	39.20
	Performance	Client	461	7.44	51.50
		Compound	203	4.55	64.80
		Containee	470	2.29	19.60
		Container	152	1.32	12.60
		Subclass	1,381	7.61	58.30
		Superclass	459	2.01	25.80

4.5 Discussion

In this section, we discuss the main outcomes of the study, providing more details on their interpretation, as well as implications for researchers and practitioners. Comparison to related work is also presented, when applicable.

4.5.1 Interpretation of results

Firstly, while analyzing the results of RQ_1, one can notice that NPP classes are likely to present more violations, regarding runtime QAs, than PP classes (i.e., SPP and CPP). Additionally, SPP classes are more likely to have fewer violations than CPP (we observed an average of 22%). A possible explanation is that PP classes are easier to understand as the patterns also serve as an explicit design documentation and as

Table 4.14: Statistically significant results from the investigation of RQ$_4$

VT	QA	Test	Eq. of Variance		Independent T-test		
			F	Sig.	t	df	Sig(2-tailed)
New	Security	Containee vs. Container	10.56	< 0.01	2.82	494.24	0.01
		Containee vs. Superclass	25.14	< 0.01	2.54	503.48	0.01
		Client vs. Superclass	21.14	< 0.01	2.34	469.33	0.02
		Container vs. Subclass	8.97	< 0.01	-4.42	1,489.28	< 0.01
		Subclass vs. Superclass	22.63	< 0.01	4.03	1,551.12	< 0.01
	Correctness	Compound vs. Containee	7.96	0.01	-2.12	491.92	0.04
	Performance	Container vs. Subclass	4.28	0.04	-3.03	1,519.27	< 0.01
Total	Security	Containee vs. Container	15.33	< 0.01	3.18	586.69	< 0.01
		Containee vs. Superclass	33.82	< 0.01	2.91	574.87	< 0.01
		Client vs. Superclass	45.97	< 0.01	3.41	550.09	< 0.01
		Container vs. Subclass	14.77	< 0.01	-5.21	1,390.06	< 0.01
		Subclass vs. Superclass	37.49	< 0.01	4.97	1,735.38	< 0.01
	Correctness	Compound vs. Subclass	5.40	0.02	-3.00	1457.75	< 0.01
		Compound vs. Containee	8.39	< 0.01	-2.13	540.98	0.03
	Performance	Client vs. Superclass	16.05	< 0.01	2.03	677.47	0.04
		Container vs. Subclass	7.04	0.01	-3.36	1060.25	< 0.01
		Subclass vs. Superclass	15.76	< 0.01	2.84	1704.70	0.01

common language among the team members (Ahlgren and Markkula, 2005; Riehle, 2011). Therefore, it is expected to be easier to understand and maintain a piece of source code when it is using a pattern. On the other hand, adding an extra responsibility to a class (i.e., by making it participate in more than one pattern) might decrease the readability and understandability of the code.

Concerning security, our findings comply with the results of the literature. For example, Ferraz et al. (2009) suggest that GoF design patterns support implementing security requirements. Regarding correctness, Gatrell and Counsell (2011) found that PP classes are more fault-prone than NPP classes. Conversely, we observed that PP classes exhibited fewer violations than NPP classes. However, Gatrell and Counsell also observed that their finding was attributed mainly to a tendency of PP classes to be more change-prone. Thus, a normalization of the results could have shown a finding similar to ours, in which the analysis procedure included normalization. Moreover, Ampatzoglou, Kritikos, Arvanitou, Gortzis, Chatziasimidis and Stamelos (2011), found the overall number of design pattern instances not to be correlated with defect frequency. Runtime defects observed for PP classes are probably related to the complexity of the requirements that pattern instances are involved in, since design patterns are expected to be placed in design hot spots.

Concerning performance, related work have found that PP classes perform worse at times (Afacan, 2011; Litke et al., 2007; Sahin et al., 2012), whereas we observed that PP classes display fewer violations. However, related work also observed that pattern instances could also show improved performance compared to alternative (non-pattern) solutions (Sahin et al., 2012), also suggesting that the usage of design pattern do not necessarily result in change of runtime performance (Litke et al., 2007). The energy consumption and/or CPU usage of PP classes can be higher than that of NPP classes because patterns rely on certain object-oriented (OO mechanisms (e.g., polymorphism) that have higher computational cost, but such drawback is not always observed (Feitosa, Alders, Ampatzoglou, Avgeriou and Nakagawa, 2017). To further study this matter, in a previous study (Feitosa, Alders, Ampatzoglou, Avgeriou and Nakagawa, 2017), we investigated parameters that can influence the efficiency of patterns solutions compared against alternative (non-pattern) solutions. We found that the runtime benefits of a pattern can be associated with its application in more appropriate scenarios, e.g., when the implementation logic is complex in terms of size or messaging. The appropriate use can greatly reduce the overhead of OO mechanisms.

Following our RQs, by analyzing the findings of RQ_2 one can observe that Creational patterns tend to have the lowest number of violations in most cases. An explanation could be that Creational patterns are implemented with a simple source code structure. A simple implementation naturally supports better understanding and readability of the source code, which in turn would explain the existence of fewer violations. In contrast, Structural patterns had the highest frequency of violations. A possible explanation is that Structural patterns are commonly used to organize complex concepts in an OO design, aiming at reducing the accidental complexity of the software (Brooks Jr., 1987) (i.e., the complexity imposed by the designer and not the functionality responsibility). However, even if they decrease the accidental complexity, the essential complexity (i.e., the complexity inherent to the implemented functionality) of these components is still high, leading to more violations when compared to simpler designs (e.g., Creational patterns). We also noticed that the most recurrent violations are common among all pattern categories (see Table B.3 on Appendix B), and are similar to those regarding all SPP classes (see Table B.2 in Appendix B).

We note that the findings of this study are based on violations concerning runtime qualities derived by a static analysis tool. To facilitate the comparison of observations on the difference between PP and NPP classes and on the violations exhibited by Creational patterns, we summarize in Table 4.15 our key findings (static analysis column) versus findings derived by dynamic analysis in previous studies (Afacan, 2011; Ampatzoglou, Kritikos, Arvanitou, Gortzis, Chatziasimidis and Stamelos, 2011; Feitosa, Alders, Ampatzoglou, Avgeriou and Nakagawa, 2017; Gatrell and Counsell, 2011; Litke et al., 2007; Sahin et al., 2012). This summary indicates that the results from static analysis are to a large extent aligned with those from dynamic analysis, with few exceptions as discussed in the preceding paragraphs.

Findings regarding RQ_3 suggest that the ordering of the patterns (from best to

Table 4.15: Comparable observations between static and dynamic analyses

Topic	Observation	
	Results from Static Analysis (this study)	**Results from Dynamic Analysis**
PP vs. NPP classes	Suggest that NPP classes are more likely to underperform.	Show cases in which PP classes underperform and cases in which NPP classes underperform.
	Results may be attributed to the promotion of best practices.	Results may depend on consideration of good design practices. Proper use of patterns (e.g., to organize complex logic) may virtually remove the gap between pattern and non-pattern solutions.
	Suggest that violations are more often removed from PP classes than NPP classes, and that the number of violations in PP classes are lower.	Suggest that PP classes are likely to be more change-prone, which reflect on higher error-proneness.
Creational patterns	Suggest that instances of Factory Method are likely to exhibit better scores than other patterns.	Suggest that Factory Method is among patterns with the best performance.
	Suggest that creational patterns (e.g., Factory Method) are likely to exhibit better scores than patterns of other categories.	Creational patterns (e.g., Factory Method and Prototype) appear among patterns with the best performance.

worst score) tends to follow the ordering of the findings from the pattern categories (RQ_2). This may suggest that, regardless of the QA, the type of the responsibility (defined by the category) plays a major role and, thus, the category has influence on the result. For example, a Creational pattern such as Factory Method presented fewer violations in all cases (in average, 48% fewer than the second-ranked pattern). A plausible explanation is the structural simplicity of this pattern. From the studied Creational patterns, Factory Method is potentially the simplest one, because all other patterns allow more complex implementations (e.g., the cloning mechanism of Prototype and the unique instantiation of Singleton). This finding is in accordance with the related work. For both correctness and performance, studies showed that Factory Method is among the patterns with best scores (Gatrell and Counsell, 2011; Sahin et al., 2012; Vokac, 2004).

Furthermore, we notice that the difference in the patterns' definition and purpose may also reflect on the violations that are accumulated on their instances. By examining Table B.4 (Appendix B), we observe that despite similarities, some of the most recurrent violations differ among the patterns. For example, unsafe multi-threaded calls are more recurrent for four out of the 12 analyzed patterns. Another interesting observation is that instances of some patterns tend to accumulate violations of higher severity ("scary" or "scariest", according to FindBugs classification). In particular, although instances of Factory Method accumulate fewer violations of correctness, the most recurrent ones are potentially more harmful to the system (see Table B.4). This shows that even with a tendency of accumulating fewer violations, it might be important to monitor instances of certain patterns.

Finally, the findings from RQ_4 suggest that classes playing more generic roles (i.e., Superclass and Container) tend to show better scores than classes playing more concrete ones (i.e., Subclass and Containee). A possible explanation for these observations is that more specialized roles implement more intense and complex business logic and, therefore, are more prone to violations. The counting presented in Table B.6 (Appendix B) supports this hypothesis, as one can see that more specialized roles showed larger number of violations with higher severity. Moreover, the type of violation may also be different depending on the role a class plays (see Table B.5 on Appendix B). These observations may also be partially related to the likelihood of a class to be changed. It is intuitive to expect that the more frequently a class is changed the more violations are introduced into its source code. Di Penta et al. (2008) investigated the correlation of pattern roles to the changes in pattern participants. Their findings suggest that some meta-roles (i.e., Subclass and Client) changed more frequently than the other meta-roles). This may indicate that change- and violation-proneness are related.

4.5.2 Implications for practitioners and researchers

The findings of this study suggest that PP classes are likely to have fewer security, correctness and performance violations than NPP classes, even in cases in which a class participates in more than one pattern. Stated differently, this exploratory study revealed that adhering to good architectural practices (such as the use of patterns) is often accompanied by (or due to) better programming practices, leading to fewer violations. One could argue that a well-designed architecture besides its obvious benefits in supporting software evolution provides a solid basis for developing cleaner code with fewer violations. Building an application around reusable, documented and well-tested pattern instances improves comprehensibility, thereby limiting the possibilities of accidentally introducing violations. Moreover, the clean code structure facilitates easier bug localization and removal.

This study has two main implications to researchers. The exploitation of static analysis and, in particular, source code for investigating runtime QAs can add to the current state of the art on the relationship between the presence of patterns and security, correctness and performance. The comparison of our results to related work has in some cases led to contradictions, due to the different nature of our measurement approach, but some common implications can be retrieved. First, the fact that no universal assessment on the relationship between patterns and runtime qualities can be made (without performing a separate investigation per pattern type) is a common finding in both our study and almost every related work in the field. Second, the fact that the structural complexity of the pattern is playing an important role on the relationship of patterns and runtime qualities is confirmed by our case study (e.g., Creational patterns present fewer violations).

Furthermore, to the best of our knowledge, this study is the first to provide empirical evidence on the relationship between GoF patterns and security. In addition, it is interesting to notice that most of the statistically significant results were w.r.t. security. Therefore, we suggest the usage of static analysis when investigating this QA. Finally, the investigation of pattern roles and, in particular, meta-roles has provided interesting and insightful findings to our study, showing to be a valuable source of information when it comes to investigate GoF patterns. The investigation of roles allows a finer-grained analysis of the patterns, while considering meta-roles brings the discussion to a more abstract level, considering characteristics as mechanisms used by the patterns. Thus, we also encourage the consideration of meta-roles when investigating GoF patterns, especially if such characteristics are clearly relevant for interpreting results of the study.

4.6 Threats to Validity

In this section, we present and discuss the threats to the validity of our study, in particular, construct validity, reliability and external validity. Internal validity is not applicable, as the study does not examine causal relations. Construct validity reflects the connection between the object of study, or studied phenomenon, and the RQs. Reliability is related to the possibility of others replicating the performed case study and obtaining the same results. Finally, external validity comprises possible threats to the generalization of the findings on this study to the entire population.

Concerning construct validity, it can be argued that static analysis does not assess runtime qualities as effectively and precisely as dynamic analysis. Indeed, dynamic analysis has been used much more extensively than static analysis in assessing runtime qualities and such results are more well-established. To partially mitigate this threat, we compared some of our results with those from studies using dynamic analysis. The comparison indicates that the results from static analysis are to a large extent aligned with those from dynamic analysis (with some exceptions discussed in Section 4.5.1), so we consider this threat to some extent addressed. Another threat related to construct validity is that the SSA tool is limited by its precision and recall: false positives and negatives may bias the presented results. However, to the best of our knowledge, the used tool is among the most reputed in the community, and has adequate performance (see Section 4.3.3). For mitigating this threat, we manually verified its precision and recall by checking 50 random pattern instances for each

GoF pattern that is detected by SSA tool (i.e., over 500 instances in total—standing for 32% of the total dataset in terms of pattern instances), which were true positives. We note that the level of agreement between the researchers was approximately 98%, since only for very few instances there was an initial disagreement that was resolved through discussion among the two of the authors. Additionally, regarding FindBugs, we acknowledge that the list of bug patterns is by no means exhaustive and additional bugs related to security, correctness and performance could be used. However, to the best of our knowledge this tool is also among the most reputed in the community, and has adequate performance (see Section 4.3.3).

Finally, this study assumes that PP classes in the dataset contribute to pattern instances that are correctly implemented. A pattern implementation might not be the correct one, due to either a programming mistake or pattern grime (Izurieta and Bieman, 2013), or because the deployed pattern is not the optimal way to solve the underlying problem considering that the effect of a pattern on quality is affected by different factors (Ampatzoglou et al., 2012). This poses a threat to construct validity. To mitigate it, we checked all the manually verified pattern instances; we found that their implementation was correct and they were a suitable solution for the problem at hand. Since these instances account for 32% of the dataset, we consider that this threat is to a large extent mitigated.

In order to mitigate reliability, two different researchers were involved in the data collection procedure, double-checking all outputs. Furthermore, the same researchers also double-checked the data analysis. Finally, all primitive data can be reproduced by using the same cases and tooling. The pattern detection tool (SSA v4.5) and bug detection tool (FindBugs, v3.0.0) were downloaded from the provided sources, while we have made the tool developed by us (SSA+ v1.0) publicly available.

Finally, concerning external validity, we identified the following threats. First, not all parts of the architecture can benefit from the introduction of patterns, as the pattern goal might be irrelevant to the functionality/design of that part. Thus, the outcomes of this study do not generalize to all parts of the codebase or the design space, but only to those where the use of a design pattern would be beneficial and applicable. Second, we investigated a limited number of OSS projects. However, the five projects selected are the most popular projects in SourceForge that fitted our selection criteria. Additionally, they vary in terms of both domains and characteristics; this partially alleviates this threat. Next, we investigate a limited number of patterns, as well as pattern instances. A larger sample could strengthen the results and increase our confidence on generalizing our findings. Similarly, we investigated a subset of all runtime QAs and, therefore, our results cannot be generalized to all QAs without further investigation. Finally, we investigated OSS projects written in Java only, while all used tooling was Java-specific. Therefore, our observations focus on this programming language and cannot be generalized to other OO languages without further investigation.

4.7 Conclusion

In this chapter, we investigated the relationship of 12 GoF patterns to three runtime QAs, namely security, correctness and performance. In particular, we conducted a case study on multiple versions of five OSSs among the most popular Java projects on SourceForge platform (Bonita BPM, Convertigo, Eclipse Checkstyle Plug-in, Hibernate and Logical DOC), from which we collected 12,857 classes, being approx. 25% single-pattern participants, 13% coupled-pattern participants and 62% non-pattern participants.

To investigate the relationship between GoF patterns and the aforementioned QAs, we explored source code violations, defining three metrics for each QA, namely the number of *New* violations, *Removed* violations and *Total* number of violations. Considering these metrics, we estimated the levels of the three QAs for each collected class of each version. We investigated the relation of these metrics to GoF patterns with regards to four different perspectives: pattern participation (i.e., NPP, SPP or CPP), pattern category, pattern and meta-role. Results of the study suggest that classes not participating in any pattern are more prone to violations, as well as that participation in more than one pattern can also be connected to the existence of more violations. In addition, classes participating in Creational patterns, especially Factory Method, or playing more generic meta-roles (e.g., Container) are likely to have fewer violations than other classes. However, we advise being attentive of these violations as we found them to often be of higher severity.

The findings of this study, although they do not imply any causality between the introduction of patterns and the removal of violations, provide evidence that code residing around design patterns adheres to good programming practices. This observation is consistent with the wide-spread belief that the application of patterns complies with the adoption of software architecture principles. To complement the findings from this study, the next chapter aims at investigating the impact of GoF patterns on one performance indicator that has recently attracted the attention of both researchers and practitioners, i.e., energy consumption, through a controlled experiment.

Based on:

Daniel Feitosa, Rutger Alders, Apostolos Ampatzoglou, Paris Avgeriou, Elisa Y. Nakagawa, (2017) *"Investigating the Effect of Design Patterns on Energy Consumption,"* in Journal of Software: Evolution and Process, vol. 29, no. 2, e1851, DOI:10.1002/smr.1851

Chapter 5

Investigating the Effect of Design Patterns on Energy Consumption

Abstract

GoF patterns are well-known best practices for the design of object-oriented systems. In this chapter, we aim at empirically assessing their relationship to energy consumption, i.e., a performance indicator that has recently attracted the attention of both researchers and practitioners. To achieve this goal, we investigate pattern-participating methods (i.e., those that play a role within the pattern) and compare their energy consumption to the consumption of functionally equivalent alternative (non-pattern) solutions. We obtained the alternative solution by refactoring the pattern instances using well-known transformations (e.g., replace polymorphism with conditional statements). The comparison is performed on 169 methods of two GoF patterns (namely State/Strategy and Template Method), retrieved from two well-known open source projects. The results suggest that for the majority of cases the alternative design excels in terms of energy consumption. However, in some cases (e.g., when the method is large in size or invokes many methods) the pattern solution presents similar or lower energy consumption. The outcome of our study can be useful to both researchers and practitioners, since we: (a) provide evidence on a possible negative effect of GoF patterns, and (b) can provide guidance on which cases the use of the pattern is not hurting energy consumption.

5.1 Introduction

There has been an increase of energy demand within the ICT domain (Procaccianti et al., 2015). This is a multi-faceted problem, as one can consider the effects of networks, hardware, drivers, operating systems, and applications on energy consumption. In this chapter, we focus on applications and, particularly, how they can be optimized in terms of energy consumption. Software optimizations in this context have been discussed at three levels of granularity:

- ***at architectural level***, e.g., research that deals with energy efficient architectures for networked systems (e.g., data centers, cloud computing, etc.) (Hammadi and Mhamdi, 2014; Procaccianti et al., 2015; Zhang et al., 2016).
- ***at design level***, e.g., identification of differences in energy efficiency when applying design patterns (cf. Section 5.2).
- ***at source code level***, discussions on topics such as multi-threading (Liu, 2012; Pinto et al., 2014), refactoring (Johann et al., 2012; Perez-Castillo and Piattini, 2014; Sahin et al., 2014; Tiwari et al., 1996) and related algorithms (Jain et al., 2005; Noureddine et al., 2012b,a, 2015).

The scope of our work lies at the design level, as we *look into the effect of GoF (Gang of Four) design patterns and their alternative solutions on software energy consumption*. GoF design patterns are recurring solutions to common problems in object-oriented software design (Gamma et al., 1995). GoF design patterns can be applied in almost any type of software, varying from small devices to large data-centers. In Java applications it has been reported that up to 30% of system classes participate in one or more GoF design pattern occurrences (Ampatzoglou et al., 2015; Khomh et al., 2009), leading to a significant influence on overall energy consumption. Solutions provided by these patterns exploit object-orientation mechanisms (e.g., polymorphism) to enforce more flexible and maintainable designs.

The effect of applying a pattern is not uniform across all of its instances, and all quality attributes (Ampatzoglou, Charalampidou and Stamelos, 2013b). In particular, several studies (Ampatzoglou, Charalampidou and Stamelos, 2013b; Hsueh et al., 2008; Huston, 2001) report that the effect of a pattern on a quality attribute depends on certain pattern-related parameters, like the number of classes, number of methods invoked, or number of polymorphic methods. Therefore, it is reasonable to expect that GoF design patterns have a potential impact (positive or negative) on the energy consumption of software-intensive systems, depending on certain pattern-related parameters. In the case where a pattern is not the optimal design solution, alternative (non-pattern) design solutions can be employed. Alternative design solutions have been proposed by several authors, including GoF design pattern advocates (Adamczyk, 2004; Fowler et al., 1999; Gamma et al., 1995; Lyardet, 1997; Victório et al., 2010). More details on GoF design pattern alternatives can be found in a recent literature review (Ampatzoglou, Charalampidou and Stamelos, 2013a). We note that knowing the impact of patterns on energy efficiency can be beneficial in both green- and brown-field software development. In Greenfield projects (i.e., fresh development), such a knowledge can support the monitoring of energy efficiency, whereas in Brownfield projects (e.g., refactoring of system to new purpose), it can support the decision making process on what parts of the system to refactor and how.

In this chapter, we investigate the effect of GoF patterns and their alternatives on energy consumption, as well as the pattern-related parameters that might influence this effect. Specifically, we focus on two GoF design patterns, namely Template Method, and State/Strategy (Gamma et al., 1995); we note that State and Strategy patterns have a similar structure (Tsantalis et al., 2006) and, therefore, a similar expected effect on energy consumption. Therefore, the two patterns are discussed as one (for more details, see Section 5.3.1). The rationale for selecting the specific patterns is twofold:

- ***Usage frequency:*** behavioral patterns are the most commonly used patterns, accounting for about half of the design pattern usages in a system (Ampatzoglou, Charalampidou and Stamelos, 2011). Additionally, State/Strategy patterns are the most used patterns among all, and Template Method the third. Therefore, the accumulated impact of these patterns on energy consumption is expected to be high;

- ***Main object-orientation mechanism:*** object-orientation has three pillars[1]: encapsulation, inheritance, and polymorphism (Weisfeld, 2013). Polymorphism is the most commonly explored principle within the GoF patterns (19 out of 23 patterns uses polymorphism). However, it is important to highlight that encapsulation and inheritance, although less explored, are also present in the solution of many patterns. From these mechanisms, polymorphism potentially influences energy consumption the most, as it comprises a complex procedure to map the polymorphic calls to the correct implementation (Harper and Morrisett, 1995). Both State/Strategy and Template Method use polymorphism as their main mechanism to provide the pattern solution and, therefore, have potentially high impact on the energy consumption. The two studied patterns use polymorphism with different goals: State/Strategy pattern uses it to define the interface to interact with the states/strategy, while Template Method pattern uses it to define the points of specialization to be implemented by the concrete classes. In particular, the State/Strategy pattern encapsulates the different states/strategies, whereas the Template Method pattern exploits inheritance, since concrete classes extend the functionality of the abstract class. For that reason, we point that other pillars are part of our investigation, although

[1] Some authors advocate a fourth pillar: abstraction. However, this is a higher level concept, which is provided as combination of the other three pillars and, therefore, is not relevant for our argumentation.

polymorphism is the main mechanism.

To investigate the energy consumption, we compare the energy efficiency of pattern solutions with the energy efficiency of their alternative designs (one for each pattern), through a crossover experiment. We note that the alternative designs were developed in a standardized way (see Sections 5.3.2 and 5.3.4). In the experiment, we focus our investigation on pattern-related methods[2] so as to enable a fine-grained analysis of the energy consumption. In addition to exploring the differences between pattern and alternative solutions, we also investigate some pattern-related parameters that can cause the pattern to be either beneficial or harmful with respect to energy consumption. For the experiment, we selected two large well-known open source software (OSS) systems.

The remainder of this chapter is organized as follows. In Section 5.2, an overview of the related work on energy consumption in design patterns, and alternatives to design patterns is provided. Section 5.3 presents background information necessary for understanding the experiment, i.e., the selected design patterns and their alternative solutions. Section 5.4 presents the experiment planning, which describes the research questions, hypotheses, the used tool and collected variables. Section 5.5 overviews the execution of the experiment (i.e., data collection and validation). In Section 5.6, we elaborate on our analysis and answer the research questions. In Section 5.7, we discuss the obtained findings, by focusing on the most important observations and presenting implication for researchers and practitioners. The threats to the validity of our study are discussed in Section 5.8, followed by the conclusion of this chapter in Section 5.9.

5.2 Related work

This section presents research efforts that discuss the effects of design patterns on energy consumption. We focus on the consumption of design patterns, the types of patterns being investigated, and the proposed alternatives for patterns. After discussing the related work, an overview of how our research compares to related work is provided.

In the work of Bunse et al. (2013), a case study on the overhead of design patterns compared to “clean software” is presented. In this context, “clean software” is a chunk of design that could be refactored into a pattern solution. The software in this study mainly targets mobile devices. The design patterns discussed are Facade, Abstract Factory, Observer, Decorator, Prototype, and Template Method. This initial investigation shows that each of these design patterns has overhead when compared

[2] Pattern-related methods are methods that play a role within the design pattern.

to their “clean” counterparts. Most of the patterns have a relative small overhead, except for the Decorator pattern, which, based on this study, consumes more than double the amount of energy compared to the “clean” counterpart.

Additionally, Sahin et al. (2012) performed a more extensive investigation on the impact of design patterns on energy usage. In particular, this study takes into account the feasibility, impact, consistency, and predictability of the energy consumption of 15 design patterns, from all GoF pattern categories. The creational design patterns discussed are the Abstract Factory, Builder, Factory Method, Prototype, and Singleton. The structural patterns discussed are the Bridge, Composite, Decorator, Flyweight, and Proxy pattern. Finally, the behavioral patterns that were selected are the Command, Mediator Observer, Strategy, and Visitor. Results of the study suggest that the use of design patterns, either increases or decreases the amount of energy used. Additionally, there are no relations of the category of the design pattern and the impact on energy usage. Finally, this study shows that it is not possible to precisely estimate the impact of design patterns on energy consumption when only considering artifacts on design level.

Litke et al. (2007) conducted an initial exploration of the energy consumption of design patterns. This chapter includes an analysis of five design patterns, for which the energy consumption and performance are described. These design patterns were tested by the use of six example applications written in C++. These applications were first tested as clean, i.e., without the usage of design patterns, and then transformed with the designated design pattern. The design patterns discussed are the Factory Method, Adapter, Observer, Bridge, and Composite. For Factory Method, Adapter, and Observer, differences were found between the original application and the one containing the specified design pattern. The results show that applying Factory Method or Adapter patterns does not necessarily impose a serious threat to the energy consumption. However, a significant overhead was identified by employing the Observer pattern, but additional research is still required to investigate the cases when Observer is indeed a threat to energy consumption. Since the Bridge and Composite pattern had no significant difference in power consumption, the authors suggest further analysis.

In a recent paper, Noureddine and Rajan (2015) performed a comparison on the energy consumption overhead caused by 21 design patterns and explored in details the effects of two design patterns (Observer and Decorator pattern). The effects discussed in this paper are the energy consumption of applications using the pattern solution, the non-pattern solution, and an optimized alternative for the design patterns. The optimized solutions for the alternatives are integrated into the applications by making changes to compilers, so that the optimizations are automatically processed when compiling. This study suggested that simple transformations to the

Observer and Decorator patterns are able to provide reductions in energy consumption in the range of 4.32% to 25.47%. We clarify that the patterns investigated in our study are included among the 21 patterns initially investigated by Noureddine and Rajan. However, the comparison of these results (from the initial investigation) to ours is limited, since some extra details (e.g., implemented alternatives, source code properties) would be necessary to further elaborate the discussion (see Section 5.7.1).

To ease the comparison of our work to the aforementioned studies, we summarize the main differences in Table 5.1, according to the following aspects: (a) Design patterns addressed; (b) Number of nontrivial systems used; (c) Number of pattern instances analyzed; (d) Number of pattern-related methods analyzed; (e) Level of energy measurement[3] (process level or method level); (f) Level of investigation[4] (instance level or method level); and (g) Number of investigated parameters that influence energy consumption. Based on Table 5.1, the main contributions of this study compared to the research state-of-the-art are the following:

- **Usage of nontrivial systems**—our investigation is performed considering two nontrivial systems and a considerable amount of pattern instances and pattern-related methods. This setup allows allow us to observe realistic results that are more representative to the population of existing software-intensive systems;

- **Exploitation of a method-level approach for measuring energy consumption**—in addition to the more traditional approach of process-level measurement. Being able to isolate the energy consumed by specific method calls, we obtain measurements with lower overhead, allowing a more in-depth investigation of both pattern and alternative solutions, in the sense that we focus on pattern-related methods of each pattern instance; and

- **Exploration of parameters of the processed patterns**—in this study, we investigate not only the energy efficiency of State/ Strategy and the Template Method design pattern, comparing them against their respective alternative (non-pattern) design solutions, but also the parameters of their application that render them either beneficial or not. We clarify that related work has indicated parameters as possible causes for greater energy consumption, but without any investigation of these parameters.

[3] Measurement at process level considers the energy consumed by the operating system process of the running software; measurement at method level considers the energy consumed by a specific method within the software process.

[4] Investigation at instance level considers pattern instances as subjects for analysis while the method level considers the pattern-related methods as subjects.

Table 5.1: Overview of related work

Reference	Design patterns	Nontrivial systems	# of instances	# of methods	Measurement level	Investigation level	# of parameters
(Bunse et al., 2013)	6[a]	0	6	0	Process	Instance	0
(Sahin et al., 2012)	15[b]	0	15	0	Process	Instance	0
(Litke et al., 2007)	5[c]	0	5	0	Process	Instance	0
(Noureddine and Rajan, 2015)	21[d]	0	N/A*	0	Process	Instance	0
This study	3[e]	2	21	169	Process and Method	Method	3

[a] Facade, Abstract Factory, Template Method, Prototype, Decorator, and Observer.

[b] Abstract Factory, Builder, Factory Method, Prototype, Singleton, Bridge, Composite, Decorator, Flyweight, Proxy, Command, Mediator, Observer, Strategy, and Visitor.

[c] Factory Method, Observer, Adapter, Bridge, and Composite.

[d] Decorator, Observer, Mediator, Strategy, Template Method, Visitor, Abstract Factory, Builder, Factory Method, Prototype, Singleton, Bridge, Flyweight, Proxy, Chain of Responsibilities, Command, Interpreter, Iterator, State, Adapter, and Composite.

[e] State, Strategy, and Template Method.

[*] Not available, the authors only mention “several small examples”.

5.3 Design Patterns and Alternatives

In this section we present background concepts that facilitate the understanding of our experiment. In particular, we discuss the GoF design patterns that are explored in this study (State, Strategy, and Template Method), elaborating on their design structure and an overview of their uses and consequences. Additionally, we present and discuss their alternative solutions (referred in this chapter as State/Strategy Alternative and Template Method Alternative). The identification of design pattern alternatives can be a nontrivial activity, since some GoF design patterns have no reported alternatives in the literature (Ampatzoglou, Charalampidou and Stamelos, 2013a). To consider a design as a design pattern alternative, it should:

- originate from the *literature*;
- provide exactly the *same functionality* as the pattern; and
- have notable *structural differences* compared to the pattern.

We used two main sources to find alternatives: the seminal book on design refactoring by Fowler et al. (1999) and a systematic literature review conducted by Ampatzoglou, Charalampidou and Stamelos (2013a), in which an overview of GoF design pattern alternatives are presented and discussed. Based on the aforementioned criteria, we selected well-known alternative solutions from the literature, as they are expected to be more recurrent in existing software. Although we acknowledge the existence of design patterns and alternatives that are optimized for energy efficiency (which would obviously lead to better solutions), we have deliberately not included them in our study. The reason for this decision is that we intend to focus on widely-known solutions that have been applied to various software projects, by developers who are not aware of energy optimization mechanism. Investigating such optimized solutions can potentially introduce bias to our results, since neither patterns nor alternatives would be in their standard form.

5.3.1 State/Strategy

The State pattern allows an object to change its behavior by switching from one state to another (Gamma et al., 1995). One classic example for the State pattern are traffic lights that turn from green to yellow, yellow to red and red back to green. The collection of all states defines the space in which the context (traffic light) is able to change its behavior. This behavior is implemented by each of the states separately. The context class has at least one state instance object (i.e., a concrete state) that represents its current state and thus functions as a central interface for clients to communicate with (see model on the left in Figure 5.1). This context delegates the handling of requests to its current state object. The State pattern is used in scenarios where either the behavior of an object depends on its state and needs to be changed during runtime, or the operations have large, multipart conditional statements that depend on the object's state (Gamma et al., 1995). Applying the State pattern has a number of consequences: the specific behavior for each state is localized; the state transitions are made explicit; and State objects can be shared when they have no instance variables.

The Strategy pattern allows for the encapsulation of certain families (such as algorithms), allowing them to be interchangeable depending on client requests or specific behaviors of the context (Gamma et al., 1995). The context class has at least one object of the concrete strategy that provides its (unique) functionalities, which are implemented according to a template defined by the strategy interface (see model on the right in Figure 5.1). The Strategy pattern can be used in a number of different situations (Gamma et al., 1995), e.g., when a class has different behaviors (depending on a specific situation) or when there are multiple implementation options to be chosen. Consequences of using this pattern include (Gamma et al., 1995): it becomes an alternative for sub-classing the context directly or using conditional statements, by decoupling the algorithms into their own family; and it may cause memory and computational overheads, because it increases the number of used objects, and concrete strategies may not use all information they receive when called.

By inspecting the class diagrams of State and Strategy patterns (see Figure 5.1),

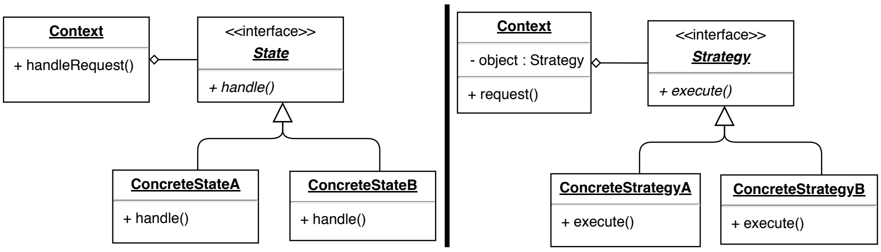


Figure 5.1: UML model of State (on the left) and Strategy (on the right) patterns

we observe that they have an equivalent structure (i.e., skeleton design) (Gamma et al., 1995; Tsantalis et al., 2006). Both patterns have a context that is called by an external client and a family that consists of an interface with concrete classes. Both contexts contain an object that represents at least one or more states/strategies that can be uniformly handled. The main difference is the logic beneath the patterns, i.e., the behavior is fundamentally different. In the case of the State pattern, the current object (state) within the context is updated after the execution of every behavior (the method handle, in the diagram). This is not necessary for the Strategy pattern, as strategies may be interchangeable during runtime. Additionally, the change of strategies is more an additional feature than a rule for the Strategy pattern, whereas for State this is the basic concept of the pattern. In this study, we treat both patterns mutually, since the expected changes to measure energy consumption is focused on the design, i.e., structure and the use of their common object-orientation mechanisms. The aforementioned fundamental differences regard the behavior of the pattern instance and, thus, are not expected to be a confounding factor for our study, unless these fundamental differences systematically change design attributes (e.g., method size). Nevertheless, we have not identified such cases in our dataset (see Section 5.5.2).

5.3.2 State/Strategy Alternative

In a literature review performed by Ampatzoglou, Charalampidou and Stamelos (2013a) many alternatives for the State/Strategy pattern are presented (Adamczyk, 2003, 2004; Ferreira and Rubira, 1998; Henney, 1999, 2002; Sobajic et al., 2010; Victório et al., 2010). Similarly, Fowler et al. (1999) discuss several alternatives for these two patterns. Among these available options, we have chosen to replace the use of polymorphism with the use of conditional statements. In this solution, the entire structure of the State/Strategy pattern is removed and the complete logic is implemented in the context, which now has a local enumerator object that enables the shifting between the different behaviors. Listing I shows an example of alternative implementation for a Strategy pattern instance. While implementing an alternative design, the implementation of each concrete strategy would be replaced with the behavior of the corresponding state and the state update.

Listing 5.1: Example implementation of Strategy alternative

```
public class Strategy {

    public enum Strategies{
        Strategy1,
        Strategy2,
        Strategy3
    };

    private enum currentStrategy;

    public int[] sort(int[] list) {
      switch(currentStrategy) {
          case Strategy1:
              // Implementation of Strategy 1.
          break;
          case Strategy2:
              // Implementation of Strategy 2.
          break;
          case Strategy3:
              // Implementation of Strategy 3.
          break;
          case default:
              return 0;
          break;
      }
   }
}
```

Despite the simplicity of the recommended changes, creating alternatives requires some effort, as design patterns may be implemented in various different ways. These variations should be reflected into the alternative designs. Based on our experience, one specific type of variation had direct impact in the implementation of the alternative: the structure of the implemented pattern may differ from the originally proposed structure (Tsantalis et al., 2006). Specifically, the proposed structure of State/Strategy has a standard Interface-Class (IC) hierarchical structure; however, it may also be implemented with an abstract class between the interface and the class (an intermediate level of inheritance), becoming an Interface-AbstractClass-Class (IAC) hierarchical structure. Such a structure may contain several abstracts classes in the middle. To deal with abstract classes in the alternative, each behavior

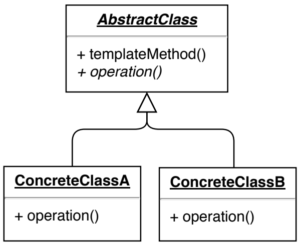


Figure 5.2: UML model of the Template Method pattern

defined in a concrete class would be combined with the abstract class behavior. If that is not possible, e.g., when a class or abstract class is used from the Java library, an additional object would be created to be able to access its functionalities. We clarify that other, less recurrent, variations are possible, but they are not handled in this study. For example, a State/Strategy may comprise multiple interfaces, which are partial responsibilities, and concrete classes may implement all or some of them.

5.3.3 Template Method

Similarly to Strategy, the Template Method isolates different algorithms or operations to their own subclass. However, this pattern allows the subclasses to alter certain steps of an algorithm without changing the structure of the algorithm. An abstract class has at least two operations, one primitive, which is used by the concrete subclass to implement the steps of an algorithm, and a template method that contains the default structure (see Figure 5.2). The Template Method pattern can be used to avoid code duplication, and to control or restrict any extensions of an abstract class, so that an abstract function or hook function can only be called on certain locations.

5.3.4 Template Method Alternative

Fowler et al. (1999) presents several alternatives for the Template Method and Ampatzoglou, Charalampidou and Stamelos (2013a) discuss one alternative. From these options, we chose the starting point from the Form Template Method (FTM) refactoring, presented by Fowler et al. (1999). Generally, FTM transforms a non-pattern code into a Template Method (see Figure 5.3). In contrast to State/Strategy alternative (Section 5.3.2), in which we completely eliminated polymorphism, the alternative for Template Method does use polymorphism, but in a different fashion. Therefore, this study design cannot be considered appropriate for comparing the

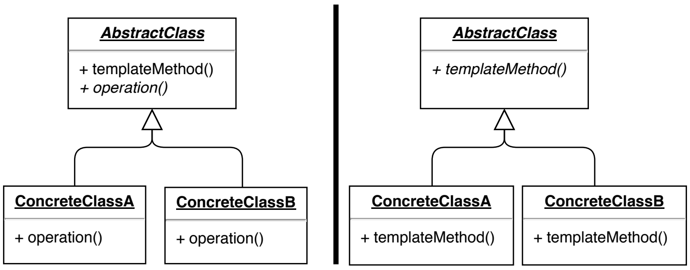


Figure 5.3: Comparison of the Template Method pattern (on the left) against its alternative (on the right)

effect of using polymorphism on energy efficiency.

By using this alternative, both primitive operations and specific behavioral operation now reside in each concrete class. However, the Template Method also leaves room for variants in its implementation. In such cases, the adjustments that would be applied in the alternative to handle these variations are described below. Similarly to State/Strategy, the Template Method allows all or none of these adjustments to be included.

- **Depth of Inheritance Tree:** Even though the Template Method uses only one abstract class, it is possible that the methods are already defined in an interface. This makes it harder to remove the primitive methods when creating the alternative implementation. In these cases, the primitive method is not removed, but it is moved to the concrete class. This allows us to both keep the IAC structure and to implement the alternative.

- **Private methods:** It is possible for a template method to call private methods within the abstract class. If this is the only case, the private method is called, the private method is also moved down to the concrete class. When this is not possible, the operations within the method are moved inside the template method. This is not feasible in cases the operations rely on multiple other methods or sources. In such a case, the private method is changed to protected.

As for State/Strategy, other, less recurrent, variations are possible, but are not handled in this study. For example, a concrete class may aggregate the abstract class, possibly creating recursive calls, which are not originally intended for template method pattern instances.

5.4 Experimental Planning

In this section, we present the design and materials of the experiment reported in this chapter. This experiment is reported based on the guidelines of Wohlin et al. (2012) and on the structure proposed by Jedlitschka et al. (2008). Initially, the research objective, questions and respective hypotheses of the study are discussed, followed by the process to select objects of study and experimental units. Next, an overview of the variables and instruments used to the data collection are presented. Finally, the analysis procedure is described. For presentation purposes, we report the data collection procedure along with the execution process, in Section 5.5.

5.4.1 Objectives, Research Questions, and Hypotheses

The goal of this study is defined according to the Goal-Question-Metrics approach (van Solingen et al., 2002), as follows: *"**analyze** instances of State, Strategy, and Template Method patterns **for the purpose of** evaluation **with respect to** their energy consumption **from the point of view of** software developers **in the context of** open source systems"*. To achieve this goal, we set three research questions (RQs):

RQ$_1$: What is the difference between the application of the Template Method pattern and an alternative design solution in terms of energy consumption?

RQ$_2$: What is the difference between the application of the State/Strategy pattern and an alternative design solution in terms of energy consumption?

RQ$_3$: What are the parameters that influence the energy consumption of State, Strategy, and Template Method pattern instances?

RQ$_1$ and RQ$_2$ aim at investigating whether the energy consumption of patterns and alternative solutions is significantly different. Such information is of paramount importance to make more informed decisions when selecting patterns over alternatives, while developing energy efficient software. To answer RQ$_1$ and RQ$_2$, we formulated the following hypotheses:

H$_0$: There is no difference between the energy consumed by software using a design pattern solution and software using an alternative design solution.

H$_1$: The energy consumed by software using a design pattern solution is significantly lower than the energy consumed by software using an alternative solution.

H$_2$: The energy consumed by software using a design pattern solution is significantly higher than the energy consumed by software using an alternative solution.

RQ_3 aims at exploring if there are pattern-related parameters that affect the energy consumption of the patterns, and for which ranges of these parameters the pattern can be characterized as beneficial or harmful. Such thresholds can serve as guidance for decision making on when to apply a design pattern or not. To answer this research question, we isolate groups (e.g., A and B) of pattern-participating methods whose members have a similar difference in the energy consumption (compared to the alternative solution) and investigate specific structural characteristics of the pattern solution (for more details, see Section 5.4.4). To test the difference between every two groups, we formulated the following hypotheses:

H_0: There is no difference between the parameter values of the two groups (A and B).

H_1: The parameter value of group A is higher than the value of group B.

H_2: The parameter value of group B is higher than the value of group A.

5.4.2 Design Type and Experimental Units

To answer the research questions and test the hypotheses, we designed a crossover experiment (Wohlin et al., 2012), in which pattern-related methods are the experimental units. Pattern-related methods are methods of pattern instances that play a role within the design pattern. For our two selected patterns, these methods are the template method (Template Method pattern) and the methods implementing the behavior of states or strategies (States/Strategy pattern). We selected this unit for three reasons: (a) units with finer granularity facilitate a more detailed investigation of parameters (i.e., design characteristics) that influence the energy efficiency of design pattern solutions; (b) to standardize the data collection, since patterns may have multiple pattern-related methods, each one implementing different responsibilities; and (c) the alternative solutions provide the same functionality compared to pattern-related methods, but with a different implementation, what also promotes standardization of the data collection. For each experimental unit (i.e., a pair of pattern and alternative solutions), we record all data needed to answer the research questions, i.e., the energy consumption measurements for both pattern and alternative solutions, and design characteristics of the pattern solution.

To collect data for the experiment, it is necessary to select software systems and pattern instances from which to sample pattern-related methods. Regarding software systems, we decided to use OSS that met the following criteria:

- are written in the Java programming language, since the tool for retrieving design pattern instances (see Section 5.4.3) is limited to Java;

- are nontrivial systems that are either widely used or known, so as to avoid the use of toy examples; and
- contain instances of both the Template Method or the State/Strategy patterns.

Two OSS projects were selected for the study. Selecting more projects would be unrealistic as all alternative solutions had to be manually implemented by us, which is a time-consuming task. However, we do investigate a sufficient number of pattern instances (more than related work) and pattern-related methods. For further discussion, please see how we deal with threats to validity (Section 5.8). The first OSS project is ***JHotDraw***[5], a Graphical User Interface (GUI) framework written in Java that allows the creation of technical and structured graphical images. The project started in 2000, having about 80,000 downloads at this point, and the current version (7.6) has 680 Java source files, containing 80,535 SLOC. JHotDraw was developed as a design exercise, for applying GoF design patterns, becoming a powerful framework that is acknowledged by the software engineering community as a benchmark for GoF design patterns detectors (Aversano, Canfora, Cerulo, Del Grosso and Di Penta, 2007; Seng et al., 2006). The second OSS project is ***Joda Time***[6], an Application Program Interface (API) that can replace the standard date and time classes, providing better quality and in-depth functionalities. The project started in 2003, having almost 500K downloads at this point, and the current version (2.9.2) has 329 Java source files, containing 85K SLOC. Joda Time has a high rating on GitHub and has also been used for research purposes (Manotas et al., 2014).

Despite the careful selection of representative software for the study, we acknowledge that nontrivial (complex) systems may have associated risks, in the sense that the transformation of a nontrivial pattern instance to an alternative solution might not be uniform. To mitigate this risk, we developed a strategy while selecting pattern instances / pattern-related methods, and implementing the alternative solutions. Firstly, to select pattern instances for the study, we consider only those that meet the following criteria:

- **Used within the application:** It is possible that the found pattern instances are not used within the applications themselves, e.g., functionalities provided as an API, whose pattern instances are partially implemented by the API user;
- **Reachable:** Some pattern instances are not reachable directly, imposing a long (and hard to predict) sequence of calls, what may bias the measurement process. One option is to modify the source code to make the pattern instance easier to reach, but it would bias the results as well;

[5] http://www.jhotdraw.org/
[6] http://www.joda.org/joda-time/

- **Performing deterministic tasks:** Certain pattern instances may perform non-deterministic tasks, such as saving data to files or transferring data over the network. This could interfere with the actual measurement process; and
- **Not too complex:** In some cases, the pattern instances could have a relatively high number of members, e.g., twenty or more concrete states/strategies or are variants of the original pattern that are not handled in our study (see Sections 5.3.2 and 5.3.4). These pattern instances would make the process of implementing the alternatives infeasible. On top of that, such pattern instances would represent a threat to study validity, as these comprise exceptional cases.

Regarding method selection, the same criteria applied to pattern instances is used. We believe that the pattern instances and pattern-related methods filtered by these criteria are representative of the population, as excluded cases are mostly exceptional. Finally, concerning the implementation of alternative solutions, we have to ensure that the original business logic is preserved, avoiding unnecessary changes to the original source code. As the alternatives preserve the original business logic and only the difference in the energy consumption is analyzed, we believe that we have mitigated much of the risk associated with the usage of nontrivial programs.

5.4.3 Variables and Instrumentation

To answer the research questions and test the hypotheses stated in Section 5.4.1, a number of variables are derived. These variables are divided into two distinct categories: (a) pattern-related information (*pattern*, *method* and *m-** in Table 5.2, which are explained in the first subsection); and (b) measurements of energy consumption (**-ptt* and **-alt* in Table 5.2, which are explained in the second subsection). These variables are recorded for each unit of analysis (i.e., pattern-related methods). The entire process of identifying and measuring the units of analysis culminates in the creation of a dataset of all extracted variables for each unit. This dataset is recorded as a table in which the columns correspond to collected variables. In the following subsections, we present and discuss the variables and the tools used to extract them.

Pattern-related Information

To collect the necessary data for all units of analysis, we first find all the pattern occurrences within the OSS applications. To detect the design patterns occurrences, we use a tool developed by Tsantalis et al. (2006). This tool uses a Similarity Scoring Algorithm (SSA) for detecting design structures similar to a desired GoF design pattern. Among the 12 detectable patterns are Template Methods and State/Strategy

Table 5.2: List of collected variables

Variable	Description	Tool
pattern	Pattern Type (Template Method or State/Strategy)	SSA
method	The pattern-related method that is measured	
m-sloc	SLOC of the pattern-related method	-
m-mpc	MPC of the pattern-related method	
papi-ptt	Energy consumption (in Joules) of the pattern solution, at process level	PowerAPI
papi-alt	Energy consumption (in Joules) of the alternative solution, at process level	
jalen-ptt	Energy consumption (in Joules) of the pattern solution, at method level	Jalen
jalen-alt	Energy consumption (in Joules) of the alternative solution, at method level	
ptop-ptt	Energy consumption (in Joules) of the pattern solution for triangulation	pTop
ptop-alt	Energy consumption (in Joules) of the alternative solution for triangulation	

(identified jointly due to structural similarity). The extraction of the design patterns is done by isolating subsystems of a given application through static analysis, which enables the identification of relationships between the elements of each separate subsystem. The SSA tool has been assessed by several studies (such as Kniesel and Binun (2009) and Pettersson et al. (2010)), which have positively evaluated its performance, precision, and recall rates. SSA was, therefore, selected for this study because of the following:

- *it provides detection of the design patterns of interest*, i.e., Template Method and State/Strategy; and
- *it provides acceptable performance*, as described by Tsantalis et al. (2006), also when compared to similar tools (Kniesel and Binun, 2009; Pettersson et al., 2010).

SSA is limited to the Java programming language, since the similarity analysis is performed on compiled Java class files. After the application of the pattern detection tool on a project, the results are compiled into one Extensible Markup Language (XML) file that contains all the instances found within a given application.

Additionally, a set of metrics has to be extracted, which are used to investigate parameters that influence the energy consumption of pattern instances (see Section 5.4.4). In order to select these metrics, we considered the SQualE platform (Balmas et al., 2010), as it summarizes a broad and comprehensive list of metrics from the literature. From this list, we identified two metrics that could be measured at method level: SLOC and MPC[7]. SLOC is measured as the amount of source line of code

[7] MPC consists of the number of direct invocations to methods that are not owned or inherited by the class being measured.

of the method, while MPC is measured as the amount of calls, within the method, to other methods (these calls do not include those to methods of the same class, even if inherited). We clarify that the parameters SLOC and MPC are calculated for the pattern solution only. For answering RQ_3, we are interested in identifying characteristics of the pattern design solution that are related to energy efficiency. In addition, SLOC and MPC do not change considerably in the alternative solution, since the transformation mostly causes a reorganization of the code and how methods are called. In other words, our goal is not to evaluate the change of complexity, but how the complexity of the pattern solution influences the difference of energy consumption between the solutions, especially because this complexity is dictated by the business logic, which is not modified.

Assessment of Energy Consumption

To measure the energy consumption of software applications, there are multiple tools based on both software and hardware (Noureddine et al., 2013). In this study we, opted to use software tools, as they allow finer-grained measurements (i.e., at the method level) (Noureddine et al., 2013). Although hardware measurement offers a higher precision, it estimates the energy consumed by the whole machine, and our study investigates the consumption difference at the methods level. Therefore, we prioritized a finer-grained technique over a more precise one. In addition, selecting and configuring a hardware measurement tool may represent a complex and expensive task (Diouri et al., 2014), which if not accurately performed can introduce additional bias. In order to select the appropriate tools, we searched the literature and identified nine software tools for measuring energy consumption. We analyzed two comparative studies that included these tools (Chen et al., 2012; Noureddine et al., 2013), in addition to other literature, so as to verify their theoretical and empirical validity in scientific setups. Based on this analysis, two tools presented the highest precision, namely PowerAPI and pTop; a third tool, namely Jalen, although with lower precision, is able to deliver finer-grained measurements. Other tools that we considered either do not have sufficient validation or present lower precision regarding their respective granularity of measurement, or require additional hardware investments.

PowerAPI is an API that enables real-time profiling of the energy consumption at the level of operating system (OS) processes (Noureddine et al., 2012b,a, 2013, 2015). This tool currently supports measuring energy from CPU and network, which are represented through power modules. The available implementations that are provided for this tool are created for GNU/Linux distributions, but they are independent of the hardware. To measure the energy consumption of the CPU, the

Thermal Design Power (TDP) is taken into account, which is the maximum amount of heat (which is generated by the CPU) that requires to be dissipated by the cooling system. The precision for measuring the power consumption of software applications with PowerAPI was estimated by Noureddine et al. (2015) by comparing it against a power meter. This estimation showed that the calculated margin of error vary from 0.5% to 3%.

Jalen is an energy consumption profiler, which was created by the same developers of PowerAPI (Noureddine et al., 2012b, 2013, 2014, 2015). Jalen can collect energy consumption on different levels of granularity such as the method level. Similarly to PowerAPI, Jalen is limited to the use on GNU/Linux distributions due to the sensors used for the hardware components. Since Jalen injects monitoring code through the bytecode instrumentation, it reduces the precision of the measurement. In a comparison of tools performed by Noureddine et al. (2015), the measured time for individual Tomcat's server requests was 57% higher in average. However, since we are comparing two different versions of the same applications (i.e., pattern and alternative solutions), this cannot be considered as a confounding factor.

pTop is a profiler that can determine energy consumption on the OS process-level and is designed to work solely on GNU/Linux distributions (Do et al., 2009; Noureddine et al., 2013). pTop calculates the energy consumption through a daemon that profiles the resource utilizations for all processes, whereas the power consumption of the system CPU, network interface, memory and hard drive are tracked. Each different system component needs to be configured (possibly calibrated as well) according to its specifications. Just like PowerAPI, it uses the TDP to calculate the energy consumed by the CPU. The precision of pTop was analyzed by comparing its results to a wattmeter (Noureddine et al., 2013). Results of this analysis show that the average median error for pTop was less than 2 watts.

All the aforementioned energy measurement tools are suitable candidates to obtain reliable results. However, PowerAPI and Jalen are designed to specifically measure the energy consumption of Java applications, not including the overhead caused by the Java Virtual Machine (JVM). Due to the granularity of the energy measurement of Jalen (i.e., method level), the output is not influenced by the energy expenditure of other parts of the system, which makes it a more suitable tool. However, in order to compare the related work to ours, it is also necessary to consider the same perspective used in related work, i.e., process level measurements, in this case by using PowerAPI. Therefore, we decided to use both PowerAPI and Jalen for the study. We clarify that both tools have a limitation of being able to measure energy consumed by the CPU only. Therefore, among other reasons, we restricted the experimental units to those that do not use extra resources (e.g., hard drive, or network). Additionally, we decided to use pTop, which is more commonly known in the scientific community, for triangulation purposes, to validate the measurements obtained from PowerAPI and Jalen, and to verify the memory energy consumption (see Section 5.5.2).

5.4.4 Analysis Procedure

During the data analysis, the previously described variables (see Table 5.2) are used to answer the research questions. As mentioned in Section 5.4.3, we collect data using two different tools (PowerAPI and Jalen) and, therefore, every task of the analysis is performed for the data of each tool separately, and results are compared. In addition, the data regards two design patterns (Template Method and State/Strategy) and every step of the analysis is repeated for both patterns separately. The data analysis is twofold, described in the following.

General Analysis of Energy Consumption

Initially, we compare the energy measurements (**-ptt* and **-alt*) to test the hypotheses posed by research questions RQ_1 and RQ_2. For evaluating whether or not the pattern solution is significantly different from the alternative solution, we perform two steps:

1. *Check distribution*. To decide whether to use parametric or non-parametric tests, we verify the distribution of each dependent variable metric (i.e., *papi-ptt*, *papi-alt*, *jalen-ptt*, and *jalen-alt*) by employing the Shapiro-Wilk test (Field, 2013). If not normal, a Wilcoxon signed ranks test (Field, 2013) is used for assessing the difference between pattern and alternative solutions; otherwise, paired sample t-test (Field, 2013) is used; and

2. *Compare energy consumption*. Next, we compare whether the difference between pattern and alternative solutions is statistically relevant. For that, we employ the dependent sample test for investigating the data obtained by PowerAPI and Jalen (i.e., *papi-ptt* vs. *papi-alt* and *jalen-ptt* vs. *jalen-alt*).

Analysis of Design Parameters

Once the difference in the energy consumption between pattern and alternative solutions is observed, we want to investigate parameters that may influence this difference. For that, we isolate controlled groups (i.e., clusters) with similar difference in the energy consumption and test the hypotheses posed by RQ_3. This analysis comprises the following steps:

1. *Create clusters based on consumption*. First, we create clusters based on the difference between the energy measurements for PowerAPI (i.e., *papi-diff = papi-ptt - papi-alt*) and Jalen (i.e., *jalen-diff = jalen-ptt - jalen-alt*). For that, we employ the agglomerative hierarchical clustering technique, considering the average linkage method (or between-groups linkage) and using squared Euclidian distance (Hastie et al., 2009);

2. *Merge clusters based on design parameters*. Next, we investigate whether or not the clusters are statistically different with regards to the analyzed design parameters (*m-sloc* and *m-mpc*). As the clusters comprise independent samples, we employ Mann-Whitney tests (Field, 2013) for this investigation. The analysis for each parameter is performed separately and clusters that are not statistically different are merged; and

3. *Verify trends*. Finally, based on the final disposition of the clusters, we verify trends with regards to both SLOC and MPC.

It is important to clarify that during the analysis we noticed cases in which the pattern solution was more energy efficient than the alternative solution and, however, the clustering algorithm did not separate these units (see Section 5.6.4). Therefore, aiming at complementing the answer for RQ_3, an additional analysis is performed, which comprises the following steps:

1. *Group units*. Based again on the difference between the energy consumption, we separate the experimental units into two categories: (a) pattern solution consumed more energy than the alternative solution; and (b) pattern solution consumed less energy than the alternative solution; and

2. *Compare parameters*. Next, we analyze if the design parameters (SLOC and MPC) may have an influence on determining which solution is more energy-efficient. For that, we employ Mann-Whitney tests for investigating whether each parameter is statistically differ between the two groups created in the previous step.

5.5 Execution

In this section we explain how data for the experiment was collected. Firstly, we describe the data collection procedure, showing details of the most relevant aspects. Next, we present and discuss the validation of the collected data according to the planned experiment.

Table 5.3: Descriptive of identified pattern occurrences and pattern-related methods

OSS	Included occurrences		Excluded occurrences	
	TM	**SS**	**TM**	**SS**
JHotDraw	7(**15**)	6(**56**)	5	25
Joda Time	7(**80**)	1(**18**)	5	17
TOTAL	14(**95**)	7(**74**)	10	42

TM = Template Method, SS = State/Strategy

5.5.1 Data Collection

The data collection is composed of four steps. Firstly, we extracted the pattern instances and selected the pattern-related methods (i.e., experimental units). To collect the experimental units, a set of pattern occurrences were extracted from JHotDraw and Joda Time, and were manually inspected to decide whether pattern instances could be included or excluded (see Section 5.4.2). Table 5.3 distinguishes between the number of pattern occurrences that were included and excluded (according to the process described in Section 5.4.2) for each OSS and GoF design pattern. For each included pattern instance, a set of units of analysis was collected. The total number of collected units for each OSS and GoF design pattern is presented between parentheses in Table 5.3. We clarify that, despite the limited number of included pattern instances, we believe that the number of experimental units (95 and 74) is satisfactory, providing statistically significant results (see Section 5.6). Moreover, the effort required to implement the alternatives (as described in Sections 5.3.2 and 5.3.4) also restricted the amount of experimental units that could be collected.

Next, for each unit, we calculated the parameters SLOC and MPC (i.e., based on the pattern solution, see Section 5.4.3). Before starting the measurement process, we implemented the alternative solution for each pattern instance as described in Sections 5.3.2 and 5.3.4. Then, to measure the energy consumption of the units, a standard measurement process was defined. This measurement process needed to be consistent throughout the whole test run, so no external interference is introduced to the results. First, a selection was done for the hardware system to be used for the analysis, along with the OS and distribution. For the hardware system, we chose the MSI wind box DC100 minicomputer due to its simplicity, availability, and compatibility with the measurement tools. The MSI wind box contains the following components:

1. AMD Brazos Dual Core E-450 (1.65GHz) with a TDP of 18 Watts;

2. 4GB of DDR3 memory; and

3. AMD Radeon HD 6320 graphics adapter.

Since the measurement tools are tailored for GNU/Linux system, we used one of the distributions released for that OS. As we wanted less interference during the measurement process, a clean installation of Ubuntu is used, which contains only the essential packages and has no user interface. However, since JHotDraw requires a graphical shell to call certain functionalities, a simplistic window manager, i3[8], was installed on top of this distribution. For orchestrating and standardizing the execution of the measurement tools and pattern related methods, a script was created for performing the following procedure: start the measurement tool, wait a few seconds for the tool to load, execute the usage scenario containing the pattern-related method, wait for the application to finish and stop the measurement tool. Each usage scenario embedded multiple executions of a part of the application that called one pattern-related method (i.e., experimental unit), guaranteeing measurable energy consumption (i.e., more than 30 seconds). Any selected part of the application was the simplest possible and was fully checked to guarantee no hard external bias (e.g., read/write operations). Each usage scenario was executed with the pattern solution and the alternative solution. For reliability purposes, the aforementioned procedure was executed 100 times for every pair scenario-solution, obtaining 100 measurements for each experimental unit. Finally, we obtained the final value for each unit of analysis by excluding outlying measurements and calculating the average between the remaining measurements.

5.5.2 Validation of the Collected Data

There were three main assumptions in the experimental design that needed validation. Firstly, two researchers verified every manual data collection task. These tasks were the selection of the patterns instances and pattern-related methods, the calculation of the SLOC and MPC parameters, and measurement of energy consumption. Secondly, as we considered experimental units from State and Strategy pattern instances mutually, we verified whether there was a difference between the energy consumed by them. Our results suggest no visual or statistically relevant differences. Last, the energy consumption data was validated by triangulation.

As mentioned in Section 5.4.3, the energy consumption was obtained by two tools, one working at process level (PowerAPI) and another working at method level (Jalen). Our motivation for selecting these two tools was that they both estimate the energy consumption based on the JVM, therefore, reducing the bias from the overhead caused by the OS. In addition, PowerAPI has higher precision when

[8] https://i3wm.org/

compared to other tools, while Jalen, although having a lower precision, provides more fine-grained measurements (as it captures only the energy consumption of the method). By obtaining the two different perspectives, we aimed at comparing our study to related work, as well as verifying the results w.r.t. the different levels of measurements.

As expected, the tools provided measurements of different magnitudes, which are related to the different characteristics of the tools. In addition, PowerAPI and Jalen use a similar mechanism for exploring the JVM to calculate the results, which could be biased. Besides, both tools can collect the energy consumed by the CPU only and, although we restricted the experimental units to those not requiring additional resources (e.g., hard-drive, network), not considering the energy consumed by the memory could still represent a bias. Therefore, we sought to provide further validation of the estimated measurements. To this end, we selected a process level tool, pTop, which can estimate the energy consumed by both CPU and memory, as well as has a higher precision, but estimates measurements by exploring the process management of the OS.

The data collected by pTop suggest that the energy consumed by the memory is negligible (approx. 0.0001% of the total energy consumed for every experimental unit). In addition, to verify that our data collection was consistent, we triangulated the measurements. For that, we performed eight Spearman correlation tests. For each pattern (pattern = Template/Method or State/Strategy), we tested the correlation between each design solution (pattern and alternative) of PowerAPI/Jalen and pTop (i.e., *papi-ptt* vs. *ptop-ptt*; *papi-alt* vs. *ptop-alt*; *jalen-ptt* vs. *ptop-ptt*; and *jalen-alt* vs. *ptop-alt*) . By observing that all tests proved a rather very strong correlation (see Table 5.4), we considered all the measured data to be consistent and reliable for data analysis. Finally, it is interesting to notice that Jalen has a lower correlation to pTop, compared against PowerAPI. This is yet another evidence of the consistency of the results, as Jalen is a method level tool and, thus, do not have the overhead caused by the rest of the application.

5.6 Analysis

In this section we present the results of the experiment. Firstly, we show the descriptive statistics of the dataset. Next, we present the results of the analysis carried out for each research question, which was executed as described in Section 5.4.4. We clarify that every statistical test was performed using the tool IBM SPSS Statistics[9] and are reported based on the guidelines suggested by Field (2013).

[9] http://www-03.ibm.com/software/products/en/spss-statistics

Table 5.4: Pearson correlation test for validating estimated measurements from PowerAPI and Jalen

Pattern	Tool	Pattern solution (pTop)			Alternative solution (pTop)		
		N	Correlation Coefficient	Sig.	N	Correlation Coefficient	Sig.
Template Method	PowerAPI	95	0.946	<0.01	95	0.947	<0.01
	Jalen	89	0.893	<0.01	87	0.877	<0.01
State/ Strategy	PowerAPI	74	0.963	<0.01	74	0.929	<0.01
	Jalen	71	0.933	<0.01	71	0.791	<0.01

TM = Template Method, SS = State/Strategy

5.6.1 Descriptive Statistics

For every experimental unit, pattern-related variables were collected (variables pattern, method, *m-sloc* and *m-mpc*), and an alternative solution was implemented as described in Section 5.3. Afterwards, the tools PowerApi, Jalen, and pTop were used to collect the energy consumption from both pattern and alternative solutions (**-ptt* and **-alt*). We remind that an experimental unit comprises a pair of pattern and alternative design solutions. A summary of all numeric variables (i.e., SLOC, MPC, and energy consumption measurements) is presented in Table 5.5 and Table 5.6, showing relevant descriptive statistics for Template Method and State/Strategy, respectively. As can be seen in Table 5.5 and Table 5.6, few measurements were performed by Jalen for the pattern and/or the alternative solution. This is due to a limitation from Jalen, which tries to measure a specific method, but it is unable to encapsulate the entire process. This is caused when either the length in time that the method uses is too little, or when the method delegates its functionality in a way that Jalen cannot track. Such cases were properly treated during the statistical analyses, which are discussed in the following subsections.

Before performing the data analysis based on the energy measurements from PowerAPI (*papi-**) and Jalen (*jalen-**), these measurements were checked against the measurements from pTop (*ptop-**). The details of this validation process are presented and discussed in Section 5.5.2. When observing the measurement from the three tools, one can notice that they are different, following the order Jalen < PowerAPI < pTop. This difference in the measurements is expected. Jalen measures the consumption at a method level (i.e., not considering the consumption of the whole program); PowerAPI measures the consumption of the Java process (i.e., the program); and pTop measures the consumption of the OS's process (i.e., which also include the overhead of the JVM). When ordering the values, it is possible to notice that greater overheads result in greater values, i.e., Jalen < PowerAPI < pTop.

Table 5.5: Descriptive statics of numeric variables for the Template Method pattern (*pattern* = Template Method)

Variable	N	Min	Max	Mean	Std. Error (Mean)	Std. Deviation
m-sloc[a]	95	2.00	36.00	6.03	0.59	5.75
m-mpc[b]	95	0.00	12.00	1.33	0.20	1.97
papi-ptt[c]	95	92.30	1086.77	327.88	27.11	264.23
papi-alt[c]	95	92.12	924.09	270.84	23.77	231.67
jalen-ptt[c]	89	43.85	799.88	200.38	14.57	137.47
jalen-alt[c]	87	22.58	777.32	150.13	10.38	96.84
ptop-ptt[c]	95	189.78	2198.84	719.86	59.68	581.72
ptop-alt[c]	95	193.85	2185.72	594.85	47.37	461.66

[a] Measured in number of uncommented lines in the pattern solution
[b] Measured in number of method invocations in the pattern solution
[c] Measured in Joules

Table 5.6: Descriptive statics of numeric variables for the State/Strategy pattern (*pattern* = State/Strategy)

Variable	N	Min	Max	Mean	Std. Error (Mean)	Std. Deviation
m-sloc[a]	74	0.00	36.00	5.68	0.69	5.93
m-mpc[b]	74	0.00	29.00	2.13	0.54	4.66
papi-ptt[c]	74	157.58	1664.17	738.17	56.51	499.11
papi-alt[c]	74	136.37	1002.25	341.95	19.88	175.54
jalen-ptt[c]	68	27.38	1260.11	486.34	42.54	350.75
jalen-alt[c]	66	20.20	635.89	186.96	14.72	119.56
ptop-ptt[c]	74	316.08	4124.87	1640.06	129.72	1145.63
ptop-alt[c]	74	273.21	2260.22	786.15	46.77	413.04

[a] Measured in number of uncommented lines in the pattern solution
[b] Measured in number of method invocations in the pattern solution
[c] Measured in Joules

5.6.2 RQ_1: Template Method

The first research question aims at exploring the energy consumption of Template Method pattern instances and their alternative solutions, focusing on identifying if there is a statistically significant difference between the solutions (pattern and alternative) regarding energy consumption. For that, we considered the energy consumption measured by two different tools, one at process level (*papi-**) and one at method level (*jalen-**). While the former tool provides a more traditional and system-wide measurement, the latter provides a more fine-grained measurement allowing us to focus on the point of interest (pattern-related method), excluding any interference from the rest of the system. Although we did not expect to find differences in the results obtained from the two tools (because both pattern and alternative solutions are subject to the same interference), the method-level measurements should

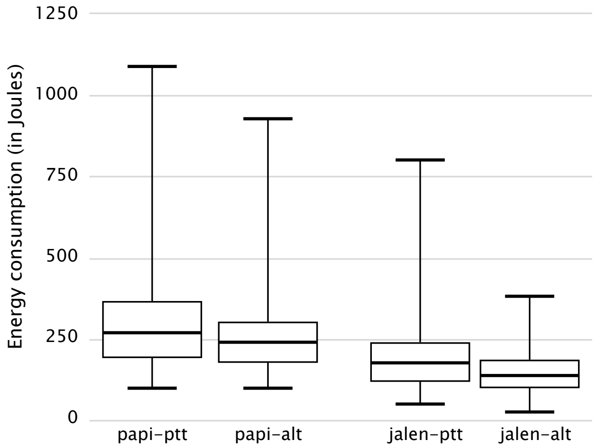


Figure 5.4: Visual comparison of the energy consumption for Template Method

provide lower overhead (i.e., smaller energy measurements).

To answer this research question, we examined the pair of variables obtained from PowerAPI and Jalen (i.e., *papi-ptt* vs. *papi-alt* and *jalen-ptt* vs. *jalen-alt*). In order to decide if we were using parametric or non-parametric tests for assessing the statistical significance of the differences between the pair of variables, we employed the Shapiro-Wilk test to check the distribution of data for each variable. The results of the test suggest that data were not following the normal distribution and, thus, a non-parametric test had to be employed. Therefore, we used the Wilcoxon signed ranks test to evaluate the hypotheses posed for RQ_1 (see Section 5.4.1) and, thus, investigate the energy consumption data from PowerAPI and Jalen. In addition, to support visualizing the difference in the energy consumption, Figure 5.4 shows the box-plot for each compared variable.

From the analysis results, two main findings can be highlighted. First, pattern solutions consumed more energy than their alternatives based on the results of both PowerAPI ($p < 0.01$, $z = -4.92$) and Jalen ($p < 0.01$, $z = -5.57$). This is evident in the results from both tools (following the result of the descriptive statistics—Table 5.5), which suggest a decrease of 17.4% (PowerAPI) and 24.34% (Jalen) on the energy consumption of the alternative solution. Second, Jalen showed a greater difference than PowerAPI (by comparing the z-score), which also follows the trend observed by comparing their descriptive statistics. It is also important to highlight that, as expected, method level measurements (from Jalen) showed lower consumption then process level (from PowerAPI). These findings corroborate that: (a) method level measurements have lower overhead, since they isolate application and OS noises,

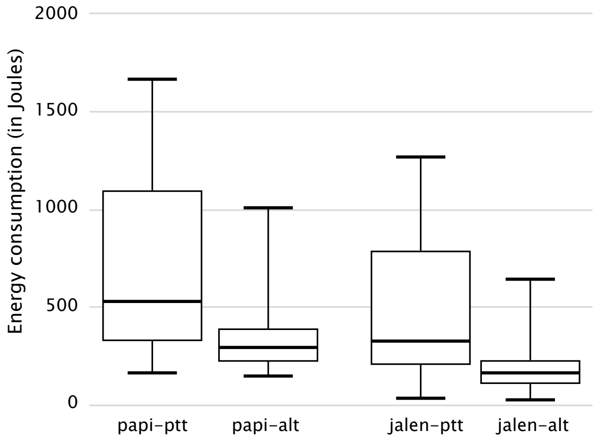


Figure 5.5: Visual comparison of the energy consumption for State/Strategy

and (b) pattern solutions indeed show increased energy consumption when compared against their alternatives. Summarizing, we can answer RQ_1 by affirming that, for ***Template Method, pattern solutions tend to consume more energy than the alternative solutions (implemented as described in Section 5.3) and that this observation becomes more evident when analyzing at method level***. However, further investigation on this assertion is presented in Section 5.6.4 (see RQ_3).

5.6.3 RQ_2: State/Strategy

Next, we explored the energy consumption of State/Strategy pattern instances and their alternative solutions, focusing on identifying whether or not there is a statistically significant difference between the solutions (pattern and alternative) regarding energy consumption. For that, we followed the same process as described for RQ_1, using, however, data related to State/Strategy (pattern = State/Strategy). Thus, first we performed the Shapiro-Wilk test to confirm that the data was not normal, as expected due to the result from the previous data analysis. As the variables were not normal, we used Wilcoxon signed ranks test to investigate the energy consumption data from PowerAPI and Jalen. In addition, to support visualizing the difference in the energy consumption, Figure 5.5 shows the box-plot for each compared variable.

Similarly to RQ_1, the pattern solutions consumed more energy than their alternatives based on the results of both PowerAPI ($p < 0.01$, z = -6.19) and Jalen ($p < 0.01$, z = -6.8). This result is in accordance with what we expected from observing the descriptive statistics in Table 5.6 and the box-plots in Figure 5.5. However, when looking at the differences between each pair of pattern and alternative solutions, we can highlight some notable aspects. First, results obtained from PowerAPI and Jalen are very close to each other, as it can be observed by the mean value in Table 5.6 and z-score of the test. Second, the average results from Jalen are lower than those from PowerAPI, but still similar, especially by taking into consideration that Jalen only measures the energy consumption at method level. Even though these values are very close, the standard deviation and standard error mean (see Table 5.6) from Jalen are proportionally higher (in comparison with the mean) than those of PowerAPI. The relatively high standard deviation and standard error for the Jalen pair is caused by differences in measurements on method level. The method level measurements seem to be significantly distant from the mean. Nonetheless, the drop in energy consumption for both pairs is remarkable, as the average decrease in energy consumption is 53.68% for PowerAPI and 55.51% for Jalen. Summarizing, we can answer RQ$_2$ by affirming that, ***concerning State/Strategy, pattern solutions tend to consume more energy than the alternative solution (implemented as described in Section 5.3), although method level measurements show that this result requires further investigation (due to the high standard deviation and error)***. We present this further analysis in the next section, in which we discuss parameters that influence energy consumption.

5.6.4 RQ$_3$: Influence of Source Code Parameters

The third research question aims at investigating parameters that influence the energy consumption of design pattern instances. To achieve this goal, we considered two metrics (SLOC and MPC) collected from every pattern-related method (i.e., based on the pattern solution, see Section 5.4.3) to investigate clusters of experimental units via a three-step analysis. First, to cluster the experimental units, we performed an agglomerative hierarchical clustering (using between-groups linkage and squared Euclidean distance, see Section 5.4.4) based on the difference in energy consumption (i.e., **-diff = *-ptt - *-alt*). Second, we employed Mann-Whitney tests to evaluate the hypotheses posed for RQ$_3$ (see Section 5.4.1), verifying whether neighbor clusters (i.e., that are at the same level in the hierarchical tree) are statistically different w.r.t. SLOC and MPC. If no statistically significant difference was found, we merged the clusters and performed the test again with the neighbor of the merged cluster. Finally, we investigated the final clusters to identify trends regarding the studied metrics (SLOC and MPC).

In Figure 5.6 we present the outcome of the hierarchical clustering for Template Method data. The two charts on the top show the distribution of the experimental units among the clusters. Each point consists of a pair <pattern solution; alternative solution>, in which the Y axis is the energy consumption of the pattern solution

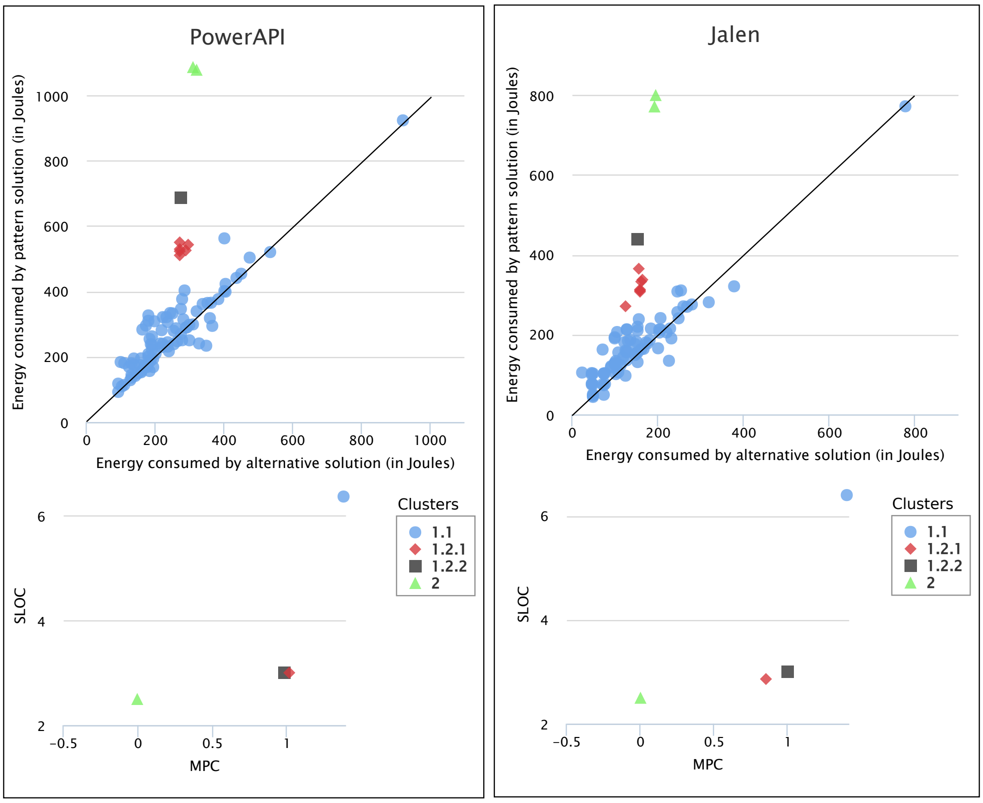


Figure 5.6: Hierarchical clustering of Template Method units of analysis

and the X axis is the energy consumption of the alternative. The clusters can be identified by the different shape and color presented in the legend. The two charts on the bottom of Figure 5.6 show the centroids of the clusters with regards to SLOC and MPC. The values for SLOC and MPC of each cluster are obtained as the average of the units of the cluster. By checking these charts, it is already possible to notice some separation between clusters w.r.t. the two metrics.

We investigated the separation between clusters, and the results of the statistical tests are presented in Table 5.7 (along with the tests for State/Strategy data). As multiple tests were performed for investigating every pair of cluster regarding every metric, we report only the statistically significant results. Based on these tests, one can see which clusters were merged. For example, when comparing the clusters of Template Method (see upper charts of Figure 5.6) one can see four clusters (1.1; 1.2.1; 1.2.2; 2), from which two are very close (1.2.1; 1.2.2) and the separation was

Table 5.7: Mann-Whitney test for comparing clusters

Pattern	Tool	Metric	Mann-Whitney test		
			Clusters	Z	Sig.
Template Method	PowerAPI	SLOC	1.1 & 1.2	-2.46	0.02
	Jalen	SLOC	1.1 & 1.2	-2.94	<0.01
State/Strategy	PowerAPI	SLOC	2.1 & 2.2	-3.62	<0.01
		MPC	2.1 & 2.2	-2.86	<0.01
		SLOC	1 & 2.1	-4.77	<0.01
		MPC	1 & 2.1	-5.03	<0.01
		SLOC	1 & 2	-4.31	<0.01
		MPC	1 & 2	-4.70	<0.01
	Jalen	SLOC	1.1.1 & 1.1.2	-2.91	<0.01
		MPC	1.1.1 & 1.1.2	-2.16	0.03
		SLOC	1.1 & 1.2	-2.46	0.01
		MPC	1.1 & 1.2	-3.83	<0.01
		SLOC	1 & 2	-3.62	<0.01
		MPC	1 & 2	-3.63	<0.01

not statistically relevant w.r.t. to SLOC and MPC, forming a merged cluster (1.2). By further inspecting the statistical tests and charts, one can see three clusters distant from each other (1.1; 1.2; 2) w.r.t. to both SLOC and MPC, and that they follow a trend in which clusters that group more energy-efficient solutions (e.g. 1.1) have bigger SLOC and MPC scores. This observation suggests that the higher the SLOC and MPC, the less advantageous the alternative solutions.

Figure 5.7 shows the scatterplots that concern the State/Strategy pattern, on which we performed the same analysis described for Template Method. When investigating the clusters (based on the statistical tests—see Table 5.7), one can deduce that three clusters remain for PowerAPI (1; 2.1; 2.2) and four for Jalen (1.1.1; 1.1.2; 1.2; 2). Although the data of the two tools led to slightly different clusters, the results suggest the same trends, which are similar to the ones observed for the Template Method pattern. In particular, both SLOC and MPC influence the benefit of using an alternative instance instead of the pattern solution. Moreover, the cluster 1.1 from the PowerAPI data (which is similar to cluster 1.1.1 from Jalen) is the closest to the bisect line (i.e., pattern solution = alternative solution) and, by checking the metrics chart, it is clear that this cluster has much higher SLOC and MPC when compared to the others.

The performed analysis is so far able to provide evidence that both SLOC and MPC influence the energy efficiency of a pattern solution and that both parameters should be taken into account when deciding between using a pattern solution or an alternative solution. However, there is one interesting question that has not been an-

Table 5.8: Mann-Whitney test for comparing most energy efficient solutions

Pattern	Tool	Metric	Mann-Whitney test	
			Z	Sig.
Template Method	PowerAPI	SLOC	-3.62	<0.01
		MPC	-2.86	<0.01
	Jalen	SLOC	-2.91	<0.01
		MPC	-2.16	0.03
State/ Strategy	PowerAPI	SLOC	-3.62	<0.01
		MPC	-2.86	<0.01
	Jalen	SLOC	-2.91	<0.01
		MPC	-2.16	0.03

swered by the clustering, yet. By observing all the scatterplots that were presented until this point, it is clear that in some cases pattern solutions were more energy efficient than the alternative solutions (i.e., experimental units below the bisect line in the upper charts of Figures 6 and 7). However, the use of automated clustering algorithms did not separate these units. Therefore, we decided to perform a second analysis. We grouped the experimental units into the two categories (pattern > alternative; and pattern < alternative). Next, we were able to investigate whether or not SLOC and MPC may have an influence on determining if a pattern solution is more energy-efficient than the alternative solution. To explore the differences between these groups, in terms of SLOC and MPC, we employed Mann-Whitney tests. Table 5.8 shows the results of the test for both Template Method and State/Strategy.

Based on the results of Table 5.8, it becomes clear that SLOC has a significant influence on the energy efficiency of the pattern instance for both GoF design patterns, suggesting that the longer the method is, the more possible it becomes that the pattern solution is more energy efficient. For the State/Strategy pattern, it is also statistically evident that the number of calls to other methods influences the energy efficiency of the solution, suggesting that more calls are related to a higher possibility of the pattern solution being more efficient than the alternative.

Summarizing the evidence so far, it is possible to answer RQ$_3$ by affirming that ***both parameters, i.e., number of source lines of code and the number of invoked methods, influence the energy efficiency of a pattern solution, suggesting that higher SLOC and/or MPC are related to more energy efficient pattern solutions when compared against their alternative solutions***.

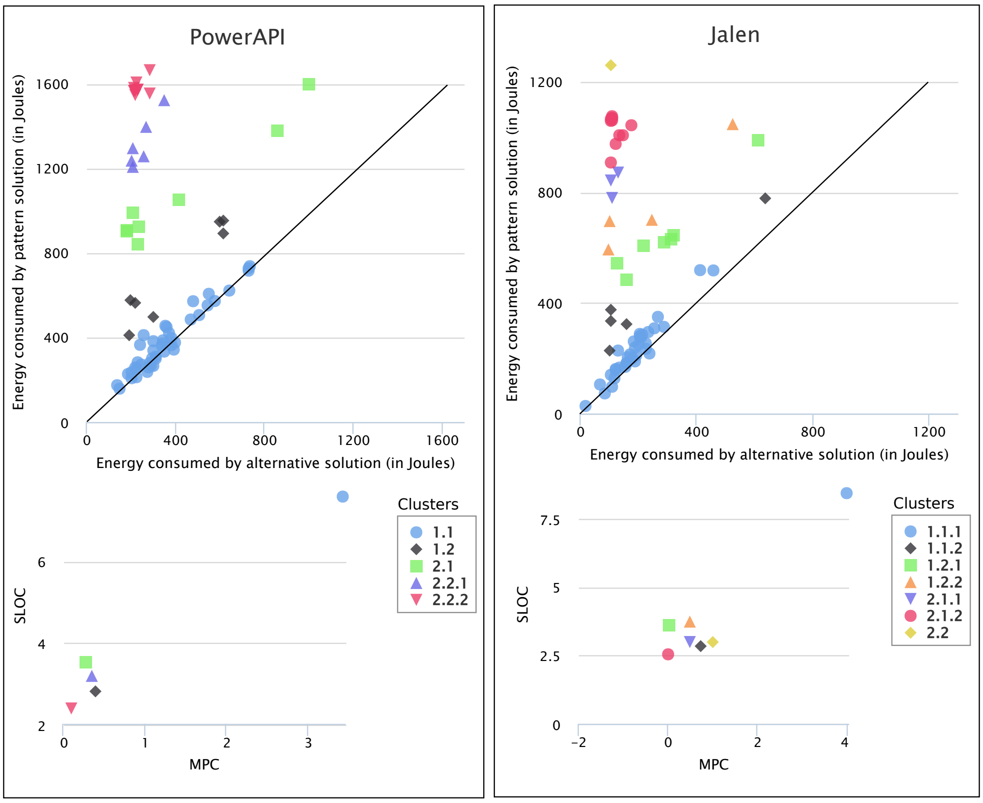


Figure 5.7: Hierarchical clustering of State/Strategy units of analysis

5.7 Discussion

In this section we discuss the main outcomes of this study. First, we discuss the findings of the experiment, comparing them with related work. Second, we discuss the implications to researchers and practitioners. However, we need to clarify that the discussion presented in this section regards only the Template Method and State/Strategy patterns, as well as that our observations and interpretations are constrained by the limitations of the experimental settings and threats to validity (see Section 5.8).

5.7.1 Interpretation of Results

The results of our experiment suggest that the alternative solutions are more energy efficient than the pattern solutions for both Template Method and State/Strategy. This difference is higher for State/Strategy (approx. 54% for PowerAPI and 56% for Jalen) than to Template Method (approx. 17% for PowerAPI and 24% for Jalen). These results are in accordance to related studies (see Section 5.2), which have reached similar conclusions, i.e., that the alternative solutions tend to be more energy efficient. Specifically, Bunse et al. (2013), as well as Noureddine and Rajan (2015), also report on the Template Method pattern, and suggest that this pattern presents a small, yet significant, overhead. Noureddine and Rajan (2015) also investigate State and Strategy patterns separately, and report a smaller overhead for State (approx. 3%) and an equally small improvement for Strategy (approx. 3%). This difference between results may be related to certain characteristics of the study design (e.g., the used pattern alternative or subjects of the study), but more details regarding these characteristics would be necessary to elaborate on the rationale. To sum up, the differences between pattern and alternative solutions observed in our study are likely to be influenced by the overhead caused by employing polymorphism (i.e., the main mechanism of both patterns). When calling polymorphic methods, the JVM has to dynamically indicate the correct implementation to be used. Commonly, this indication is done by moving the instruction pointer[10] to the memory address containing the right method. Although simple, this kind of operation can become computationally expensive if overused.

While investigating the influence of SLOC and MPC on the energy consumption of pattern solutions, we were able to notice that both GoF patterns tend to provide a slightly more energy-efficient solution when used to implement more complex behaviors (i.e., with longer methods and multiple calls to method of external classes). This observation is also intuitive from three perspectives:

1. GoF design patterns are not beneficial in simple/non-complex design problems (even w.r.t. other quality attributes (Hsueh et al., 2008; Huston, 2001)), since the extra complexity that they introduce is higher than the one that they resolve;

2. The effect of polymorphism weakens when these patterns are handling complex situations. The longer the method, the lower the ratio of method localization compared to the overall computation and, therefore, the overall overhead caused by the polymorphic mechanism of Template Method or State/Strategy; and

3. It is understandable that patterns promote improved structuring of the source code, which may sometimes lead to a smaller and/or more efficient bytecode

[10] Also known as program counter, instruction address register, instruction counter and instruction sequencer, instruction pointer is a processor register that indicates the current assembly command to be executed.

(for the JVM), which in turn leads to slightly more energy-efficient software. We observed such cases, e.g., when the pattern-related method comprises a set of external invocations (i.e., to methods that are not owned or inherited by the class being measured). In such cases, the JVM might be applying internal optimizations, which would not be possible in the alternative, as the structure pattern-related method is altered.

Although we have provided evidence that alternative solutions are in most of the cases more energy efficient than pattern solutions (approx. 79% of the cases), there are cases in which the opposite holds. Sahin et al. (2012) have also reported on pattern instances that can be more energy efficient compared to alternative solutions. In comparison to Sahin et al., we provide a more fine-grained analysis by relating this differentiation to two metrics (i.e., SLOC and MPC). This finding can also be possibly explained by the overhead caused by polymorphism, as we were able to identify statistically significant differences on the metrics between pattern-efficient (i.e., pattern solution consumed less energy than the alternative solution) cases and alternative-efficient cases. On average, pattern-efficient solutions have 65.83% more source lines of code and 43.37% more method invocations than the alternative-efficient solutions.

Finally, there is a crosscutting observation to all findings in this chapter, which deals with differences in energy consumption at method and process levels. The measurements from Jalen were lower than the measurements from PowerAPI (40.42% on average). This observation is intuitively correct since the measurements from Jalen are more localized (focused on only one method). Furthermore, it is interesting to notice that differences between pattern and alternative solution were smaller for Jalen (12.11% in average), a fact that suggests that the remaining parts of the applications (i.e., not the pattern-related methods) were, to some extent, biasing the analysis. Another possible explanation could be that the dynamic binding procedure[11] may not be fully captured by Jalen at times, as it focuses on the pattern-related method being measured. However, we sought to mitigate this threat by: (a) verifying cases of dynamic biding while selecting experimental units (i.e., pattern related methods); (b) looking for outlying measurements; and (c) checking the correlation of the measurements against pTop (see Section 5.5.2).

5.7.2 Implications to Researchers and Practitioners

The findings of this chapter suggest that pattern solutions are less harmful or even beneficial to energy consumption when the responsibility assigned to the pattern

[11] Dynamic binding procedure refers to the action of resolving a binding (e.g., decide which method or variables with same names to use) at runtime, when it is not possible at compile time.

instance (i.e., the implemented behavior) is nontrivial. Therefore, we advise practitioners on considering this parameter when deciding whether or not to apply Template Method or State/Strategy patterns. GoF patterns serve several purposes: structuring and organizing source code; supporting quality attributes, such as maintainability and reusability; and improving communication between stakeholders by providing a common language. For these reasons, GoF patterns have become a common practice in software development. Several studies that have investigated only a subset of the GoF patterns report that approx. 30% of the classes of a system may participate in pattern instances (Ampatzoglou et al., 2015; Ampatzoglou, Kritikos, Arvanitou, Gortzis, Chatziasimidis and Stamelos, 2011; Khomh et al., 2009). However, as studies have shown, there are also side effects on using GoF patterns (Ampatzoglou, Charalampidou and Stamelos, 2013b), and energy efficiency is one of the aspects in which the software is negatively affected. Thus, we also advocate the careful consideration of drivers (e.g., energy efficiency) of the software project, balancing them against the forces (e.g., complexity of the behavior to be implemented) that influence the decision on applying a certain pattern or not.

Based on the aforementioned negative relationship between GoF patterns and energy efficiency, one may wonder why using GoF patterns in systems that have energy efficiency as a main concern. Nevertheless, GoF patterns are widely adopted and, therefore, we expect that even systems that have energy as a concern may have a non-negligible amount of GoF pattern instances, either intentionally (to promote other quality attributes) or unintentionally. Therefore, the results of our study can be used to help control a system's efficiency in different situations. On the one hand, while developing software, our findings may support the management of unintentional harm to energy efficiency (via not necessary use of GoF patterns), as well as intentional use to balance various quality attributes. On the other hand, when refactoring a system for a new purpose, the findings of this study may support the decision making process on what parts of the system to refactor and how.

This study has three main implications to researchers. First, the usage of nontrivial systems for investigating patterns energy consumption is a challenging task, since researchers need to a) deal with pattern variants, b) decide which variants have to be investigated, c) incorporate these variations into the alternative solution, and d) measure the energy consumption of the pattern instance by executing the same scenario for which the pattern instance was intended to. However, the obtained evidence can be very insightful as shown by this study. Therefore, we do suggest that when investigating the energy consumption of GoF patterns, nontrivial systems should be used. Second, the use of method level energy measurements has proven to provide extra information for investigating the hypotheses both visually and statistically. It also contributed to the reliability of our findings by triangulating results of process and method level measurements. Third, when investigating the energy efficiency of GoF patterns, exploring design parameters (e.g., SLOC and MPC) proved to be highly relevant. By investigating the parameters, we were able to not only suggest whether or not the pattern solution is worse than the alternative solution, but also, and most importantly, we were able to interpret this phenomenon. By further investigating this hypothesis and observing the magnitude of the influence of these parameters, we were able to highlight the circumstances under which the patterns are more efficient than the alternative. Therefore, we suggest exploring similar parameters and other design and source code properties when investigating the influence of GoF design patterns to energy consumption.

5.8 Threats to Validity

In this section, threats to construct validity, internal validity, reliability, and external validity of this study are discussed. Construct validity reflects how far the studied phenomenon is connected to the intended studied objectives. Internal validity expresses to what extent the observed results are attributed to the performed intervention, and not to other factors. Reliability is linked to whether the experiment is conducted and presented in such a 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 is that the transformation of nontrivial systems may be risky since, due to their complexity, it is more error-prone. Although "synthetic" programs could facilitate the control over external factors, we believe that nontrivial programs were imperative to investigate pattern-related methods. Thus, to mitigate this bias, we took several measures while selecting experimental units (see Sections 5.4.2 and 5.4.3). The collected energy measurements pose another threat, as we consider consumption only by the CPU. If we included energy consumed by other resources, such as hard drive and network, the results might change. First, by only looking at CPU consumption, it enabled us to use three different measurement tools to increase the confidence on the obtained measurements. To mitigate this threat further, we verified that the energy consumed by the memory was negligible and do not represent a considerable bias (see Section 5.5.2), as well as restricted the selection of pattern instances to those that do not require operations such as writing to or reading from files and communicating through network. Another threat concerns the level of measurement (i.e., process or method level), which can be a source of bias to the study as different perspectives could lead to different results. For that reason, we performed the analysis at both levels (process and method), and checked their correlation. Additionally, some lack of precision could have been introduced by a limitation of the used energy measurement tools. To mitigate this threat, we selected tools that have been validated in different studies. In addition to that, we performed data triangulation for all measurements by using three different tools. Moreover, the measured data may also be slightly biased, since small environmental changes might exist between different executions, leading to different values. To mitigate this threat, we used a basic OS, installing only strictly required dependencies, and every measurement was performed multiple times, using the average value for the analysis.

The main threat to the internal validity of our experiment is related to whether the observed differences in the energy consumption were caused by the implemented alternatives, and not by other factors. To mitigate this threat, we acted from measurement and implementation perspectives. On the one hand, we used Jalen, which is able to measure only the energy consumed by the experimental unit (i.e., pattern related-method), discarding the energy consumed by the rest of the application, JVM, and OS. In addition, the procedure to measure the energy consumed by pattern and alternative solutions was identical. On the other hand, while implementing the alternatives, we assured that only the design changes proposed for the alternatives (see Sections 5.3.2 and 5.3.4) were implemented, not altering the behavior of the pattern-related method. Another threat to this category is the fact that the set of parameters that we investigated for answering RQ_3 is not exhaustive, and we cannot guarantee that differences in energy consumption have been comprehensively explained, since there might be other parameters that influence the energy consumption of design patterns.

In order to mitigate reliability threats, two different researchers were involved in the data collection, double-checking all outputs. In addition to the two researchers, a third one was involved in the analysis procedure. To implement the alternative solutions, the provided guidelines are sufficient and any replication should lead to the same results. To complement that, all scripts and source code are available online[12] and, therefore, all raw data can be reproduced with small variations by using the same energy measurement tools and environment setup. Finally, data analysis bias is limited in this study, since no subjectivity was involved.

Finally, concerning external validity, we have identified four possible threats. First, we investigated a limited number of OSS projects. However, the two selected projects are very different, both in terms of domains and characteristics (e.g., Joda Time has more than the double of SLOC per class when compared to JHotDraw); this partially alleviates this threat. Second, we investigate a limited number of pattern instances, as well as a limited range of pattern variants. However, we evaluate a fair

[12] http://www.cs.rug.nl/search/uploads/Resources/JSEP_Feitosa_etal_resources.zip

number of pattern-related methods (i.e., units of analysis), what partially alleviates this threat. Nevertheless, a larger sample could strengthen the results, and increase our confidence on generalizing our findings. Next, the presented results are dependent on the used alternatives and pattern solutions. Thus, different alternatives or pattern variations could lead to altered results. For example, alternative and/or pattern solutions optimized for energy efficiency may increase the observed difference between the solutions, or even invert it. However, the focus of our study was to analyze representatives of existing and commonly used nontrivial software, in terms of both pattern and alternative design solutions, as such investigation would impact a plethora of software. Therefore, we selected the alternatives that we believe to be the most common, as well as considered the original definition of the studied patterns (also with small and similarly common variations), so as to have a more representative sample of solutions that exist in practice. Finally, the results of this study cannot be directly generalized to other GoF patterns, especially those that do not use polymorphism as their main mechanism.

5.9 Conclusions

In this chapter, we investigated the effect of Template Method, and State/Strategy GoF design patterns on energy consumption. In particular, we conducted an experiment on two nontrivial OSSs, JHotDraw and Joda Time, from which we identified 21 pattern instances and 169 pattern-related methods (i.e., methods that use the pattern structure), implemented an alternative (non-pattern) solution for each instance (which contained the alternative implementation of the pattern-related methods), and measured the energy consumption of both solutions using tools at both process and method levels. Based on the collected data, we identified which solution was more energy-efficient and what parameters affect the efficiency of the pattern solution. To this end, we collected two metrics from every pattern-related method, SLOC and MPC, and correlated them to the efficiency of the pattern solution.

The results of the study suggest that the alternative solution surpasses the pattern solution in most cases. However, in some cases the pattern solution had similar or even slightly lower energy consumption than the alternative solution. Since these cases were identified in large pattern-related methods and/or methods with high number of method invocations, it is suggested that these patterns are more suitable when more complex behaviors have to be implemented. We clarify that some factors, such as the considered design pattern alternatives, may have influence on the aforementioned observations, and that altering these factors may change the aforementioned observations (for more details, see Section 5.8).

The findings of this study have value for both practitioners and researchers. On the one hand, practitioners can reuse this knowledge to perform more informed decision-making when applying GoF patterns. On the other hand, researchers can learn from the reported experiences and reproduce aspects of this study when investigating GoF design patterns and/or energy consumption.

The findings reported in this chapter and Chapter 4 suggest that the practice of applying design patterns is a promising solution to safeguard quality attributes in CES development as the effect of GoF patterns on critical quality attributes is controlled and deterministic. However, similarly to any other design artifact, design pattern instances tend to drift from their original implementation. The next chapters will unfold and report on the investigation of one phenomenon related to the decay of design pattern instances, and how it may influence the observation obtained so far.

Based on:

Daniel Feitosa, Paris Avgeriou, Apostolos Ampatzoglou and Elisa Y. Nakagawa, (2017) *"The Evolution of Design Pattern Grime: An Industrial Case Study,"* in Proceedings of the 18th International Conference on Product-Focused Software Process Improvement (PROFES '17), Innsbruck, Austria, pp. 165–181, DOI:10.1007/978-3-319-69926-4_13

Chapter 6

The Evolution of Design Pattern Grime: An Industrial Case Study

Abstract

Instances of GoF design patterns may comprise a large portion of software systems. However, there are concerns regarding the efficacy with which software engineers maintain pattern instances, which tend to decay over the software lifetime if no special emphasis is placed on them. Pattern grime (i.e., degradation of the instance due to buildup of unrelated artifacts) has been pointed out as one recurrent reason for the decay of GoF pattern instances. Seeking to explore this issue, this chapter presents an investigation about the existence of relations between the accumulation of grime in pattern instances and various related factors: (a) projects, (b) pattern types, (c) developers, and (d) the structural characteristics of the pattern participating classes. For that, we empirically assessed these relations through an industrial exploratory case study involving five projects (approx. 260,000 lines of code). Our findings suggest a linear accumulation of pattern grime, which may depend on pattern type and developer. Moreover, we present and discuss a series of correlations between the accumulation of pattern grime and structural characteristics. The outcome of our study can benefit both researchers and practitioners, as it points to interesting future work opportunities and also implications relevant to the refinement of best practices, the raise awareness among developers, and the monitoring of pattern grime accumulation.

6.1 Introduction

In Java applications, it has been reported that the number of classes that participate in GoF pattern occurrences can vary from 15% to 65% (e.g., in software libraries) (Ampatzoglou et al., 2015; Khomh et al., 2009), leading to a significant influence on the overall quality of the system. However, the effect of patterns on quality is not uniform (Ampatzoglou, Charalampidou and Stamelos, 2013b); the same pattern can have both a positive and a negative effect on the quality of a software product. Therefore, gaining more insights on how exactly patterns have an impact on quality is of paramount importance. A significant parameter that determines how pattern instances affect quality is the amount of artifacts (e.g., methods and attributes) that exist in the pattern-participant classes, which however, are not compliant to the original pattern definition (Izurieta and Bieman, 2013). Izurieta and Bieman (2013) named this phenomenon pattern grime and defined it as the degradation of design pattern instance due to buildup of unrelated artifacts in pattern instances. For example, grime can be introduced to a Template Method pattern instance by adding public methods that are not invoked inside the template method. Similarly, grime is introduced to a concrete state class of a State pattern instantiation when adding public methods other than those defined in the state interface. For both the aforementioned examples, such changes would lead to a reduced cohesion for the specific class, as well as reduced levels of source code understanding. Thus, the accumulation of grime can certainly be harmful to the quality of pattern instances and the overall system (Dale and Izurieta, 2014; Izurieta and Bieman, 2008, 2013).

Despite the potential effect of pattern grime on software quality, there is currently a lack of studies that investigate factors related to the accumulation of pattern grime. Therefore, in this study, we take a first step by exploring two types of factors related to the accumulation of pattern grime, i.e., different: ***projects***, ***pattern types***, ***developers***, and ***structural characteristics of pattern-participating classes*** (e.g., coupling and lack of cohesion). To this end, we performed an industrial case study, in which we analyzed five projects (that sum up to approx. 260,000 source lines of code) containing eight different GoF pattern types and implemented by 16 developers. To measure grime, we provide an open-source tool that automates the assessment of several pattern grime metrics. The outcome of this study sheds light on the factors that influence the accumulation of grime in pattern instances. Our results can be used by architects and designers to develop best practices while using design patterns, but also to monitor the evolution of grime and its respective effect on software quality.

The remainder of this chapter is organized as follows. In Section 6.2, we present work related to ours. The design of the case study is presented in Section 6.3, reported according to the guidelines of Runeson et al. (2012), i.e., the Linear Analytic Structure. In Sections 6.4 and 5 we present the results of our study and discuss the most important findings, respectively. We report on the identified threats to validity and actions taken to mitigate them in Section 6.6. Finally, we conclude the chapter in Section 6.7.

6.2 Related Work

In this section, we present work reporting on empirical studies on the evolution of grime and / or its relation to other characteristics of software pattern instances (e.g., quality attributes and metrics).

Izurieta and Bieman (2007) investigated the evolution of various design pattern instances from an open-source project to understand how patterns decay. The results suggest that the main reason for pattern instances to decay is due to grime. Schanz and Izurieta (2010) proposed a taxonomy for subtypes of modular grime (one type of grime) and performed a pilot study on nine pattern instances evolving throughout eight versions of one industrial software. The study validated the proposed classification, as well as suggested an increase of pattern grime. Regarding how the accumulation of grime correlates to other characteristics of the system, Griffith and Izurieta (2014) proposed a taxonomy for one type of grime, class grime, and performed a pilot study on randomly selected pattern instances from open-source projects to investigate the effects of class grime on design pattern understandability, and found this quality attribute to be negatively affected by the accumulation of class grime. In another study, Izurieta and Bieman (2008) evaluated the testability of design pattern instances from three different patterns and found that as grime is accumulated, other issues such as code smells also appears, and the testability of the pattern instances decreases.

Izurieta and Bieman (2013) studied the accumulation of grime and rot (another form of pattern decay, due to deterioration of the structural or functional integrity) during the evolution of pattern instances of three open-source systems. The study also correlated grime to testability, adaptability and pattern instability. The results are similar to those observed in the aforementioned studies, including increase of pattern grime and negative correlation with testability and adaptability. The authors also reported that they could not identify rot of pattern instances nor correlation between grime and pattern instability. Dale and Izurieta (2014) reported an experiment to study the correlation between three subtypes of modular grime and technical debt. Pattern instances of three example systems were used and modular grime was systematically injected in the instances. The results suggest that one subtype of modular grime (i.e., strength) is more strongly correlated to technical debt, in the sense that strong coupling (through class attributes) is correlated with stable grime, while weak coupling (other kinds of coupling) is correlated to increased technical debt.

In comparison to related work, we contribute the following: (a) we studied five industrial nontrivial projects that collectively provided 36,571 units of analysis (i.e., editions to pattern instances' source code, see Section 6.3). Therefore, we can compare our results with those obtained from the analysis of open-source projects and toy examples; (b) among other facets, we investigated how pattern grime is accumulated by different developers (16 in total), which has not been considered in previous studies; and (c) we studied the correlation between pattern grime and multiple structural metrics of pattern instances, which has not been thoroughly explored in previous studies.

6.3 Study Design

6.3.1 Objectives and Research Questions

The goal of this study, described using the Goal-Question-Metric (GQM) approach (van Solingen et al., 2002), is formulated as follows: *"**analyze** instances of GoF design patterns **for the purpose of** investigating the factors of project, pattern type, developers and structural characteristics of pattern participants **with respect to** their relationship with pattern grime, **from the point of view of** software designers **in the context of** industrial software development"*. Based on this goal, we defined the following research questions (RQs):

RQ$_1$: How does grime accumulate in pattern instances?

> **RQ$_{1.1}$:** Are there differences in accumulated grime among different projects?
>
> **RQ$_{1.2}$:** Are there differences in accumulated grime among different pattern types?
>
> **RQ$_{1.3}$:** Are there differences in accumulated grime among different developers?

RQ$_2$: Are structural characteristics of the pattern participants related to the accumulation of grime?

RQ$_1$ aims at assessing pattern grime within the five projects and exploring differences across three different factors: projects, types of pattern (e.g., Observer, Template Method) and developers. We chose these factors as they may potentially influence the accumulation of grime: the projects vary in requirements, design, size, scope etc. and may thus influence grime accumulation; the types of patterns exhibit different solutions and may allow or inhibit the accumulation of grime; the developers have diverse backgrounds and experience thus knowingly or inadvertently accumulating grime differently.

RQ$_2$ aims at investigating the relationship between levels of grime and a different type of factor: the structural characteristics of pattern-participating classes.

This helps to further understand the details of how the structure of the pattern itself relates to accumulating grime, and can thus inform best practices on the usage of design patterns.

6.3.2 Case Selection, Unit of Analysis, and Subjects

To answer the research questions, we designed an exploratory case study (Runeson et al., 2012), in which we analyze five industrial projects from one company in the domain of web and mobile applications development. These projects provided us with a diverse and comprehensive sample of developers (and projects) to investigate: one team of six people worked on three of the projects, while the other two projects were developed by two other teams (of five people each) independently. We selected an industrial case study, since there is a lack of empirical studies on pattern grime for such projects; *most of the previous studies have been performed on toy examples or open-source projects*.

As cases, we used the pattern instances of the explored projects. From each case, we recorded multiple units of analysis, based on the evolution of the specific instance. In particular, we recorded a unit of analysis for every change in the instance (i.e., pair of successive commits). We decided to focus on pairs of commits to *isolate and assess events (changes to pattern instances) performed by a single developer*. This allows to investigate developers as a potential factor influencing grime (see $\text{RQ}_{1.3}$).

6.3.3 Variables and Data Collection

To answer the research questions, we extracted four groups of variables: (1) Identification of units of analysis; (2) Pattern information; (3) Assessment of grime change; and (4) Assessment of structural change. Presented in Table 6.1, these variables are recorded for each unit of analysis (i.e., change to pattern instance). The entire process of identifying and measuring the units of analysis culminates in the creation of a dataset of all extracted variables for each unit. This dataset is recorded as a table in which the columns correspond to collected variables. We clarify that due to a non-disclosure agreement signed with the company in this case study we cannot share the created dataset. In the following we describe each group of variables.

Identification of Units of Analysis

This group regards the variables: *commit*, and *developer*. To identify every unit of analysis, we queried the git repository and extracted the author information and files that were changed for every commit. We ignored merge commits, as they do not provide new information regarding changes to files. In addition, we considered only changes to java classes that participate in a pattern instance.

Pattern Information

This group regards the variables: *instance-id*, and *pattern*. The collection of the pattern instances is a time-consuming task. For that reason, we used two tools, namely SSA (Similarity Score Analysis, v4.12) (Tsantalis et al., 2006) and SSA+ (v1.0.0), to detect pattern instances and performed a series of validations. In short, these tools allow us to detect pattern instances of 12 types: Adapter / Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State / Strategy, Template Method, and Visitor. We do not elaborate on the SSA tool nor its validation, since we used a similar design setup to detect patterns in Chapter 4 (see Section 4.3.3), in which all relevant information can be found. We note that we manually verified various (randomly selected) outputs. Regarding SSA+, it detects 10 extended pattern-participant classes, i.e., that participate in the pattern but it is not part of the main pattern structure (e.g., Concrete State / Strategy). The full list of detected extended pattern participants is available in the tool's website[1]. To validate SSA+, we also manually verified randomly selected outputs.

Assessment of Grime Change

This group regards the variables: *cg-**, *mg-**, and *og-**. According to Izurieta and Bieman (2007), there are three types of grime, which can be assessed independently: class, modular and organizational. To measure these types, we selected six metrics, as shown in Table 6.1. Each metric is estimated based on diverse design elements of pattern-participating classes: (a) class grime metrics are based on attributes and public methods; (b) modular grime metrics are based on incoming and outgoing dependencies; and (c) organization grime metrics are based on package and their dependencies. We clarify that we do not elaborate further on the metrics, which are calculated as described by Izurieta and Bieman (2007). We chose these metrics because they allow us to assess different aspects of each type of grime, and they were previously used and validated to assess pattern grime in nontrivial systems (Izurieta and Bieman, 2013).

To automate the calculation of the metrics, we created an open-source tool, spoon-pttgrime[2] (v0.1.0), available online as a public repository, which also contains further information on how the metrics are calculated. This tool takes as input Java source files of a project and an XML file describing the pattern instances in

[1] https://github.com/search-rug/ssap

[2] https://github.com/search-rug/spoon-pttgrime

the project (i.e., the output from SSA+). For each pattern instance, spoon-pttgrime calculates the six aforementioned metrics by querying the project's AST using the Spoon library (Pawlak et al., 2016). To validate the tool, we manually verified the output for 20 pattern instances (randomly selected) over five consecutive commits. Bugs were fixed and additional verification rounds showed no errors. As we are interested in assessing the change of grime in pattern instances for a pair of commits, we subtracted the grime estimation at the current commit (identified by the unit of analysis) from the estimation of the immediate previous commit (i.e., its parent).

Assessment of Structural Change

This group regards the variables *s-**. To assess structural change, we selected three sets of metrics, proposed by Chidamber and Kemerer (1994), Li and Henry (1993), and Bansiya and Davis (2002), accounting for the 21 metrics presented in Table 6.1. We selected these metrics because they allow us to investigate many characteristics of the structure of pattern participants, and because they are well-known by both researchers and practitioners. To calculate the metrics, we used Percerons Client, i.e., a tool developed in our research group that automates the assessment of these metrics for Java classes. Percerons is a software engineering platform (Ampatzoglou, Michou and Stamelos, 2013) to facilitate empirical research in software engineering and has been used for similar reasons in (Griffith and Izurieta, 2014) and (Alhusain et al., 2013).

6.3.4 Analysis Procedure

To answer RQ$_1$, we analyze the descriptive statistics of the variables for unit identification, pattern information, and assessment of grime change. As our study comprises several projects / subjects and encompasses several GoF patterns, we derive data subsets, so as to group the units of analysis based on the different analyzed factors (i.e., project, pattern and developer). When necessary we also perform linear regressions and parametric or non-parametric tests (Field, 2013) in order to devise trends and test differences between groups. To answer RQ$_2$, we first analyze whether the distribution of all measurements for each metric is normally distributed. If true, we can select the Pearson correlation method (Field, 2013), otherwise the Spearman's rank correlation method (Field, 2013). For each pattern grime metric, we perform the analysis as follows: first we calculate the correlation between the grime metric and all structural metrics; next, we identify strong correlations ($>$ 0.8) that are statistically significant, and discuss the results from the perspective of grime accumulation.

Table 6.1: List of collected variables

Group	Variable	Description
Unit Information	project	Project from which the pattern instance was extracted
	commit	Hash of the commit in the git repository
	dev	Developer author of the commit
Pattern Information	inst_id	ID of the pattern instance the class belongs to
	pattern	GoF design pattern of the instance
Assessment of Grime Change	cg-npm	Difference in the total number of alien public methods in all classes of the pattern instance (Class grime)
	cg-na	Difference in the total number of alien attributes in all classes of the pattern instance (Class grime)
	mg-ca	Difference in the pattern instance afferent coupling (Modular grime)
	mg-ce	Difference in the pattern instance efferent coupling (Modular grime)
	og-np	Difference in the number of packages within the pattern instance (Organizational grime)
	og-ca	Difference in the fan-in at the package level (Organizational grime)
Assessment of Structural Change	s-wmc	Difference in the average weighted methods per class
	s-dit	Difference in the maximum depth of inheritance tree
	s-noc	Difference in the average number of children
	s-cbo	Difference in the average coupling between object classes
	s-rfc	Difference in the average response for a class
	s-lcom	Difference in the average lack of cohesion in methods
	s-nom	Difference in the average number of methods
	s-mpc	Difference in the average message-passing coupling
	s-dac	Difference in the average data abstraction coupling
	s-size1	Difference in the lines of code
	s-size2	Difference in the number of properties
	s-dsc	Difference in the design size in classes
	s-noh	Difference in the number of hierarchies
	s-ana	Difference in the average number of ancestors
	s-dam	Difference in the data access metric
	s-camc	Difference in the cohesion among methods of class
	s-moa	Difference in the measure of aggregation
	s-mfa	Difference in the measure of functional abstraction
	s-nop	Difference in the number of polymorphic methods
	s-cis	Difference in the class interface size
	s-fan-in	Difference in the afferent couplings

6.4 Results

In this section, first we briefly describe the collected data and subsequently address each research question independently. We note that we investigated six metrics for pattern grime, and therefore report the results for all metrics and highlight findings independently for each one, when this is relevant. We collected a total of 1,422 commits, from the five studied projects, that include the creation or modification of pattern-participating classes. From these commits, 94% (i.e., 1,341) include mod-

Table 6.2: Amount of grime accumulated per commit

Metric	Minimum	Maximum	Mean	Std. Deviation
cg-npm	-1.00	15.00	0.28	0.64
cg-na	-1.50	9.50	0.12	0.45
mg-ca	-3.75	44.00	0.21	1.18
mg-ce	-10.00	85.00	1.61	4.53
og-np	-0.25	2.00	0.02	0.14
og-ca	-2.00	35.00	0.14	1.13

ifications to one or more pattern instances. We identified 2,349 pattern instances of eight different GoF patterns: (Object) Adapter-Command, Factory Method, Observer, Singleton, State-Strategy, and Template Method. Each pattern instance was created and then modified up to 178 times (i.e., the maximum number of modifications for a single instance). From the total number of pattern instances, 87% (i.e., 2,039) were modified at least once after being created, and 64% (i.e., 1,500) at least five times. The data collection resulted in the identification of 36,571 units of analysis (i.e., creation / modification of a pattern instance in a commit).

6.4.1 RQ$_1$ - Accumulation of Grime

To study the differences in accumulated grime among different projects, types of patterns and developers, we first present how the assessed pattern grime metrics change within the instances' evolution. Table 6.2 shows the following descriptive statistics for the six metrics (previously presented in Table 6.1): minimum and maximum values, mean value among all units of analysis and standard deviation (i.e., how much measurements vary from the mean value). Based on the Table 6.2, we notice that grime can either reduce (i.e., negative measurement) or increase. However, the data suggest that on average, grime in pattern instances tends to increase during the instance's evolution. Another observation is that the number of packages in a pattern instance (*og-np*) seems to be the grime indicator that is less likely to change, which is a probable sign of common design practices. Moreover, despite considerably higher maximum values, we notice that the measurements are consistently close to the mean, since the standard deviation is not much higher than the mean (especially compared to maximum values).

Next, we are interested in investigating how grime accumulated in different projects (RQ$_{1.1}$). Figure 6.1 depicts this information for the six metrics. P1 is the project with most collected commits (605), while P5 provided the least commits (76). The y-axis represents the mean amount of grime accumulated per modified instance in a given commit. The x-axis represents consecutive commits. We note that the x-

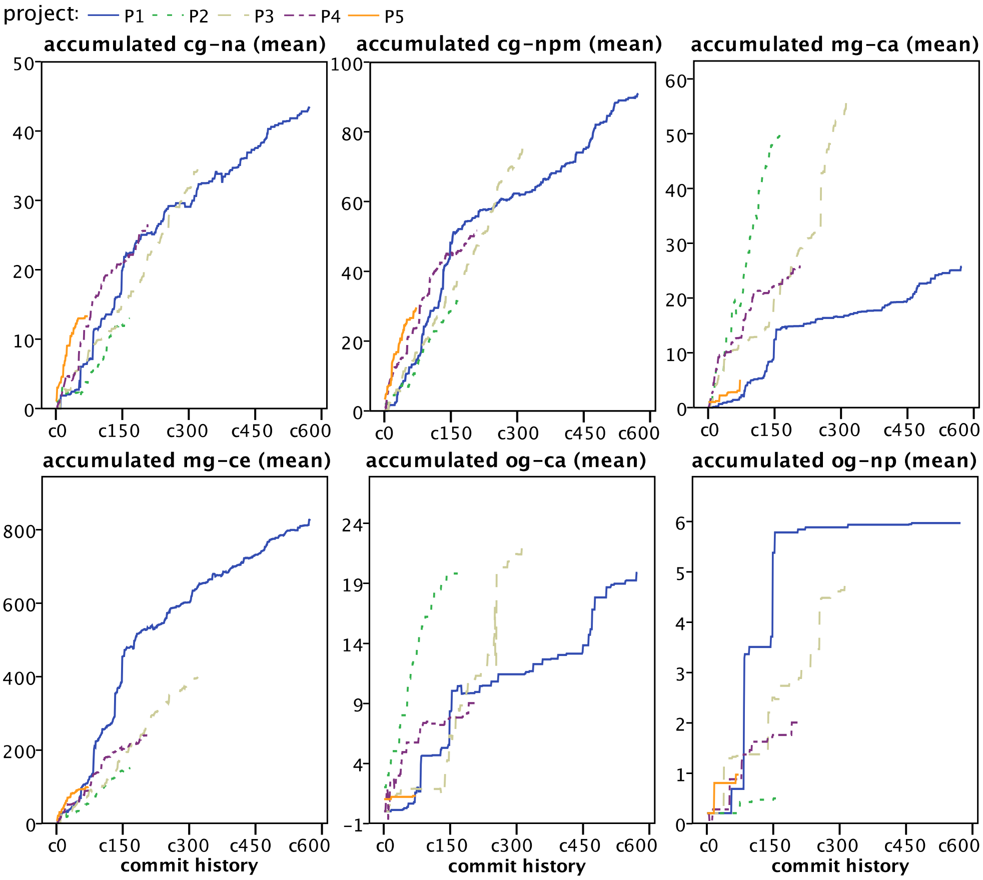


Figure 6.1: Accumulation of grime per project for each grime metric

axis does not represent the full history of commits. Our goal is to investigate the evolution of pattern instances and, thus, we considered only commits that include the modification of pattern-participant classes. By inspecting Figure 6.1, we observe that every project individually reflects the trend of the population, i.e., pattern grime linearly increases during the project evolution. To verify this, we performed linear regression for every pair <metric, project> and assessed how well the calculated equation fits the data.

In Table 6.3, we present the results, which are all statistically significant. We notice that the vast majority of the equations are powerful descriptors, since R2 (i.e., how close the data fit the regression line) is close to 1. The exceptions are the tuples <*og-np*, *P1*>, <*og-np*, *P5*>, and <*og-ca*, *P5*>, which regard metrics of organization

Table 6.3: Linear regression of pattern grime accumulation per project

Metric	Project	Equation	R^2	Metric	Project	Equation	R^2
cg-npm	P1	13.91 + 0.15x	0.91	cg-na	P1	5.47 + 0.07x	0.93
	P2	-0.28 + 0.19x	0.99		P2	-0.59 + 0.08x	0.92
	P3	-1.79 + 0.24x	0.99		P3	-0.51 + 0.11x	0.99
	P4	7.44 + 0.24x	0.95		P4	1.89 + 0.13x	0.95
	P5	5.32 + 0.37x	0.95		P5	3.44 + 0.17x	0.89
mg-ca	P1	2.27 + 0.04x	0.90	mg-ce	P1	129.84 + 1.40x	0.89
	P2	-1.72 + 0.34x	0.99		P2	-9.04 + 1.00x	0.99
	P3	-2.24 + 0.17x	0.93		P3	-15.42 + 1.34x	0.99
	P4	5.68 + 0.11x	0.92		P4	15.36 + 1.21x	0.96
	P5	0.71 + 0.04x	0.87		P5	26.47 + 1.20x	0.89
og-np	P1	2.00 + 0.01x	0.58	og-ca	P1	1.09 + 0.03x	0.92
	P2	-0.06 + 0.00x	0.82		P2	3.11 + 0.12x	0.95
	P3	-0.20 + 0.02x	0.96		P3	-3.86 + 0.08x	0.90
	P4	-0.02 + 0.01x	0.89		P4	2.61 + 0.03x	0.81
	P5	0.12 + 0.01x	0.61		P5	1.04 + 0.00x	0.64

grime. This is due to the drastic change in the accumulated grime observed for these tuples, which may reflect systematic changes in the design (e.g., package renaming).

Further, we analyzed the dataset regarding different GoF patterns ($RQ_{1.2}$). In Table 6.4, we show the descriptive statistics for each metric and identified pattern. We clarify that we do not report the results for the Observer and Template Method patterns, as the number of units of analysis for them is negligible (18 and 5, respectively). The results suggest that different patterns are subject to different levels of grime. For example, it seems that little grime is accumulated in instances of Singleton after their creation, whilst instances of Factory Method tend to accumulate the most amount of grime. To statistically investigate the difference between patterns, we performed pairwise comparisons (Mann-Whitney tests), which are reported within the supplementary material (Feitosa, Avgeriou, Ampatzoglou and Nakagawa, 2017a). The results showed that the differences in most comparisons (86% of the 36 tests) is statistically significant, thus supporting our findings.

The last facet we investigated was how different developers accumulate grime ($RQ_{1.3}$). We clarify that we do not report the complete descriptive statistics for each metric and developer, which are available in the supplementary material (Feitosa, Avgeriou, Ampatzoglou and Nakagawa, 2017a). In Table 6.5, we present the number of pattern instances maintained by the 16 developers, changes to pattern instances and mean value of the grime metrics. By analyzing the results, we notice that some developers seem to consistently accumulate more grime than others (e.g., D7, D8 and D9), or less grime than others (e.g., D1 and D3), with respect to most metrics. Furthermore, we can observe that developers that changed pattern instances more

Table 6.4: Amount of grime accumulated per pattern

Metric	Pattern	Num. of instances	Num. of changes	Min.	Max.	Mean	Std. Deviation
cg-na	Adapter-Command	770	13,225	-3.00	17.00	0.12	0.53
	Factory Method	61	776	-3.42	13.00	0.15	0.78
	Singleton	83	281	-1.00	1.00	0.01	0.16
	State-Strategy	1121	19,937	-4.00	13.00	0.10	0.44
cg-npm	Adapter-Command	770	13,225	-3.00	26.00	0.21	0.77
	Factory Method	61	776	-7.58	21.67	0.35	1.42
	Singleton	83	281	-2.00	4.00	0.06	0.44
	State-Strategy	1121	19,937	-8.00	21.33	0.21	0.80
mg-ca	Adapter-Command	770	13,225	-2.00	44.00	0.08	0.89
	Factory Method	61	776	-7.00	102.00	0.59	4.15
	Singleton	83	281	-1.00	7.00	0.49	0.96
	State-Strategy	1121	19,937	-15.00	44.00	0.12	0.87
mg-ce	Adapter-Command	770	13,225	-20.00	197.00	1.19	5.60
	Factory Method	61	776	-13.00	60.00	1.44	4.40
	Singleton	83	281	-4.00	17.00	0.47	1.85
	State-Strategy	1121	19,937	-30.00	159.00	1.23	5.66
og-ca	Adapter-Command	770	13,225	-2.00	35.00	0.06	0.75
	Factory Method	61	776	-36.00	36.00	0.17	2.32
	Singleton	83	281	-1.00	27.00	0.41	2.19
	State-Strategy	1121	19,937	-6.00	34.00	0.06	0.62
og-np	Adapter-Command	770	13,225	0.00	2.00	0.01	0.13
	Factory Method	61	776	-1.00	3.00	0.03	0.21
	Singleton	83	281	0.00	1.00	0.01	0.10
	State-Strategy	1121	19,937	-1.00	3.00	0.01	0.15

often tend to accumulate less grime. Seeking to support our observations statistically, we compared pairs of developers based on every metric using the Mann-Whitney test. By observing the findings of the test, we suggest that 73% of the 396 tests are statistically significant, and that the non-significant tests regard mostly the number of packages (*og-np*). Detailed results are reported on the supplementary material (Feitosa, Avgeriou, Ampatzoglou and Nakagawa, 2017a).

Summarizing the results for RQ_1, pattern grime: (a) *is likely to increase linearly over system evolution*; (b) *grows similarly across different projects*; (c) *accumulates at different paces depending on the pattern type and the individual developer*. The interpretation of all findings reported in this section, as well as their implications to researchers and practitioners are discussed in Section 6.5.

Table 6.5: Average amount of grime accumulated per developer

Developer	Num. of instances	Num. of changes	cg-na	cg-npm	mg-ca	mg-ce	og-ca	og-np
D1	465	7,525	0.08	0.17	0.04	0.93	0.04	0.00
D2	1,132	6,232	0.12	0.34	0.20	1.25	0.05	0.00
D3	549	5,232	0.07	0.07	0.04	0.85	0.01	0.00
D4	837	5,141	0.10	0.14	0.13	0.96	0.04	0.01
D5	335	3,442	0.10	0.23	0.04	1.35	0.02	0.02
D6	469	1,554	0.13	0.24	0.14	2.28	0.24	0.00
D7	292	1,406	0.17	0.29	0.23	2.54	0.19	0.05
D8	326	1,346	0.20	0.26	0.21	1.72	0.18	0.02
D9	161	697	0.13	0.38	0.27	1.68	0.20	0.02
D10	225	636	0.07	0.37	0.24	1.05	0.27	0.01
D11	233	515	0.01	0.19	0.34	0.37	0.06	0.00
D12	170	431	0.23	0.28	0.06	1.89	0.00	0.00
D13	41	56	0.79	1.64	0.89	8.04	0.29	0.21
D14	13	17	0.00	0.00	0.00	2.06	0.00	0.00
D15	3	8	0.00	0.03	-0.25	0.00	0.38	0.00
D16	2	4	0.00	0.00	0.00	0.75	0.00	0.00

6.4.2 RQ$_2$ - Structural Characteristics and Pattern Grime

To assess the correlation between pattern grime and structural metrics, we first verified whether all measurements for each metric are normally distributed. We found that not all are normally distributed and, thus, we decided to use a non-parametric method to study the metrics: Spearman's rank correlation. All assessed correlations are presented in Table 6.6, and are all statistically significant.

Regarding the metrics for ***class grime***, we make the following observations. The metric *cg-npm* is strongly correlated (> 0.8) to *s-wmc*, *s-nom*, and *s-cis*. This may be an indication that when many methods are added to pattern-related classes it is common that a large portion of them are not related to the pattern realization. The metric *cg-na* is strongly correlated to *s-dac* and *s-moa*. This may be an indication that a considerable part of the pattern instance coupling is coming from added attributes. This may not be necessarily an alert for bad design, but it rather depends on how many attributes are added. Regarding ***modular grime***, we notice that the metric *mg-ca* is strongly correlated to *s-fan*-in only, which is a metric that is similar to *mg-ca*, but at class level. This suggests that most of the pattern instance afferent coupling comes from regular afferent coupling of the pattern participants. This may indicate that pattern instances tend to evolve by adding functionality not related to the pattern. The metric *mg-ce* is not strongly correlated to any metrics, whereas the strongest correlations are with *s-rfc* and *s-cbo*. These moderate correlations also indicate that, to some extent, the introduction of coupling in pattern instances is also introducing grime. Finally, regarding ***organizational grime***, the metric *og-np* is

Table 6.6: Correlation between grime and structural metrics

	cg-npm	**cg-na**	**mg-ca**	**mg-ce**	**og-np**	**og-ca**
s-wmc	*0.86*	0.44	0.38	0.48	0.46	0.38
s-dit	0.44	0.53	0.55	0.43	*0.99*	0.71
s-noc	0.45	0.52	0.60	0.41	*0.99*	0.73
s-cbo	0.47	0.67	0.50	0.73	0.46	0.41
s-rfc	0.65	0.54	0.31	0.65	0.41	0.33
s-lcom	0.70	0.35	0.32	0.36	0.35	0.31
s-nom	*0.86*	0.44	0.38	0.48	0.46	0.38
s-mpc	0.43	0.44	0.22	0.55	0.35	0.27
s-dac	0.36	*0.87*	0.34	0.59	0.56	0.42
s-size1	0.69	0.53	0.31	0.58	0.41	0.35
s-size2	0.79	0.65	0.38	0.58	0.44	0.38
s-dsc	0.44	0.53	0.56	0.43	*0.99*	0.70
s-noh	0.38	0.43	0.50	0.35	0.77	0.60
s-ana	0.45	0.53	0.51	0.43	*0.93*	0.66
s-dam	0.34	0.56	0.42	0.43	0.76	0.54
s-camc	-0.14	0.16	0.17	0.03	0.45	0.30
s-moa	0.37	*0.90*	0.36	0.61	0.58	0.44
s-mfa	0.03	0.11	0.03	0.08	0.21	0.07
s-nop	0.71	0.35	0.48	0.34	0.51	0.43
s-cis	*0.97*	0.41	0.41	0.44	0.48	0.41
s-fan-in	0.46	0.42	*0.90*	0.36	0.70	0.63

strongly correlated to *s-dit*, *s-noc*, *s-dsc*, and *s-ana*. Despite the strong correlations, this finding may be inconclusive as *og-np* rarely changes and this is probably the main reason for such high correlations. Finally, the metric *og-ca* is not strongly correlated to any metrics, whereas the strongest correlations are with *s-noc*, *s-dit* and *s-dsc*. These moderate correlations may indicate that, to some extent, the addition of new classes to the pattern instance is to serve a new purpose, i.e., serve a class not served before.

6.5 Discussion

In this section, we discuss the findings of our case study, as well as their implications. First, we interpret our findings, elaborating on explanations and consequences for the observed results. Next, we present how our findings can benefit both researchers and practitioners.

6.5.1 Interpretation of Results

In Section 6.4, we reported the raw findings of our case study, whereas in this section, we interpret them and compare them against the state-of-the-art. First, regarding the evolution of grime, we observed that ***pattern grime is constantly increasing along the versions of a system***. This result can be considered intuitive as it aligns with Lehman's laws on software evolution: software quality deteriorates as the software becomes larger and more complex. However, there is an interesting aspect of this finding: the amount of grime that is accumulated in pattern instances clearly suggests that pattern-participating classes are not "closed to modifications", in the sense that they are continuously "polluted" with artifacts (e.g., methods, dependencies, etc.) that are not pattern-related. This pollution potentially influences how the application of design patterns affects quality attribute indicators of a system. Thus, pattern instantiation does not have a constant effect on quality, but it changes along evolution. This finding is in accordance to the literature, which suggests that the effect of GoF design patterns on product quality is not uniform along different pattern instances (Ampatzoglou, Charalampidou and Stamelos, 2013b), and aligns with results of studies with similar setups (Dale and Izurieta, 2014; Izurieta and Bieman, 2008, 2013). In particular, Izurieta and Bieman (2013) used the same pattern grime metrics and investigated some patterns in common (e.g., Singleton and Factory Method), but by inspecting open-source systems. The results of both studies agree on the increase of grime metrics.

Regarding the three parameters that were investigated in RQ_1 (i.e., grime in different projects, patterns, and developers), the results suggested that ***the levels of grime are similar at the different projects of the same company despite the little overlap of developers among projects***. This outcome can be potentially explained by the fact that the developers were guided by the same practices, since they usually follow the same company process. Nevertheless, this finding needs to be further validated through a follow-up study conducted in different companies. Another finding is that ***the levels of grime are different among pattern types***, which complies with the literature suggesting that different patterns have different effect on quality attributes (e.g., (Ampatzoglou et al., 2015) on stability). In particular, we noticed that instances of the Singleton pattern are the least likely to accumulate grime, whereas instances of Factory Method are the most grime-prone. The acknowledgment of certain good practices (e.g., avoid creation of God Classes) can lead to more "grime-free" Singleton instances. However, if not careful, developers may enlarge the responsibility of classes unnecessarily, as observed with Factory Method instances, which may include methods that suffer from the Feature Envy, Shotgun Surgery, or Divergent Change smells. Therefore, *we suggest monitoring pattern grime to identify spots of bad quality in the system*. Such a practice may support the preservation of quality indicators (such as understandability and testability) at acceptable levels and thus increase productivity. Moreover, in comparison with related work, Izurieta and Bieman (2013) also show that Singleton pattern instances tend to accumulate less grime, whereas on the contrary Factory Method instances tend to accumulate grime faster than other investigated patterns. This observation further supports that open-source and industrial systems have similarities with regards to the accumulation of pattern grime.

From the last investigated parameter, we found that ***the levels of grime also differ among developers***. Their tendency to accumulate grime likely depends on diverse factors. In particular, varied levels of programming skills, knowledge of the system and of GoF patterns can explain the different tendency to accumulate grime. This finding supports the belief that personalized quality assessments are required in industry (Amanatidis et al., 2017). Furthermore, we observed that developers that performed more changes are related to lower levels of accumulated grime, suggesting that most tasks (resulting in more changes) are assigned to more experienced developers, inclined to accumulate less grime. *We suggest using such information about developers in order to improve the software development proces*s. For example, since our industrial partner use agile methodologies, such information can be considered in daily Scrum meetings in which issues are assigned to individual developers. The personalization of software development and the effect on human factors in the quality of the software have been extensively studied in the last years, underlying the importance of such strategies.

Finally, regarding the relation of structural metrics with grime metrics, the results point out that some of the most established structural quality metrics are related to the grime metrics. For example, the fan-in metric is at least moderately correlated to all grime metrics. This finding may be explained by the fact that pattern grime is calculated at the detailed-design level. Since class dependencies consist one of the main elements of object-oriented design, it is intuitive to expect the obtained correlations, e.g., between two metrics that are calculated based on class dependencies. However, we note that the strength of the correlations varies among pattern instances, which shows that structural metrics can be adequate predictors of grime accumulation.

6.5.2 Implications to Researchers and Practitioners

Researchers can benefit from our results from several perspectives. We presented a thorough exploration of the accumulation of pattern grime and we identified several factors that influence how pattern grime grows during the evolution of pattern instances. This exploration not only reinforces the importance of investigating pattern grime, but also suggests several opportunities of future work, e.g., investigate characteristics of developers that tend to accumulate grime. In addition, the identified correlations between pattern grime and structural metrics help on understanding how pattern grime is introduced, as well as open further possibilities to investigate other relevant aspects of software systems and processes, for example technical debt.

We also foresee benefits of our results to practitioners. Because we investigated five nontrivial projects, our findings can help practitioners improve best practices on the usage of design patterns, e.g., by warning developers to avoid accumulating grime on Singletons pattern instances. Moreover, the metrics and correlations that we present can be considered in processes for monitoring the evolution of the software systems, e.g., high levels of fan-in in pattern-participating classes may indicate that considerable grime is being inserted.

6.6 Threats to Validity

In this section, we discuss threats to construct validity (i.e., if the studied phenomenon is connected to the set objectives), reliability (if the study can be replicated), and external validity (i.e., generalizability). We do not analyze internal validity, as we do not try to establish causal relationships.

Concerning construct validity, the tool SSA is limited by its precision and recall: false positives and negatives may bias the presented results. However, to the best of our knowledge the used tool is among the most reputed in the community, and has adequate performance (see Section 6.3). For mitigating this threat, we verified its precision and recall manually by checking 30 random pattern instances for each GoF pattern that was detected (i.e., over 100 instances in total), which were all successful. Additionally, regarding SSA+ and spoon-pttgrime, we acknowledge that the tools may have bugs. However, we verified over 50 random outputs of each tool and, to the best of our knowledge, no bugs were found.

In order to mitigate reliability threats, two different researchers performed the collection and analysis, double-checking sample outputs. Besides that, we acknowledge that non-disclosure agreements do not allow us to share the collected dataset. However, all used tools are freely available and replication studies can be carried out. Finally, the external validity of our study is threatened by the fact that we analyzed projects of the same company, thus, our findings may not be generalizable to other projects nor teams. However, our results relate to those obtained in other studies with similar setup, e.g., we expected modular grime to be the main contributor for the pattern grime, and we found *mg-ce* to clearly grow at a faster pace. In addition, our results are bounded by our study design. Adding other GoF patterns, pattern grime metrics, or structural metrics could lead to adjustments in our findings.

6.7 Conclusion

In this chapter, we presented an exploratory case study on how grime accumulates in pattern instances and its correlation to structural characteristics of the pattern participants. To this end, we investigated the evolution of 2,349 pattern instances of eight patterns, assessing six grime metrics of three types of grime (class, modular and organization), as well as 21 structural metrics. We explored how grime is distributed according to: (a) projects, pattern types, and developers, and (b) structural characteristics of pattern-participating classes. The results suggest that pattern grime tends to increase linearly, it is likely independent of project but depends on pattern type and developer. Moreover, we identified a series of correlations between metrics for pattern grime and structural characteristics, e.g., the coupling added to pattern participants tend to also introduce grime.

The accumulation of pattern grime observed in this study indicated that this phenomenon could have undesired effects on pattern instances. Therefore, the next chapter address this concern by reporting an investigation on whether there is a relation between the three forms of pattern grime (i.e., organizational, modular, and class) and the levels of three quality attributes addressed in this dissertation, namely performance, security and correctness.

Based on:

Daniel Feitosa, Apostolos Ampatzoglou Paris Avgeriou, and Elisa Y. Nakagawa, (2018) *"Correlating Pattern Grime and Quality Attributes,"* in IEEE Access, vol. 6, pp. 23065–23078, DOI:10.1109/ACCESS.2018.2829895

Chapter 7

Correlating Pattern Grime and Quality Attributes

Abstract

One important parameter that relates to the effect of patterns on quality is the deterioration of pattern instances due to the buildup of artifacts unrelated to the pattern structure, a phenomenon named pattern grime. In this chapter, we investigate the relation between pattern grime and three qualities, namely performance, security and correctness. To this end, we conducted a case study with five industrial projects implemented by 16 developers. Our findings suggest a correlation between the accumulation of grime and decreased levels of performance, security, and correctness. Moreover, factors such as the project itself, pattern type and the developer can influence this relation. The obtained results can benefit both researchers and practitioners, as we provide evidence on the accumulation of pattern grime and its correlation to performance, security and correctness, and how different factors affect these correlations.

7.1 Introduction

The effect of patterns on software quality is not uniform, but it depends on a number of parameters (Ampatzoglou, Charalampidou and Stamelos, 2013b). Several works have concluded that a pattern can be beneficial in some cases and harmful in others, with respect to a specific quality attribute, by studying the structural characteristics of patterns, such as the number of pattern participating classes, number of methods, etc (Charalampidou et al., 2017; Hsueh et al., 2008; Huston, 2001; Muraki and Saeki, 2002). In Chapter 6, we report on one phenomenon that may influence how beneficial pattern instances may be to quality attributes, namely pattern grime (Izurieta and Bieman, 2013). In general, accumulating pattern grime contributes to the degradation of quality in pattern instances (Dale and Izurieta, 2014; Izurieta and Bieman, 2008, 2013). Given the high percentage of class participation in GoF patterns (up to 65%), the effects of ever-growing grime can be detrimental to the overall quality of those systems.

Despite ongoing research on identifying the impact of pattern grime on software quality (Griffith and Izurieta, 2014; Izurieta and Bieman, 2008, 2013), there are still three shortcomings. First, only a few quality attributes have been addressed so far, namely testability, adaptability and understandability. Second, despite the existence of industrial case studies examining how pattern grime accumulates (Schanz and Izurieta, 2010; Feitosa, Avgeriou, Ampatzoglou and Nakagawa, 2017b), there is a lack of industrial studies regarding how the accumulation of grime relate to levels of quality attributes; the existing studies are limited to open source software. Finally, even these studies on open source have limited depth regarding the investigation of factors that contribute to this relation between grime and qualities. For example, developers with different levels of involvement in a project may accumulate grime differently. Identifying the factors related to higher levels of grime can improve the impact of design patterns on quality, as well as to a more adequate allocation of resources in a project.

In this chapter, we address the aforementioned shortcomings through an industrial case study that examines the relation between the accumulation of pattern grime and quality. The study was designed according to the guidelines of Runeson et al. (2012), reported based on the Linear Analytic Structure (Runeson et al., 2012). Thus, we offer three advancements compared to the state of the art (which is further elaborated in Section 7.2). First, we focus on three qualities that have not been studied: performance, correctness and security. Second, we consider five industrial software systems for our investigation, instead of open source. Finally, we investigate three factors that may influence the underlying relations:

- the *projects* under development have several characteristics such as application domain and type of systems (e.g., user application, library), which may influence the usage of patterns and development practices. Studies have already shown that projects can accumulate pattern grime differently (Izurieta and Bieman, 2013; Feitosa, Avgeriou, Ampatzoglou and Nakagawa, 2017b). Thus, we seek to investigate if this may reflect on the relation between grime and levels of quality as well;

- the *types of pattern* (e.g., Template Method, Singleton, etc.) have also been pointed out as a factor on how pattern grime is accumulated (Izurieta and Bieman, 2013; Schanz and Izurieta, 2010; Feitosa, Avgeriou, Ampatzoglou and Nakagawa, 2017b). The different structural and behavioral characteristics of patterns may also be related to how exactly quality is affected; and

- the *developers* often have different traits such as background and experience, which may affect their behavior and productivity (Amanatidis et al., 2017). Besides, as reported in Chapter 6, developers have also been found to accumulate grime differently, which corroborates the relevance of also investigating if this factor relates to a varying level of quality.

The study is executed based on the commits performed by 16 developers during the implementation of the same five projects investigated in Chapter 6. Similar to Chapter 4, the studied qualities are assessed through the number of violations of various coding practices, each one mapped to one of the qualities (for more details see Section 7.3.3, Step 4).

The remainder of this chapter is organized as follows. In Section 7.2, we present related work. The design of our case study is described in Section 7.3. In Sections 7.4 and 7.5, we report on our results and discuss the most important findings. We present the identified threats to validity in Section 7.6, together with actions taken to mitigate them. Finally, we conclude the chapter in Section 7.7.

7.2 Related Work

In this section, we focus on the terminology related to pattern grime, and address empirical studies that investigate the relation between accumulation of grime and quality attributes.

7.2.1 Design Patterns Grime and Quality Attributes

Pattern grime concerns the degradation of pattern instances without breaking down the original structure on the pattern definition (Izurieta and Bieman, 2013). This degradation occurs through the addition of associations that do not comply with patterns' responsibilities (e.g., addition of a public method that is not in the definition), which can accumulate along the evolution of the instance and obscure their design (Izurieta and Bieman, 2008). Izurieta and Bieman (2007) established that the added associations can be assessed from three base perspectives, i.e., there are three forms of pattern grime. *Class grime* regards class-related elements (e.g., number of attributes, methods, or children) that are unrelated to the role of a class in the pattern instance. *Modular grime* regards relationships (e.g., dependency, generalization) between classes of the pattern instance and other classes, which are not predicted in the definition of the pattern. *Organizational grime* regards how pattern-participant classes are distributed into packages and/or namespaces. This threefold classification was further refined by Schanz and Izurieta (2010), who provided a taxonomy of subtypes for modular grime, and by Griffith and Izurieta (2014), with a taxonomy of subtypes for class grime.

Regarding the relation between the accumulation of pattern grime and the levels of quality attributes, we identified three empirical studies. Izurieta and Bieman (2008) investigated how grime is associated with the testability of pattern instances. For that, they considered instances of Singleton, Visitor and State patterns obtained from an open-source system and assessed their testability by the number of test cases necessary to cover them. By analyzing the testability against the accumulation of modular grime, Izurieta and Bieman found that testability decreases (i.e., more test cases are needed) as grime accumulates. Moreover, other issues such as the appearance of code smells also aggravate. In a complementary study, Izurieta and Bieman (2013) explored how pattern grime affects the testability and adaptability (measured by pattern instability) of instances from three open-source systems. They examined all three forms of pattern grime (i.e., class, modular and organizational) and again observed a negative impact. Both testability and adaptability decreased with the accumulation of grime, although the results regarding organizational grime were inconclusive due to lack of more data. Finally, Griffith and Izurieta (2014) investigated how the understandability of pattern instances changes due to the accumulation of grime. To this end, they focused on class grime and randomly collected pattern instances from a database of open-source components (Ampatzoglou, Michou and Stamelos, 2013). By correlating the accumulated grime with understandability (assessed according to the QMOOD quality model (Bansiya and Davis, 2002)), they found that this quality attribute is also affected negatively.

7.2.2 Comparison to State of Research

In Table 7.1, we compare the main parameters that differentiate our study from related work. In particular, we emphasize that: (a) we investigated three quality attributes (i.e., performance, security and correctness) that have not been addressed in this context; (b) we studied five industrial nontrivial projects (in contrast to open-source ones) that collectively provided 36,571 units of analysis (i.e., modifications to the source code of pattern instances, see Section 7.3); and (c) we investigated factors that, although have been explored with regards to the accumulation of grime, have not still been examined with regards to the relation between grime and quality attributes.

Table 7.1: Comparison with related work

Ref.	**Context**	**Projects**	**Patterns**	**Instances**	**Forms of grime**	**Quality attributes**	**Factors**
Izurieta and Bieman (2008)	open-source	1	3	2	modular	testability	pattern
Izurieta and Bieman (2013)	open-source	3	7	“small number”	class, modular and organizational	testability and adaptability	lines of code
Griffith and Izurieta (2014)	open-source	not clear	16	not clear	class	understandability	none
This study	industrial	5	9	2,329	class, modular and organizational	correctness, performance and security	project, pattern, developer

7.3 Study Design

In this section, we present the protocol of our case study, designed according to the guidelines of Runeson et al. (2012), reported based on the Linear Analytic Structure (Runeson et al., 2012).

7.3.1 Objectives and Research Questions

We formulated the goal of this study using the Goal-Question-Metric (GQM) approach (van Solingen et al., 2002), as follows: *“**analyze** the accumulation of grime on GoF pattern instances **for the purpose of** evaluation **with respect to** its relationship with the levels of performance, security and correctness, **from the point of view of** software designers **in the context of** industrial software development”*. To accomplish this goal, we proposed three research questions (RQs), which are elaborated as follows.

RQ$_1$: Does the accumulation of pattern grime correlate with changes in the investigated quality attributes?

RQ$_{1.1}$: Is a correlation observed for class grime?

RQ$_{1.2}$: Is a correlation observed for modular grime?

RQ$_{1.3}$: Is a correlation observed for organizational grime?

RQ$_1$ aims at acquiring initial evidence of the relationship between the accumulation of pattern grime and changes in the levels of correctness, performance and security. We note that we address each quality attribute in isolation. To more comprehensively answer this question, we investigated all three forms of grime proposed by Izurieta and Bieman (2007), i.e., class, modular and organizational grime.

RQ$_2$: Which factors affect the aforementioned relation?

RQ$_{2.1}$: Does it vary for different projects?

RQ$_{2.2}$: Does it vary for different patterns?

RQ$_{2.3}$: Does it vary for different developers?

Next, we extend our analysis to factors that may influence the relation between pattern grime and quality attributes. In this study, we examined three factors. First, we investigated if the correlation between grime and quality attributes differs for different projects (RQ$_{2.1}$). Second, we were interested in answering this question, but for different patterns (RQ$_{2.2}$). Although these two factors were briefly addressed in related work, they have not been empirically explored so far. To complement the analysis, we also investigated whether the relationship varies depending on the developer (RQ$_{2.3}$), in the sense that the expertise or experience of developers may be reflected in the accumulated grime and/or quality attribute.

7.3.2 Case Selection and Units of Analysis

To answer the posed research questions, we designed an exploratory case study (Runeson et al., 2012). For the same reasons posed in Chapter 6 (see Section 6.3.2), we decided to use the same five industrial projects reported]from a company in the domain of web and mobile applications development. Also for the same reasons posed in the previous chapter, we used the same unit of analysis, i.e., *changes that pattern instances undergo* (i.e., the source code change between two successive commits). We clarify that the usage of such unit of analysis entail the collection of multiple data points for individual pattern instances. However, each data point concerns a different snapshot of the pattern instance, i.e., there is no repetition. Moreover, the snapshot of a pattern instance is collected only when a change is made, i.e., the collection is not made for every commit. Nevertheless, it is paramount to avoid bias by having an excessive number of units from a few instances. For that, we verified the balance of our population on this regard (see Section 7.4).

Other main sources of bias regard the authoring of commits and the size of change, which may compromise our analyses. To address these concerns, we consulted with the company, which informed us that their developers are not allowed to commit for each other neither exchange source code to commit it. Moreover, a practice of small commits is encouraged to avoid the aforementioned bad practices.

Table 7.2: List of recorded variables

Step	Variable	Description
1	project	Project from which the PI was extracted
	commit	Hash of the commit in the git repository
	dev	Developer author of the commit
2	inst_id	ID of the PI that the class belongs to
	pattern	GoF design pattern of the instance
3	cg-napm	# alien public methods in all PI classes (Class grime)
	cg-naa	# alien attributes in all PI classes (Class grime)
	mg-ca	Afferent coupling of the PI (Modular grime)
	mg-ce	Efferent coupling of the PI (Modular grime)
	og-np	# packages within the PI (Organizational grime)
	og-ca	Afferent coupling at the package level (Organizational grime)
4	cor-viol	# correctness violations in all PI classes
	per-viol	# performance violations in all PI classes
	sec-viol	# security violations in all PI classes

PI stands for "pattern instance"
# stands for "Number of"

7.3.3 Variables and Data Collection

To address the research questions, we recorded four sets of variables for each unit of analysis. Each set regards one of the major steps in the data collection: 1) Characterize commits; 2) Collect patterns instances; 3) Assess pattern grime; 4) Assess quality attributes. In the following we describe each step, the variables collected in them (highlighted between parentheses), and tools we used. A summary of the recoded variables is presented in Table 7.2.

Step 1: Characterize Commits

The versioning of the five projects was managed using Git. For each commit we recorded the project name (*project*), the commit hashcode (*commit*) and the developer responsible for the commit (*dev*). We also recorded the files that were modified in order to filter out undesired commits. In particular, we ignored merges (as no modifications to source code are applied) and commits that did not modify pattern instances. We clarify that the latter filtering is performed in the next step.

Step 2: Collect Pattern Instances

This collection was performed for every commit, which is a time-consuming task. Hence, we automated this task using two tools: SSA (Similarity Score Analysis, v4.12) (Tsantalis et al., 2006), and SSA+[1] (v1.0.0). The same tools and their validation are discussed in details on Chapter 4 (see Section 4.3.3). We note that, to further validate the performance of the tool, we manually assessed 50 instances, which were all true positives.

Based on the collected information, we assign an ID (*inst_id*) for every instance and record it together with the type of the pattern (*pattern*). We note that IDs are assigned when instances are first detected and then reused when the same instance is detected again in later versions, i.e., they are persistent across versions of the project. Instances were considered equivalent if the main pattern participants had the same class name or matched a renamed version of the class (obtained from Git).

Step 3: Assess Pattern Grime

For every unit of analysis (i.e., change to a pattern instance), we assessed the amount of pattern grime accumulated with regards to its three forms (i.e., class, modular and organizational). For the same reasons described in Chapter 6 (see Section 6.3.3), we selected the same six metrics. In short, we calculate for class grime: (a) number of alien attributes (*cg-naa*), i.e., that are not described in the original pattern; and (b) number of alien public methods (*cg-napm*). We clarify that we consider only public methods, as they are responsible for exposing functionality of the pattern instance to the whole system. For modular grime, we calculate: (a) afferent pattern coupling (*mg-ca*) that is the amount of in-coming dependencies (or fan-in), representing the responsibility of the pattern instance (Izurieta and Bieman, 2013); and (b) efferent pattern coupling (*mg-ce*) that is the amount of dependencies on classes external to the pattern instance (or fan-out), representing the instability of the pattern instance (Izurieta and Bieman, 2013). Organizational grime is assessed by calculating: (a) number of packages (*og-np*) that contain classes participating on the pattern instance; and (b) afferent coupling at package level (*og-ca*). Although afferent coupling is also calculated for modular grime (*mg-ca*), *og-ca* will depict the responsibility at a higher level of abstraction. For example, *mg-ca* may increase within the same package containing the pattern instance, which would not affect *og-ca*.

To automate the data collection of the aforementioned metrics, we used the same tool presented in Chapter 6, namely spoon-pttgrime[2] (v0.1.0), which was developed during the PhD. To further validate the tool, we verified the calculated metrics for 50 pattern instances that were randomly selected, and the results were all correct. Based on the collected information, we record the amount of grime accumulated according to each metric, i.e., the difference between two consecutive versions of

[1] https://github.com/search-rug/ssap

[2] https://github.com/search-rug/spoon-pttgrime

the pattern instance.

We note that other indicators of grime have been proposed in the literature, which are based on taxonomies of modular and class grime (Griffith and Izurieta, 2014; Schanz and Izurieta, 2010). However, they are not independent grime indicators in the sense that they are subtypes of the indicators that we already investigate. Moreover, these additional indicators have been so far validated only through synthetic experiments (Dale and Izurieta, 2014; Griffith and Izurieta, 2014; Schanz and Izurieta, 2010), and there is no tool to automate their measurement. At the same time, the size of the population of our study makes it infeasible to assess them manually. Therefore, we decided to consider such indicators in our future work, and not include them in this study setup.

Step 4: Assess Quality Attributes

As mentioned in Section 7.1, we estimated the studied quality attributes based on the number of violations of various coding practices. For that, we followed a similar processes to that described in Chapter 4 (see Section 4.3.3). In short, we used FindBugs (v3.0.1), which considers bug patterns as rules to identify violation of good coding practices (Hovemeyer and Pugh, 2004). In particular, FindBugs organizes its rules (i.e., bug patterns) into nine high-level categories[3], from which five can be mapped into the studied quality attributes: correctness (Correctness and Multithreaded Correctness categories), performance (Performance category), and security (Security and Malicious Code categories). We note that, despite the name of the tool, we do not consider its output as bugs but simply as warnings, i.e., violations of good coding practices, and take them as indicators of quality. A similar approach was used by Khalid et al. (2016), who correlated the violations of three categories (one being performance) to quality as perceived by end-users. They found the data to be closely related, which supports the violations as quality indicators.

We selected FindBugs for the same reasons discussed in Chapters 2 and 4. Moreover, the validation and limitations of the tools is discussed in both chapters. We note that, to boost the precision of FindBugs, we exclude violations with low level of confidence. To estimate the level for each quality attribute, we calculate the amount of rules violations in the pattern-participant classes of a unit of analysis (*cor-viol*, *per-viol*, and *sec-viol*). We clarify that lower numbers of violations reflect a higher level of quality.

[3] The categories are: Security, Correctness, Multithreaded Correctness, Performance, Malicious Code, Bad Practice, Internationalization, Experimental and Dodgy Code

7.3.4 Analysis Procedure

To investigate the collected data, we performed various statistical analyses. First, to answer RQ_1, we calculated the correlation between every pair of <grime metric, quality indicator> (e.g., pattern efferent coupling vs. performance violations). We assess the strength of the correlation according to the guidelines of Evans (1996): 'very weak' (0.00-0.19); 'weak' (0.20-0.39); 'moderate' (0.40-0.59); 'strong' (0.60-0.79); and 'very strong' (0.80-1.00). To select the most fitting method for correlation analysis, we first tested the normality of our data, using the Kolmogorov-Smirnov test (Field, 2013), which is more appropriate for large samples. We clarify that for normally distributed variables, we used Pearson correlation method (Field, 2013), otherwise, we used the Spearman's rank correlation method (Field, 2013). Moreover, the correlations calculated in this study do not entail bias from consecutive measurements with same value, also known as artificial boost. This is because every unit of analysis regards different states of a particular pattern instance. Therefore, consecutive measures with same value suggest that a specific metric is not designed to capture this particular change, and this information is relevant to our study.

To answer RQ_2, we performed the following steps for each factor (i.e., project, pattern, developer) that might affect the relation between pattern grime and quality. First, we grouped the dataset according to the factor. Next, we verified whether the groups differentiate between themselves with regards to the measured variables. For that, we performed an Analysis of Variance (ANOVA) (Field, 2013) to confirm a disparity among groups, followed by post-hoc tests for pairwise comparisons. We note that we applied Levene's test (Field, 2013) to assess the assumption of equal variances of the tested populations. When the assumption was met, we used regular ANOVA, followed by Tukey's Honestly Significant Differences (HSD) tests (Field, 2013). Otherwise, we used Welch's ANOVA, followed by Games-Howell tests (which are more appropriate for large samples). Finally, for the groups that are statistically different, we calculate the correlation for each pair of grime and quality metrics and identify statistically relevant correlations.

7.4 Results

In this section, we present a summary of the collected data, as well as the results of the analysis performed to answer the research questions posed in Section 7.3.1. During the data collection, we identified 1,422 commits that contain the creation or modification of pattern-participating classes of the five investigated projects, from which the majority (94%) regard the modification of one or more pattern instances. Based on the commits, we isolated 2,329 pattern instances of eight different GoF

Table 7.3: Summary of dataset

Project	Pattern	Number of instances	Number of units of analysis
P1	(Object) Adapter/Command	284	8150
	Singleton	80	155
	State/Strategy	351	11586
P2	(Object) Adapter/Command	86	484
	Observer	3	21
	Singleton	16	42
	State/Strategy	275	2113
	Template Method	1	6
P3	(Object) Adapter/Command	327	3090
	Factory Method	53	545
	Singleton	29	152
	State/Strategy	375	3995
P4	(Object) Adapter/Command	136	2144
	Decorator	1	1
	Factory Method	13	266
	Singleton	21	73
	State/Strategy	230	3275
P5	(Object) Adapter/Command	16	206
	Factory Method	2	33
	Singleton	5	10
	State/Strategy	25	224

patterns: (Object) Adapter / Command, Decorator, Factory Method, Observer, Singleton, State / Strategy, and Template Method. In Table 7.3 we present a summary of the units of analysis by project and patterns.

In Section 7.3.2, we highlighted the necessity of having a balanced population (i.e., pattern instances should have similar number of modifications) to avoid bias from pattern instances with excessive number of units of analysis. After studying the history of commits, we assessed that each pattern instance underwent a maximum of 178 modifications. Moreover, 87% of the pattern instances (i.e., 2,039) were modified at least once, and 64% (i.e., 1,500) at least five times. Our analysis suggests that although the population is not evenly balanced, the discrepancies are not enough to harm the statistical analysis of our study nor the answers to the research questions.

In summary, we collected a total of 36,571 units of analysis (i.e., creation/modification of a pattern instance in a commit). For each unit, we recorded the amount of pattern grime that was accumulated according to six metrics (*cg-**, *mg-** and *og-**) and the number of violations regarding the three studied quality attributes (**-viol*). We clarify that due to a non-disclosure agreement signed with the company in this

Table 7.4: Descriptive statistics per commit

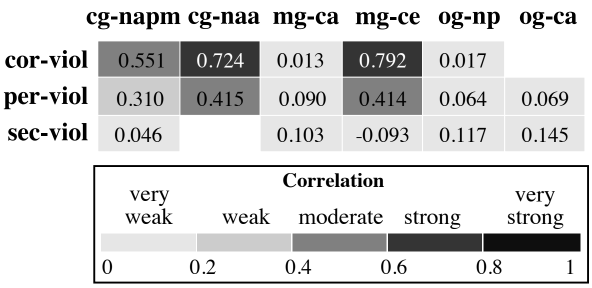
Variable	Minimum	Maximum	Mean	Std. Deviation
cg-napm	0.00	52.80	13.66	8.27
cg-naa	0.00	41.50	7.95	5.00
mg-ca	0.00	107.00	8.01	10.72
mg-ce	0.00	325.00	100.18	61.22
og-np	1.00	5.00	2.34	0.53
og-ca	0.00	54.00	6.75	8.06
cor-viol	0.00	35.00	2.39	3.19
per-viol	0.00	8.00	0.97	1.45
sec-viol	0.00	20.00	0.25	1.40

case study, we cannot share the created dataset, nor certain details regarding specific projects and developers.

To characterize our population, in Table 7.4 we present the descriptive statistics for these variables. We notice that pattern efferent coupling (*mg-ce*) is the grime metric that changes the most, which may be a sign of bad practices since it represents the dependency of the pattern instance on other classes. On the counterpart, number of packages (*og-np*) is the metric that changes the least, which is expected given that pattern instances normally grow within the same package. Furthermore, we notice that violations of good practices regarding correctness appear to be considerably more frequent than regarding performance and security. This observation may be partially related to the fact that the majority of the rules checked by FindBugs concern correctness: out of all the rules for the three studied qualities, correctness accounts for approx. 70%, while performance and security correspond to approx. 15% each. Nevertheless, we could detect considerably fewer violations concerning security rather than performance, which suggest that other parameters are also relevant, such as the type of application or even the specific security-related violations that FindBugs checks.

7.4.1 RQ$_1$ - Grime and Quality Attributes

To answer RQ$_1$, we calculated the correlation between all pairs of <*grime metric*, *quality indicator*> (e.g., *cg-ce* vs. *per-viol*) as explained in Section 7.3.4. We note that we could not assume normal distribution for all variables and, thus, we used Spearman's rank correlation method. Moreover, 'artificial boost' is not a concern in this population (see Section 7.3.4). Figure 7.1 depicts a heatmap with the results of our analysis, in which darker shades of gray denote stronger correlation. The coefficients are written within each cell except for correlations that are not statistically significant (which are blank). Based on Figure 7.1 we can make several observa-



	cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca
cor-viol	0.551	0.724	0.013	0.792	0.017	
per-viol	0.310	0.415	0.090	0.414	0.064	0.069
sec-viol	0.046		0.103	-0.093	0.117	0.145

Correlation
very weak
weak
moderate
strong
very strong
0
0.2
0.4
0.6
0.8
1

Figure 7.1: Correlation between grime metrics (*cg-**, *mg-**, *og-**) and quality attributes

tions.

The accumulation of grime seems to be related with the depreciation of correctness and performance (i.e., more violations), as we observed strong correlations (i.e., above 0.6) and moderate correlations (i.e., between 0.4 and 0.59) respectively (Evans, 1996). Furthermore, the very weak correlation with security violations (i.e., below 0.2) does not imply that a link does not exist. This only shows a lack of evidence.

Another observation is that ***metrics for assessing class grime, namely number of alien public methods (cg-napm), alien attributes (cg-naa), and pattern efferent coupling (mg-ce) displayed the strongest correlations regarding every quality attribute***. This outcome can be considered intuitive in the sense that, as structural elements at the class level, patterns are expected to be more influential at lower levels of granularity (e.g., class rather than module). The degradation of another quality, namely maintainability, due to the existence of alien methods is also reported in related work (Charalampidou et al., 2017).

The aforementioned observations are based on how grime accumulates in pattern instances. However, one may wonder if changes in the quality levels can be simply explained by natural evolution of the source code, i.e., any type of change to the pattern instance rather than pattern grime. To investigate this possibility, we assessed the correlation between lines of code (LOC) and both grime metrics and quality indicators. The results show that grime is strongly correlated (0.81) with LOC, i.e., most maintenance activities in pattern instances entail accumulation of grime. However, the correlation between grime and quality indicators was often slightly stronger compared to the correlation between LOC and quality indicators. For example, the correlation between *cor-viol* vs. *mg-ce* (0.792) is marginally stronger than *cor-viol* vs. LOC (0.785), *per-viol* vs. *mg-ce* (0.414) is stronger than *per-viol* vs. LOC (0.359), and *sec-viol* vs. *mg-ce* (-0.093) is stronger than *sec-viol* vs. LOC (-0.074). Therefore, although the difference between correlation values may be marginal at times, the overall analysis consistently shows that grime matches the degradation of quality better than natural evolution.

7.4.2 RQ$_2$ - Analysis of Factors

To further explore the relation between the accumulation of pattern and the three studied quality attributes, we investigated three factors that may influence the observed correlations as described in Section 7.3.4: projects, patterns and developers.

Comparison of Projects

We collected data from five different industrial projects, here referred to as P1 to P5. From the 36,571 units of analysis, 19,891 regard P1, 2,667 regard P2, 7,781 regard P3, 5,759 regard P4, and 473 regard P5. Moreover, P1 and P2 were developed by two different teams of developers while a third team developed P3, P4 and P5. In Table 7.5, we show the descriptive statistics of all variables for each project independently.

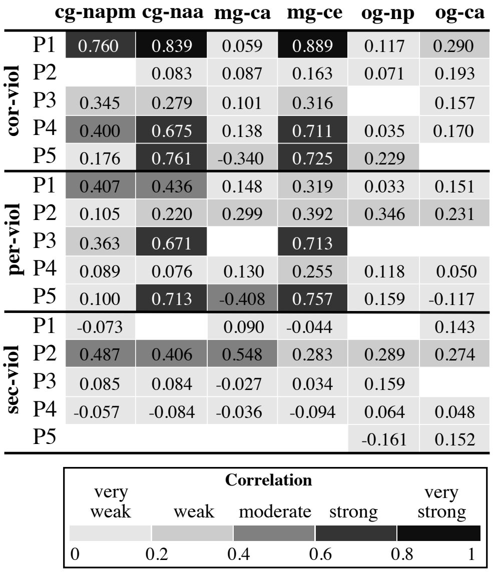
We notice that the projects are considerably distinct from each other with regard to these variables. For example, P2 has the highest mean for most grime metrics but not for quality indicators, while P5 has the lowest means for grime metrics but present the highest average of performance violations. To verify the observed differences, we compared the means between projects by performing an analysis of variances (ANOVA) for each variable, followed by one post-hoc test for each pairwise comparison (i.e., 90 in total). The results of the tests are publicly available online in a supplementary material[4]. The results show that 91% of the tests are statistically significant, i.e., the means differ between the two compared projects. Based on these findings, we hypothesize that the different characteristics of projects are indeed reflected on the relationship between the accumulation of pattern grime and the indicators of correctness, performance and security.

To verify how the accumulation of grime in projects relates to the levels of quality, we calculated the correlation between every pair of grime metric and quality indicator for each project. The results are presented in Figure 7.2 (which is interpreted as Figure 7.1), from which we observe that the correlations are noticeably different based on the projects. For example, similar to the results observed for the general population, P1 exhibits a strong correlation (i.e., above 0.6) between class grime metrics and the correctness indicator (*cor-viol*). The opposite is observed for P2, for which the data suggest a correlation between pattern grime and the security indicator (*sec-viol*), which have not been observed for the general population.

[4] https://doi.org/10.5281/zenodo.1133552

Table 7.5: Descriptive statistics per project

Variable	**Project**	**Minimum**	**Maximum**	**Mean**	**Std. Deviation**
cg-napm	P1	0.00	44.67	13.94	8.26
	P2	0.00	48.00	14.92	11.90
	P3	0.00	52.80	11.79	6.47
	P4	0.00	42.00	14.89	8.13
	P5	0.00	29.75	10.36	5.73
cg-naa	P1	0.00	22.33	8.97	4.87
	P2	0.00	30.33	5.87	3.72
	P3	0.00	30.50	5.90	3.81
	P4	0.00	41.50	8.45	6.13
	P5	0.00	13.75	4.72	2.50
mg-ca	P1	0.00	49.00	6.52	8.48
	P2	0.00	78.00	17.69	18.65
	P3	0.00	107.00	7.44	10.29
	P4	0.00	102.00	10.02	10.85
	P5	0.00	5.00	1.00	1.07
mg-ce	P1	0.00	325.00	132.05	62.91
	P2	0.00	140.00	55.07	28.23
	P3	0.00	162.00	63.31	29.65
	P4	0.00	141.00	65.50	27.99
	P5	0.00	95.00	43.13	19.02
og-np	P1	1.00	3.00	2.31	0.48
	P2	1.00	3.00	2.64	0.53
	P3	1.00	5.00	2.30	0.60
	P4	1.00	4.00	2.35	0.54
	P5	1.00	3.00	2.21	0.46
og-ca	P1	0.00	54.00	5.55	8.32
	P2	0.00	29.00	15.42	8.98
	P3	0.00	44.00	7.71	7.50
	P4	0.00	19.00	6.03	3.95
	P5	0.00	5.00	0.88	0.55
cor-viol	P1	0.00	35.00	3.74	3.68
	P2	0.00	5.00	0.16	0.55
	P3	0.00	7.00	0.66	1.19
	P4	0.00	10.00	1.24	1.25
	P5	0.00	2.00	0.63	0.65
per-viol	P1	0.00	7.00	1.16	1.58
	P2	0.00	8.00	1.12	1.50
	P3	0.00	6.00	0.72	1.15
	P4	0.00	5.00	0.57	1.10
	P5	0.00	8.00	1.16	1.66
sec-viol	P1	0.00	2.00	0.03	0.18
	P2	0.00	14.00	1.31	2.65
	P3	0.00	14.00	0.31	1.33
	P4	0.00	20.00	0.42	2.41
	P5	0.00	8.00	0.28	0.89



		cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca
cor-viol	P1	0.760	0.839	0.059	0.889	0.117	0.290
	P2		0.083	0.087	0.163	0.071	0.193
	P3	0.345	0.279	0.101	0.316		0.157
	P4	0.400	0.675	0.138	0.711	0.035	0.170
	P5	0.176	0.761	-0.340	0.725	0.229	
per-viol	P1	0.407	0.436	0.148	0.319	0.033	0.151
	P2	0.105	0.220	0.299	0.392	0.346	0.231
	P3	0.363	0.671		0.713		
	P4	0.089	0.076	0.130	0.255	0.118	0.050
	P5	0.100	0.713	-0.408	0.757	0.159	-0.117
sec-viol	P1	-0.073		0.090	-0.044		0.143
	P2	0.487	0.406	0.548	0.283	0.289	0.274
	P3	0.085	0.084	-0.027	0.034	0.159	
	P4	-0.057	-0.084	-0.036	-0.094	0.064	0.048
	P5					-0.161	0.152

Figure 7.2: Correlation between grime metrics (*cg-**, *mg-**, *og-**) and quality attributes indicators (**-viol*) for individual projects (P*)

However, we also noticed that higher values of accumulated grime are related to higher depreciation of quality (i.e., higher number of violations), which is often reflected in higher correlation coefficients. ***This evidence strengthens our finding that the relationship between pattern grime and quality attribute indicators is project-dependent***. It also suggests that the observed difference is connected to how grime accumulates in the different projects. This finding is in accordance to those of Linares-Vásquez et al. (2014), which suggest that other indirect quality indicators (such as anti-patterns or code smells) vary among projects of different application domains, as well as with Izurieta and Bieman (2013), who observed varied levels of grime and quality on the studied projects.

Comparison of Patterns

During the data collection, we identified instances of eight different patterns. From the 36,571 units of analysis, 14,074 regard the (Object) Adapter / Command (AC) patterns, 844 regard the Factory Method (FM) pattern, 432 regard the Singleton (Si) pattern, 21,193 regard the State/Strategy (SS) patterns, 21 regard the Observer pattern, six regard the Template Method pattern, and one regards the Decorator pattern. Due to the limited amount of units, we do not present results concerning the last three patterns, which are available in the supplementary material.

In Table 7.6, we present the descriptive statistics of all variables for each pattern independently. We notice that this factor also seems to influence the relations between pattern grime and indicators of the studied quality attributes. In particular, we observe that the means for every metric varies considerably among patterns. Moreover, we could not observe clear trends, i.e., patterns that consistently display the highest or lower means. For example, Factory Method displays the highest mean of security violations (*sec-viol*) but one of the lowest of correctness violations (*cor-viol*).

To verify our observations, we computed the ANOVA for each variable and performed the post-hoc tests (i.e., 48 in total). The results of the tests are available in the supplementary material. We note that Singleton instances had no variance with regards to number of packages (*og-np*) and security indicator (*sec-viol*), and, thus, these variables were not considered in the analyses for this pattern. The results show that 93% of the tests are statistically significant.

To further investigate this factor, we calculated the correlation between every pair of grime metric and quality indicator for the investigated patterns. In Figure 7.3 (which is interpreted as Figure 7.1), we present the results of the calculations, which show clearly varying correlations depending on the pattern. We notice that, as for projects, we could identify a pattern, namely Factory Method, for which the accumulation of grime is moderately correlated with the depreciation of quality indicators. Again, we observed that the combination of ***higher accumulation of grime and quality indicators often reflects in higher correlation coefficients***. All this information suggests that the ***relationship between pattern grime and quality attribute indicators also depends on the pattern type of the instance***. This finding is in accordance with the literature, which suggests that different patterns have different effects on the same quality attribute (e.g., (Ampatzoglou et al., 2015; Romano et al., 2012)).

Comparison of Developers

The case study involved 16 developers, here referred to as D1 to D16, which account for various amounts of units of analysis[5]. Due to the low number of data points, we did not include D4, D7, D8 and D10 in our analyses. In Table 7.7, we show the mean value of all variables, for each developer. We note that we do not present the complete descriptive statistics, which are available in the supplementary material. Similar to the previous factors, we observe that ***mean values regarding all variables***

[5] The number of units by each developer is: D1 - 810; D2 - 5662; D3 - 1535; D4 - 8; D5 - 470; D6 - 5368; D7 - 21; D8 - 62; D9 - 1464; D10 - 11; D11 - 811; D12 - 1648; D13 - 3565; D14 - 6748; D15 - 7825; D16 - 563.

Table 7.6: Descriptive statistics per pattern

Variable	Pattern	Minimum	Maximum	Mean	Std. Deviation
cg-napm	AC	0.00	43.00	12.59	7.70
	FM	2.40	52.80	15.63	8.89
	Si	0.00	6.00	0.82	1.35
	SS	0.67	48.00	14.55	8.39
cg-naa	AC	0.50	41.50	8.53	5.78
	FM	1.00	23.20	6.86	4.43
	Si	0.00	3.00	0.44	0.73
	SS	0.00	30.33	7.77	4.32
mg-ca	AC	0.00	49.00	6.57	8.71
	FM	1.00	107.00	19.70	23.46
	Si	0.00	52.00	7.39	10.54
	SS	0.00	78.00	8.51	10.80
mg-ce	AC	5.00	236.00	90.24	51.97
	FM	10.00	129.00	58.00	25.67
	Si	0.00	17.00	1.68	2.52
	SS	1.00	325.00	110.56	64.80
og-np	AC	1.00	2.00	1.99	0.10
	FM	1.00	5.00	3.37	0.68
	Si	1.00	1.00	1.00	0.00
	SS	1.00	4.00	2.56	0.50
og-ca	AC	0.00	46.00	5.82	7.72
	FM	1.00	44.00	13.34	8.15
	Si	0.00	54.00	13.06	8.03
	SS	0.00	46.00	6.96	8.07
cor-viol	AC	0.00	13.00	2.19	2.84
	FM	0.00	5.00	0.69	1.02
	Si	0.00	3.00	0.02	0.21
	SS	0.00	35.00	2.64	3.44
per-viol	AC	0.00	5.00	0.82	1.30
	FM	0.00	6.00	0.56	1.22
	Si	0.00	0.00	0.00	0.00
	SS	0.00	8.00	1.11	1.54
sec-viol	AC	0.00	6.00	0.04	0.26
	FM	0.00	20.00	4.71	5.81
	Si	0.00	12.00	0.48	1.67
	SS	0.00	14.00	0.21	1.05

differ among developers, i.e., they exhibit different characteristics. For example, both D11 and D15 show higher tendency to pollute pattern instances with alien methods (i.e., higher *cg-napm* values) than other developers. However, D11 seems much less prone to pollute instances with external dependencies (i.e., lower *mg-ce*).

To validate the differences observed in the measurements, we performed

Table 7.7: Descriptive statistics per developer

Dev.	cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca	cor-viol	per-viol	sec-viol
D1	9.50	3.17	8.40	38.09	2.28	7.07	0.47	0.00	0.25
D2	12.53	7.17	8.02	63.75	2.32	7.21	1.03	0.56	0.34
D3	9.83	7.11	7.68	70.78	2.30	8.59	0.26	1.14	0.29
D5	12.58	5.84	5.48	56.39	2.27	5.08	0.45	0.97	0.27
D6	14.01	9.15	7.00	127.50	2.30	5.65	3.57	1.59	0.05
D9	9.62	6.76	6.41	99.90	2.27	5.02	2.21	0.87	0.00
D11	16.36	5.88	14.27	51.18	2.59	13.07	0.07	0.83	0.96
D12	12.83	7.92	8.03	122.20	2.38	7.20	2.99	1.11	0.01
D13	12.42	8.36	7.21	112.66	2.26	5.49	2.61	1.83	0.08
D14	14.81	6.99	11.12	64.40	2.40	8.06	0.89	0.88	0.65
D15	15.65	9.78	5.57	152.28	2.34	5.28	4.83	0.64	0.01
D16	14.31	5.66	16.25	53.53	2.60	15.48	0.17	0.75	0.96

ANOVA on all variables, followed by the post-hoc tests (583 in total), which are all available in the supplementary material. We note that no variance in the security indicator (*sec-viol*) was observed for D9 and, thus, we discarded this variable for analyses regarding the developer. The results show that 80% of the tests are statistically significant. The majority of the comparisons that were not significant, concern the number of packages, which is intuitive, as pattern instances do not tend to be spread across multiple packages/namespaces.

We also calculated the correlation between variables, which are shown in Figure 7.4 (which is interpreted as Figure 7.1). The results suggest that developers accumulate grime differently and that this may reflect on the quality indicators. We also observed that ***although we found that correlations differ among developers, they are mostly consistent in the sense that more grime is correlated with more violations*** (i.e., depreciated quality). In summary, all collected information strengthens our finding that developers comprise a factor to how the accumulation is related to the depreciation of correctness, performance and security in pattern instances. Our results are in accordance with those by Amanatidis et al. (2017), who studied the accumulation of technical debt and observed an imbalance regarding the number of violations among developers.

7.5 Discussion

In this section, we revisit the findings of our study and present their connection to related work. Next, we elaborate on the main implications to researchers and practitioners.

		cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca
cor-viol	AC	0.566	0.769	0.072	0.794	0.063	0.130
	FM	0.173	0.324	0.130	0.404		
	Si	0.201	0.142		0.104		
	SS	0.547	0.709	-0.014	0.808	-0.020	-0.028
per-viol	AC	0.243	0.359	-0.018	0.378		0.049
	FM	0.435	0.457	0.124	0.480	0.289	0.134
	Si						
	SS	0.328	0.437	0.154	0.401		0.108
sec-viol	AC	-0.093	-0.051	-0.112	-0.096	0.019	-0.020
	FM	0.514	0.509	0.345	0.527	0.310	0.287
	Si	-0.218	-0.218				0.144
	SS	0.069	0.030	0.136	-0.084		0.142

Correlation

very weak	weak	moderate	strong	very strong
0 – 0.2	0.2 – 0.4	0.4 – 0.6	0.6 – 0.8	0.8 – 1

Figure 7.3: Correlation between grime metrics (*cg-**, *mg-**, *og-**) and quality attributes indicators (**-viol*) for individual patterns (AC, FM, Si, and SS)

7.5.1 Interpretation of Results

Correlation between Grime and Attributes

The findings discussed in this chapter suggest that, as pattern grime accumulates, ***classes that participate in pattern instances become more prone to quality depreciation***. In particular, such classes are more susceptible to source code that violates good practices that promote correctness, performance and security of software systems. These findings corroborate those by related work that analyzes the relations between grime and quality (Griffith and Izurieta, 2014; Izurieta and Bieman, 2008, 2013), in the sense that we also found that ***grime goes hand in hand with diminished quality***.

In our study, we noticed that ***three metrics, namely number alien attributes (cg-naa), number of alien public methods (cg-napm) and instance efferent coupling (mg-ce), were the most likely to be appropriate indicators of bad quality***; these same metrics have shown similar relevance in the related work. Moreover, these metrics correspond to structural characteristics of pattern instances (e.g., efferent coupling), and similar metrics (at class level rather than instance level) have been largely explored in the literature (e.g. (Charalampidou et al., 2017; Hsueh et al., 2008; Huston, 2001; Muraki and Saeki, 2002)) and found to be good estimators of the benefit (or harmfulness) of pattern instances to quality attributes. In the study reported in Chapter 6, we found that the degradation of certain well-known design metrics can be used as hints of the accumulation of pattern grime (Feitosa, Avgeriou, Ampatzoglou and Nakagawa, 2017b), as it is assessed based on design propertied of pattern participants. In particular, we investigated the metric suits proposed by Chidamber and Kemerer (1994), Li and Henry (1993), and Bansiya and Davis (2002). Results of that study showed that the metrics data abstraction coupling (DAC) (Li and Henry, 1993) and measure of aggregation (MOA) (Bansiya and Davis, 2002) may help identifying accumulation of *cg-naa*; the metrics weighted methods per class (WMC) (Chidamber and Kemerer, 1994) and class interface size (CIS) (Bansiya and Davis, 2002) may help identifying accumulation of *cg-napm*; and the metrics coupling between object classes (CBO) (Chidamber and Kemerer, 1994) and response for a class (RFC) (Chidamber and Kemerer, 1994) may help identifying accumulation of *mg-ce*.

Contributing Factors

The way pattern grime builds up in pattern-participant classes can depend on several factors. Our empirical investigation confirmed that three such factors indeed play a role: project, pattern and developer. With regard to projects, we observed that the difference may be related to two sub-factors. The ***type of the project seems relevant on determining the relation between grime and quality***. Two of the studied projects (P1 and P4) provide services to other applications (e.g., libraries or API's) and showed to be more prone to grime and violations; this aligns with the suggestion by Evans (1996) that parameters such as application domain can be relevant. However, we also noticed that these projects had more pattern instances (i.e., a bigger pattern code base) and that a second sub-factor, namely ***lines of code was also correlated with both grime and quality indicators***; this has also been discerned by Izurieta and Bieman (2013).

A similar observation also holds for developers: ***those that wrote more code (i.e., provided more units of analysis) were more prone to incur both grime and violations***. Finally, our main observation concerning the difference among patterns is that those using ***more complex mechanisms (e.g., State, Strategy and Factory Method, which have polymorphic calls) tend to accumulate both more grime and violations***; this is intuitive given that more complex designs are less understandable and harder to maintain.

Investigating the factors in isolation allowed us to observe that the ***correlations in different groups (based on factors) differ from the ones concerning the entire dataset***. However, although the differences may look random at first, we noticed a recurrent motif. In particular, we observed that the majority (approx. 80%) of ***mod-***

		cg-napm	cg-naa	mg-ca	mg-ce	og-np	og-ca
cor-viol	D1	0.417	0.534		0.579		
	D2	0.302	0.562	0.090	0.519		0.098
	D3	0.254	0.263	0.157	0.221		0.221
	D5	0.293	0.414		0.375	0.130	
	D6	0.732	0.778	0.053	0.849	0.106	0.238
	D9	0.723	0.791		0.743	-0.094	-0.135
	D11						
	D12	0.780	0.872		0.887		0.089
	D13	0.746	0.800	0.083	0.815	0.075	0.240
	D14	0.405	0.518	0.087	0.536	-0.024	0.034
	D15	0.753	0.881	0.153	0.928	0.170	0.437
	D16			0.093	0.195	0.105	0.152
per-viol	D1	0.118	0.121		0.121		0.079
	D2	0.214	0.299	-0.053	0.498		-0.051
	D3	0.543	0.649	0.096	0.744		-0.085
	D5	0.315	0.604		0.688		
	D6	0.494	0.482	0.111	0.445		0.137
	D9	0.580	0.669		0.555	-0.085	-0.166
	D11	0.128	0.180	0.272	0.265	0.285	0.277
	D12	0.400	0.415		0.456		-0.087
	D13	0.696	0.676	0.079	0.716	0.033	0.271
	D14	0.080	0.289	0.112	0.394	0.167	0.155
	D15	0.401	0.394	0.216	0.350	0.136	0.330
	D16		0.222	0.362	0.451	0.435	0.250
sec-viol	D1	0.221	0.283		0.124	0.115	
	D2			-0.031		0.114	0.057
	D3	0.193	0.131		0.093	0.156	
	D5			-0.091		0.100	
	D6	-0.119	-0.029	0.074	-0.063		0.171
	D9						
	D11	0.367	0.393	0.542	0.259	0.270	0.287
	D12			0.056			0.062
	D13	-0.106		0.085	-0.037		0.257
	D14	0.146	0.081	0.160		0.212	0.206
	D15	0.058	0.052	0.092	0.053	0.063	0.072
	D16	0.388	0.353	0.432	0.225	0.254	0.115

Correlation

very weak	weak	moderate	strong	very strong
0 – 0.2	0.2 – 0.4	0.4 – 0.6	0.6 – 0.8	0.8 – 1

Figure 7.4: Correlation between grime metrics (*cg-**, *mg-**, *og-**) and quality attributes indicators (**-viol*) for individual developers (D*)

erate or strong correlations *(i.e., more than 0.4)* (Evans, 1996) ***have been identified when grime and qualities metrics are at a similar level***. For example, projects that on average concentrate few violations and have low levels of accumulated grime, or the opposite. Among those, 56% regard higher values on both grime and quality indicators.

Analysis of Violations

Finally, since we estimated the levels of quality attributes through the number of violations of good coding practices, it is relevant to dig deeper into these violations. In Table 7.8, we present the most recurrent violations, assessed according to the addressed research questions, i.e., the overall dataset, per project, per pattern and per developer. We note that some developers have not violated any rules for certain quality attributes in pattern-participant classes; those are marked with "-". We observe that this ***list of violations comprises issues that are clearly harmful to the respective quality attributes***, e.g., calling unsafe methods in a multithreaded context can lead to race conditions or unpredictable states.

Thus, if these violations are among the recurrent ones, they can pose a serious threat to the system. Furthermore, the ***top issues vary among projects, patterns and developers***. The differences that we observe between developers is aligned with the findings by Amanatidis et al. (2017), who not only observed an imbalance on how developers accumulate violations but also a difference on the recurrence. Nevertheless, ***it is possible to discern the connection between groups***. For example, the two recurrent performance issues that appear for the most among developers (i.e., "Comparison of different types" and "Possible null pointer dereference"), also appear frequently among projects and patterns, and one of them is the most recurrent in the entire dataset.

7.5.2 Implications to Researchers and Practitioners

GoF patterns are popular among practitioners as established and valuable design solutions. However, the consequences of using them often become a matter of concern, especially regarding quality. This chapter sheds some light on this respect, suggesting the following implications to practitioners. We encourage the ***conscious usage of GoF patterns***, in the sense that knowledge about the patterns being applied, as well as the pattern instances in the system under development, should be disseminated within the team of developers.

In addition, ***monitoring the pattern instances is of paramount importance to maintain desired levels of quality***, especially correctness, performance and security. Moreover, practitioners can take advantage of the tool spoon-pttgrime in order

Table 7.8: Most recurrent violations

		Correctness	**Performance**	**Security**
Overall		Comparison of different types	Class member should be static	Exposed inner representation by incorporating mutable object
Project	P1	Comparison of different types	Class member should be static	Method invocation without proper security check
	P2	Unsafe call in for multithreading	Class member should be static	Exposed inner representation by returning mutable object
	P3	Possible null pointer dereference	Private method is never called	Exposed inner representation by incorporating mutable object
	P4	Unsafe call for multithreading	Unnecessary value unboxing	Exposed inner representation by incorporating mutable object
	P5	Unsafe call for multithreading	Unnecessary call to toString()	Exposed inner representation by incorporating mutable object
Pattern	AC	Comparison of different types	Class member should be static	Field should be package protected
	FM	Possible null pointer dereference	Unnecessary value unboxing	Exposed inner representation by incorporating mutable object
	Si	Comparison of different types	Inefficient use of map iterator	Exposed inner representation by incorporating mutable object
	SS	Comparison of different types	Class member should be static	Exposed inner representation by returning mutable object
Developer	D1	Comparison to null	-	Exposed inner representation by incorporating mutable object
	D2	Possible null pointer dereference	Unnecessary value unboxing	Exposed inner representation by incorporating mutable object
	D3	Possible null pointer dereference	Class member should be static	Exposed inner representation by incorporating mutable object
	D5	Unsafe call for multithreading	Unnecessary value unboxing	Exposed inner representation by incorporating mutable object
	D6	Comparison of different types	Private method is never called	Method invocation without proper security check
	D9	Possible null pointer dereference	Private method is never called	-
	D11	Variable self-assignment	Class member should be static	Exposed inner representation by incorporating mutable object
	D12	Possible null pointer dereference	Class member should be static	-
	D13	Nullcheck on dereferenced variable	Invoke of inefficient constructor	Method invocation without proper security check
	D14	Unsafe call for multithreading	Class member should be static	Exposed inner representation by returning mutable object
	D15	Comparison of different types	Private method is never called	-
	D16	Unsafe call for multithreading	Class member should be static	Exposed inner representation by incorporating mutable object

to track the accumulation of grime and plan maintenance activities. Conversely, if practitioners already use FindBugs within their development process, the number of violations (for correctness, performance and security) can be used as indicators of grime accumulation, helping the team on identifying pattern instances with potentially deteriorated design.

The findings of this study can also benefit researchers. Our work joins the small pool of studies that investigate pattern grime, especially its relation to quality attributes, and further demonstrate the relevance of researching this phenomenon and the underlying relations. In particular, ***we provide evidence that encourages the investigation of other quality attributes, as well as factors related to it***. The presented information also builds up on the body of knowledge on pattern grime, and we hope it will support future research. Particularly, we envisage confirmatory studies to seek more evidence to explain the observed variations in the relationship between grime and the studied quality attribute, as well as others. We also demonstrate that the usage of static analysis tools such as FindBugs can provide valuable information regarding the accumulation of grime. Finally, the design of our case study and used tools used can be exploited for future research efforts.

7.6 Threats to Validity

In this section, we discuss threats to the validity of the study reported on this chapter; in particular, construct validity, reliability and external validity. Construct validity concerns to what extent the objects of the study are connected to the research questions. Reliability regards the extent to which the study can be replicated with the same observed results. External validity pertains to the limitations to generalize our findings to the entire population. We note that we do not analyze internal validity, as we empirically study the correlation between variables without establishing causal relations.

Regarding construct validity, we identified the following threats. First, the SSA and FindBugs tools are limited by their precision and recall, which may bias our results due to false positives and negatives. We note that, to the best of our knowledge, these tools have adequate performance and good reputation (see Section 7.3.3, Steps 2 and 4). Nevertheless, to mitigate this threat, we randomly selected 50 pattern instances and verified the output from each tool manually. In addition, we acknowledge that the list rules provided by FindBugs is by no means exhaustive and additional rules could affect our results. However, we reiterate that the diverse list of bug patterns (i.e., 252 rules) and evidence provided by other studies that used FindBugs to estimate quality attributes (Feitosa et al., 2015; Khalid et al., 2016) suggest that the tool is adequate for the purpose used in this study. Finally, concerning the tools developed in our group (SSA+ and spoon-pttgrime), which although perform deterministic tasks, may contain bugs and bias the results of the study. To mitigate this threat, we also checked their output for 50 randomly selected pattern instances. In addition, our tools have been used in previous studies, where they were also validated.

To address reliability threats, at least two researchers were involved in both data collection and analysis. Samples of the output were checked by both researchers and the verification followed a checklist to avoid irregularities. Furthermore, most tasks were automated by the tools referenced in this chapter, which are all publicly available. Despite our effort, we acknowledge that non-disclosure agreements do not allow us to share the collected dataset. However, replications studies can be carried out to attempt to replicate our results.

Concerning external validity, the main threat is that we explored projects from the same company, from which three were developed by the same team. Such uniformity (e.g., developers subject to same company practices) may lessen the generalizability of our findings to other companies or teams. However, we note that the accumulation of grime that we observed aligns with the results of other studies, e.g., class and modular grime are the main indicators of grime. Moreover, we also aimed at identifying variations in the relationship between pattern grime and quality attributes based on project and developer, which we identified successfully despite the "uniformity" of our subjects. The other threats regard limitations of our study design. In particular, we investigated a limited number of patterns and subjects, and we acknowledge that additions to the population may affect our findings. Furthermore, we investigated projects developed in Java and our observations cannot be generalized to other languages without additional analyses. Finally, the grime metrics and quality indicators are estimators, and the usage of different variables may affect the observed results. Specifically, the inclusion of metrics based on subtypes of grime could provide more refined observations.

7.7 Conclusions

In this chapter, we reported on an exploratory case study with five industrial software systems, in which we examined the relationship between the accumulation of pattern grime and the levels of three quality attributes, namely correctness, performance and security. For that, we considered six metrics regarding the three forms of grime (i.e., class, modular and organizational), and one indicator of each studied quality attribute, estimated by the amount of violations of coding practices in pattern-participant classes. We investigated the evolution of 2,329 pattern instances over 1,422 commits, totalizing 36,571 units of analysis, in which we assessed the correlations between the grime metrics and quality indicators. Moreover, we sought to analyze factors that might influence the observed correlations, in particular, projects, pattern types, and developers.

The results suggest that pattern grime is related to the depreciation of correctness, performance and security in pattern instances. These findings are based on both class and modular grime, whilst no strong evidence is observed based on organizational grime. The results also suggest that all three examined factors can influence the relationship between pattern grime and quality attributes.

Finally, the findings reported in this chapter and Chapter 6 answer the last open research question of the PhD project. Therefore, the next chapter concludes this dissertation, summarizing the entirety of the work and results, as well as discussing opportunities for future work.

Chapter 8

Conclusions and Future Work

This chapter presents the conclusions of the PhD dissertation. In Section 8.1, the research questions posed in Chapter 1 are revisited and answered according to the findings of the empirical studies reported in Chapters 2 to 7. In the same section, the contributions of this dissertation, compared to the state of the art, are summarized. Finally, the perspectives for future work are described in Section 8.2.

8.1 Research Questions and Contributions

In Chapter 1 we formulated the main problem statement of this dissertation: "*Despite the growing body of knowledge for engineering CESs, their design process is still challenging. This is especially true due to their complexity, and hard requirements regarding critical quality attributes. Furthermore, it usually involves complex trade-offs for both critical and noncritical quality attributes. However, we currently lack practices that can support the design of CES while managing quality attributes and their trade-offs*". To address this problem, Chapter 1 also presents six research questions that were derived based on the Design Science framework, and were answered in Chapters 2 to 7. Table 8.1 repeats these research questions, lists the corresponding chapters in which they are answered, and summarizes the contribution of each chapter compared to the state-of-the-art.

To summarize a response to the problem statement, the findings reported in this thesis suggest that, when applicable, GoF patterns can promote critical quality attributes while also supporting the management of noncritical quality attributes. However, phenomena such as pattern grime and factors such as the logical and design complexity of pattern instances can affect the extent of the benefits that GoF patterns could otherwise promote. To elaborate on the response, each research question presented in Table 8.1 is briefly answered in the following.

RQ_1: Are there trade-offs when dealing with quality attributes in CES?

This research question was formulated as one part of the problem space exploration (the second part is covered by RQ_2), with the goal of empirically investi-

Table 8.1: Contributions of the PhD dissertation

Research Question	**Chapter**	**Contributions compared to the state-of-the-art**
RQ$_1$: Are there trade-offs when dealing with quality attributes in CES?	2	It compares trade-offs that appear in CESs with other application domains. Furthermore, it investigates the interplay among nine quality attributes. To the best of our knowledge, this is the first study that presents empirical evidence on such trade-offs and it is also the most inclusive study of this type in terms of investigated quality attributes.
RQ$_2$: How are CES designed?	3	It provides: a) a classification of the existing approaches to design CES; b) a list of tools for supporting these approaches; c) a list of domains for which these approaches have been developed and used; d) a list of the most commonly identified CQAs in the CES design; e) a classification of these approaches, based on the level of their empirical evidence.
RQ$_{3.a}$: How do patterns affect runtime quality attributes?	4	It investigates three runtime quality attributes using static analysis, providing evidence on their potential relationship to the application of design patterns. It identifies similarities and differences between the results obtained by static analysis and those obtained by dynamic analysis, increasing the validity of evidence on the subject, as well as adding to the current state of the art on analysis of runtime quality attributes. This is, to the best of our knowledge, the first study that provides empirical evidence on the relationship between the use of GoF patterns and security.
RQ$_{3.b}$: How do patterns influence energy consumption?	5	It considers two nontrivial systems and a considerable amount of pattern instances and pattern-related methods. This setup allows to observe realistic results that are more representative to the population of existing software-intensive systems. Furthermore, it analyses measurements at both process-level and method-level, which allows for comparing results at different levels of granularity. Finally, it investigates not only the energy efficiency of pattern instances, but also the design parameters that render them either beneficial or not.
RQ$_{4.a}$: How does pattern grime evolves?	6	It investigates how pattern grime is accumulated by different developers (16 in total) during the development of five-industrial nontrivial projects (in contrast to open-source ones). To the best of our knowledge, the focus on developers has not been considered in previous studies, while the industrial validation is also a first. Furthermore, it studies the correlation between pattern grime and multiple structural metrics of pattern instances, which has not been thoroughly explored in previous studies.
RQ$_{4.b}$: How is pattern grime related to runtime quality attributes?	7	It investigates three quality attributes (i.e., performance, security and correctness) that have not been previously addressed in this context. Furthermore, it studies five industrial nontrivial projects that collectively provided 36,571 units of analysis (i.e., modifications to the source code of pattern instances). Finally, it investigates three factors, namely project, type of pattern, and developer; although these have been explored with regards to the accumulation of grime, they have not yet been examined regarding the relation between grime and quality attributes.

gating the existence of quality trade-offs, on the implemented architecture (i.e. the system implementation), among versions of open source CESs, and compare such trade-offs with those of systems from other application domains. This question was answered in Chapter 2, where a case study that addresses this challenge was reported. The results suggest the existence of possible trade-offs between critical quality attributes (correctness, security, and performance), as well as the fact that noncritical quality attributes (e.g., reusability, understandability, etc.) are usually compromised in favor of critical quality attributes. The latter was observed more prominently in the CES domain. On the contrary, we have not observed critical quality attributes compromised in favor of noncritical ones, for either CES or other application domains.

***RQ_2*: How are CES designed?**

To complete the problem space exploration, this second research question was defined with the goal of providing a fair overview on CESs design approaches. The answer to this question was presented in Chapter 3, where a Systematic Mapping Study (SMS) was reported. In this SMS, a total of 1,673 papers were collected from five digital libraries, and 269 primary studies were filtered in the sequence. This study analyzed five facets: design approaches, applications domains, critical quality attributes, tools, and type of evidence. The findings suggested that the body of knowledge is vast and overlaps with other types of systems (e.g., real-time or cyber-physical systems). In addition, some critical quality attributes, such as performance and security, are common among various application domains. The results also suggested that many approaches and tools are often generic to CES and do not specialize for domains, such as automotive or avionics. Finally, Chapter 3 also highlighted a few approaches that could be potentially beneficial to CES and have not been thoroughly explored yet, like using design patterns to improve levels of critical quality attributes.

***$RQ_{3.a}$*: How do patterns affect runtime quality attributes?**

Design patterns were discerned in the previous answer as a promising approach to manage qualities in CES; an approach that has not been thoroughly explored so far. Therefore, this research question aimed at studying whether the presence of patterns enforces the conformance to good coding practices, thus promoting critical quality attributes. The answer to this question was presented in Chapter 4, in which the reported case study explored the relationship between the presence of GoF design patterns and violations of good coding practices (related to three critical runtime quality attributes: correctness, performance and security). The results suggested that classes not participating in patterns are more probable to violate good coding practices for correctness, performance and security. In a more fine-grained level of analysis, by focusing on specific patterns, the findings indicated that patterns with more complex structure (e.g., Decorator) and pattern roles that are more change-prone (e.g., Subclasses) are more likely to be associated with a higher number of violations (up to 50 times more violations when compared against other pattern types or roles).

RQ*$_{3.b}$: *How do patterns influence energy consumption?

The previous answer shed light on the matter of using design patterns to manage critical quality attributes. This research questions aimed at extending the previous investigation (Chapter 4) by focusing on a specific runtime quality attribute that has received increasing attention over the past years: energy consumption. This question was answered in Chapter 5 through an experiment that examined pattern-participating methods (i.e., those that play a role within the pattern) and compared their energy consumption to the consumption of functionally equivalent alternative (non-pattern) solutions. The results suggested that the alternative solution excels the pattern solution in the majority of the cases. However, exceptions in which the pattern solution had similar or even slightly lower energy consumption than the alternative solution were also identified. By further analyzing the design parameters of these exceptions, it was identified that the pattern solution was as energy efficient or more energy efficient than the alternative solution in case of large methods and/or methods with higher number of method invocations. This suggested that the studied patterns are more suitable energy-wise when more complex behaviors have to be implemented.

RQ*$_{4.a}$: *How does pattern grime evolves?

The previous investigations suggested that design patterns can be a viable solution to safeguard quality attributes in CES development. However, design patterns, just like other design solutions, are not immune to degradation throughout software evolution. The addition of functionality to pattern instances may distance their design from the original definition, which is defined as pattern grime. As this phenomenon had not been thoroughly explored, this research question aimed at studying the relationship between grime and various related factors. The answer to this question was presented in Chapter 6, which reported a case study on investigating the existence of relations between the accumulation of grime in pattern instances and various related factors: (a) projects, (b) pattern types, (c) developers, and (d) the structural characteristics of the pattern participating classes. The reported results suggested that pattern grime tends to increase linearly, and that it is likely independent of project but dependent of pattern type and developer. Moreover, Chapter 6 also presents a series of correlations between metrics for pattern grime and structural characteristics, e.g., the coupling added to pattern participants tends to also introduce grime.

***$RQ_{4.b}$*: How is pattern grime related to runtime quality attributes?**
The accumulation of pattern grime observed in the previous study indicated that this phenomenon could have undesired effects on pattern instances. Therefore, this research question focused on investigating whether there is a relation between three forms of pattern grime (i.e., organizational, modular, and class) and the levels of three quality attributes addressed in previous research questions, namely performance, security and correctness. The question was answered in Chapter 7 through a case study with five industrial projects (approx. 260,000 lines of code) implemented by 16 developers. The results suggested that pattern grime is correlated to the depreciation of correctness, performance and security in pattern instances. These findings were established on both class and modular grime, whilst no strong evidence was observed on organizational grime. The results also suggest that all three examined factors (i.e., project, pattern type, and developers) can influence the relationship between pattern grime and quality attributes.

8.2 Future Work

Based on the findings, scope and limitations of the empirical studies carried out during the PhD, several opportunities of future work could be identified. These opportunities are described in the following, grouped into five main directions: (a) creation of a recommendation system based on the knowledge obtained during the PhD, (b) extension of the research scope to other approaches and programming languages, (c) investigation of patterns and granularity of this investigation, (d) investigation of quality attributes, metrics and tools, and (e) more in-depth study of pattern grime and other phenomena.

8.2.1 Pattern Recommendation System

The main outcome of this project is that design patterns can be considered as a promising design asset for developing critical embedded systems, in the sense that: (a) they promote design-time qualities; and (b) under certain circumstances they avoid deterioration of (or can even become beneficial for) runtime quality attributes. Throughout this dissertation a set of parameters and factors that can affect the decision-making process have been unveiled. Thus, a recommendation system that will consider all the identified parameters and guide practitioners' design decision making is expected to be of significant value. Such a system would consider not only the structural parameters discussed in this thesis, but also the applicability of the patterns in the given context and design problems.

8.2.2 Scope of Studies

All the studies that have been performed in this PhD project are based on artifact analysis, originating mostly from software systems implemented in the Java programming language. An interesting extension would be the replication of studies with projects written in other languages often used for developing embedded systems, such as C and C++. To allow such replications, the tools used in the reported studies would have to be adapted or replaced with equivalent tools. An adaptation of replacement would have to provide the following features for the target language: (a) estimate metrics for the investigated quality attributes; (b) detect the investigated patterns, including extended pattern participants; (c) estimate pattern grime; and (d) estimate energy consumption at class and method levels. The main challenge in creating or adapting tools is that the languages have different Abstract Syntax Trees (ASTs). However, one solution to overcome this challenge could be to take advantage of recent standards or technologies for language-independent parsing such as OMG's Generic Abstract Syntax Tree Meta-model[1] (Canovas and Molina, 2010; Fleck et al., 2016) and Oracle's Graal+Truffle[2] (Grimmer et al., 2015; Azadmanesh et al., 2017). Furthermore, it would be interesting to consider larger populations in future work in order to discuss the impact of higher numbers of subjects to the results.

Regarding the SMS reported in Chapter 3, the body of knowledge that was presented has considerable overlap with other classes of system (e.g., hard-real time systems). Thus, it would be relevant to continue exploring such related classes and seek other approaches that could be applied to the designing of CESs.

8.2.3 Exploration of Other Patterns

The research conducted in this PhD project focused only on GoF design patterns, as they are the most famous patterns catalog that is applicable in any object-oriented language, and may comprise a large size of a system. However, GoF are not the only available patterns in the software literature. Therefore, it is relevant to consider

[1] https://www.omg.org/spec/ASTM

[2] https://graalvm.org

studying other types of patterns, such as POSA, small-memory patterns, etc. which are of great interest for embedded systems design.

Regarding GoF patterns, there are also several opportunities for extending the work reported in this dissertation. First, not all GoF patterns were considered in the reported studies and, thus, similar studies addressing additional patterns could contribute to the state-of-the-art. In the particular case of the study involving energy consumption, it would be beneficial to focus on the other two pillars of object-orientation that have not been investigated in depth (i.e., encapsulation and inheritance). Furthermore, the analysis of the collected data could also be extended by, for example, applying different normalizations e.g. in terms of project size or functionality. Finally, it would be interesting to compare the overall level of quality attributes across projects that have varying numbers of pattern instances.

8.2.4 Exploration of Quality Attributes

This PhD project focused on a subset of quality attributes and, in particular, a subset of critical quality attributes, namely correctness, performance and security. Therefore, it is interesting to extend the work reported in this dissertation by investigating additional quality attributes. Furthermore, each quality attribute is estimated using specific metrics, and the exploration of different metrics is valuable to further validate the results presented in this dissertation. Finally, to assess the various quality attributes addressed in this PhD, different tools were employed. Although the tools had been deemed adequate, the use of alternative tools, could contribute to not only triangulate the results, but also to investigate additional effects or relationships.

Regarding the empirical analysis reported in this dissertation, two main extensions should be mentioned. First, FindBugs reports a severity level for each detectable violation of good coding practices, which was not particularly explored. Thus, it would be interesting to extend the current work to factor in the severity of violations. Second, in the study involving energy consumption, popular open-source systems of various domains were analyzed, but they are not particularly tailored for energy efficiency. Thus, it would be interesting to focus also on systems that have energy efficiency among their main concerns, investigating how GoF patterns and alternatives are used within such a context and comparing these systems with other kinds of systems.

8.2.5 Pattern Grime and Beyond

Some of the investigated facets on how pattern grime accumulates can and should be further explored, e.g., what factors may be related to developers that tend to accumulate more or less grime. The investigation of additional grime metrics and factors could enhance the understanding over the consequences of accumulating pattern grime. In particular, metrics regarding subtypes of grime have been proposed in the literature and their exploration could shed light into the interplay between indicators of the types and subtypes of grime in similar study settings. Furthermore, work based on the correlation between pattern grime and structural metrics raised questions that can be investigated in confirmatory studies, e.g., whether most introduced afferent coupling is indeed resulting in the accumulation of grime.

Regarding the relationship between the accumulation of patter grime and levels of quality attributes, confirmatory empirical studies could investigate one or more of the factors explored in the PhD in more details, and seek evidence to explain the observed variations in this relationship. Moreover, the studies involving pattern grime reported in this dissertation considered industrial systems. Although open-source software has been studied in related work, there is a lack of studies with similar design. Such a study can increase the external validity of the results reported on this dissertation.

Finally, pattern grime is not the only phenomenon that can lead to the decay of pattern instances. Well-known phenomena such as technical debt can have particular effects on design pattern instances. In this regard, it is interesting to investigate the extent of the decay and how it can influence the relationship between the application of design pattern and the levels of quality attributes.

Appendix A

A.1 Supplementary Material to Chapter 3

In this following, we present the list of primary studies included in the systematic mapping study (SMS) reported in Chapter 3. We assigned an ID to each primary study, which are used in the reporting of the SMS when referencing the studies. The ID's are in the format $[S < number >]$.

[S1] Gove, R. and Heinzman, J. L.: 1991, Safety criteria and model for mission-critical embedded software systems, *Proceedings of the Sixth Annual Conference on Computer Assurance, Systems Integrity, Software Safety and Process Security (COMPASS '91)*, IEEE, pp. 69–73. **DOI:** *10.1109/CMPASS.1991.161041*

[S2] Fidge, C. J. and Lister, A. M.: 1992, Disciplined approach to real-time systems design, *Information and Software Technology* **34**(9), 603–610. **DOI:** *10.1016/0950-5849(92)90137-e*

[S3] Lutz, R. R.: 1993, Analyzing software requirements errors in safety-critical, embedded systems, *Proceedings of First IEEE International Symposium on Requirements Engineering (RE '93)*, IEEE, pp. 126–133. **DOI:** *10.1109/ISRE.1993.324825*

[S4] Kelly, J. C. and Covington, R. G.: 1994, Results of a formal methods demonstration project, *Proceedings of the WESCON '94 Conference*, IEEE, pp. 62–66. **DOI:** *10.1109/WESCON.1994.403627*

[S5] Vardanega, T.: 1994, Experience with the development of hard real-time embedded Ada software, *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*, IEEE Computer Society Press, pp. 301–308. **DOI:** *10.1109/ICSE.1994.296792*

[S6] Corbett, J. C.: 1996, Timing analysis of Ada tasking programs, *IEEE Transactions on Software Engineering* **22**(7), 461–483. **DOI:** *10.1109/32.538604*

[S7] Baufreton, P., Méhaut, X. and Rutten, É.: 1997, Embedded systems in avionics and the SACRES approach, *Proceedings of the 16th International Conference on Computer Safety, Reliability and Security (SAFECOMP '97)*, Springer, pp. 311–320. **DOI:** *10.1007/978-1-4471-0997-6_24*

[S8] Edwards, S., Lavagno, L., Lee, E. A. and Sangiovanni-Vincentelli, A.: 1997, Design of embedded systems: formal models, validation, and synthesis, *Proceedings of the IEEE* **85**(3), 366–390. **DOI:** *10.1109/5.558710*

[S9] Heimdahl, M. P. and Thompson, J. M.: 1997, Specification and analysis of system level inter-component communication, *Proceedings of the First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, IEEE, pp. 192–201. **DOI:** *10.1109/2.666842*

[S10] Büssow, R., Geisler, R. and Klar, M.: 1998, Specifying safety-critical embedded systems with statecharts and Z: A case study, *Fundamental Approaches to Software Engineering*, Springer, pp. 71–87. **DOI:** *10.1007/bfb0053584*

[S11] Dal Cin, M.: 1998, Modeling fault-tolerant system behavior, *Systems: Theory and Practice*, Springer, pp. 213–234. **DOI:** *10.1007/978-3-7091-6451-8_10*

[S12] Hollingworth, K. and Saeed, A.: 1998, CoRSA-A Constraint Based Approach to Requirements and Safety Analysis, *Computer Safety, Reliability and Security*, Springer, pp. 3–15. **DOI:** *10.1007/3-540-49646-7_1*

[S13] Muscettola, N., Nayak, P. P., Pell, B. and Williams, B. C.: 1998, Remote agent: To boldly go where no AI system has gone before, *Artificial Intelligence* **103**(1), 5–47. **DOI:** *10.1016/s0004-3702(98)00068-x*

[S14] von Hanxleden, R., Botorabi, A. and Kupczyk, S.: 1998, A codesign approach for safety-critical automotive applications, *IEEE Micro* **18**(5), 66–79. **DOI:** *10.1109/40.735945*

[S15] Winter, K., Santen, T. and Heisel, M.: 1998, An agenda for specifying software components with complex data models, *Computer Safety, Reliability and Security*, Springer, pp. 16–31. **DOI:** *10.1007/3-540-49646-7_2*

[S16] Bienmüller, T., Brockmeyer, U., Damm, W., Döhmen, G., Eßmann, C., Holberg, H.-J., Hungar, H., Josko, B., Schlör, R., Wittich, G. et al.: 1999, Formal verification of an avionics application using abstraction and symbolic model checking, *Towards System Safety*, Springer, pp. 150–173. **DOI:** *10.1007/978-1-4471-0823-8_10*

[S17] Bryant, S. E. and Key, K.: 1999, Redefining the process for development of embedded software, *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design (CACSD '99)*, IEEE, pp. 261–266. **DOI:** *10.1109/CACSD.1999.808658*

[S18] Cin, M. D., Huszerl, G. and Kosmidis, K.: 1999, Quantitative evaluation of dependability critical systems based on guarded statechart models, *Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering (HASE '99)*, IEEE, pp. 37–45. **DOI:** *10.1109/HASE.1999.809473*

[S19] Heiner, M. and Heisel, M.: 1999, Modeling safety-critical systems with Z and Petri nets, *Computer Safety, Reliability and Security*, Springer, pp. 361–374. **DOI:** *10.1007/3-540-48249-0_31*

[S20] Kandasamy, N., Hayes, J. P. and Murray, B. T.: 1999, Tolerating transient faults in statically scheduled safety-critical embedded systems, *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS '99)*, IEEE, pp. 212–221. **DOI:** *10.1109/RELDIS.1999.805097*

[S21] Thompson, J. M., Heimdahl, M. P. and Miller, S. P.: 1999, Specification-Based Prototyping for Embedded Systems, *Proceedings of the Seventh Joint Meeting of The European Software Engineering Conference and the ACM Sigsoft Symposium on the Foundations of Software Engineering (ESEC/FSE '99)*, Springer, pp. 163–179. **DOI:** *10.1007/3-540-48166-4_11*

[S22] Lajolo, M., Rebaudengo, M., Roerda, M. S., Violante, M. and Lavagno, L.: 2000, Evaluating system dependability in a co-design framework, *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '00)*, ACM, pp. 586–590. **DOI:** *10.1145/343647.343861*

[S23] Seward, D., Pace, C., Morrey, R. and Sommerville, I.: 2000, Safety analysis of autonomous excavator functionality, *Reliability Engineering & System Safety* **70**(1), 29–39. **DOI:** *10.1016/s0951-8320(00)00045-4*

[S24] Claesson, V., Lönn, H. and Suri, N.: 2001, Efficient TDMA synchronization for distributed embedded systems, *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems (SRDS '01)*, IEEE, pp. 198–201. **DOI:** *10.1109/RELDIS.2001.970769*

[S25] Doche, M., Vernier-Mounier, I. and Kordon, F.: 2001, A modular approach to the specification and validation of an electrical flight control system, *Proceed-*

ings ot the International Symposium of Formal Methods Europe (FME '01), Springer, pp. 590–610. **DOI:** *10.1007/3-540-45251-6_34*

[S26] Grieskamp, W., Heisel, M. and Dörr, H.: 2001, Specifying embedded systems with statecharts and Z: an agenda for cyclic software components, *Science of Computer Programming* **40**(1), 31–57. **DOI:** *10.1016/s0167-6423(00)00024-1*

[S27] Liu, J., Chou, P. H., Bagherzadeh, N. and Kurdahi, F.: 2001, Power-aware scheduling under timing constraints for mission-critical embedded systems, *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*, ACM, pp. 840–845. **DOI:** *10.1145/378239.379076*

[S28] Winkelmann, K.: 2001, Formal Methods in Designing Embedded Systems—the SACRES Experience, *Formal Methods in System Design* **19**(1), 81–110. **DOI:** *10.1023/A:1011295931367*

[S29] Chou, P. H., Liu, J., Li, D. and Bagherzadeh, N.: 2002, Impacct: Methodology and tools for power-aware embedded systems, *Design Automation for Embedded Systems* **7**(3), 205–232. **DOI:** *10.1023/A:1019730322551*

[S30] Garriou, D.: 2002, Symbolic simulation of synchronous programs, *Electronic Notes in Theoretical Computer Science* **65**(5), 11–18. **DOI:** *10.1016/s1571-0661(05)80436-0*

[S31] Karsai, G., Neema, S., Abbott, B. and Sharp, D.: 2002, A modeling language and its supporting tools for avionics systems, *Proceedings of the 21st Digital Avionics Systems Conference (DASC '02)*, Vol. 1, IEEE, pp. 6A3–1. **DOI:** *10.1109/DASC.2002.1067981*

[S32] Bate, I. and Burns, A.: 2003, An integrated approach to scheduling in safety-critical embedded control systems, *Real-Time Systems* **25**(1), 5–37. **DOI:** *10.1023/A:1022920502619*

[S33] Choi, Y. and Heimdahl, M.: 2003, Model checking software requirement specifications using domain reduction abstraction, *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE '03)*, IEEE, pp. 314–317. **DOI:** *10.1109/ASE.2003.1240328*

[S34] Dajani-Brown, S., Cofer, D., Hartmann, G. and Pratt, S.: 2003, Formal modeling and analysis of an avionics triplex sensor voter, *Model Checking Software*, Springer, pp. 34–48. **DOI:** *10.1007/3-540-44829-2_3*

[S35] Kopetz, H. and Bauer, G.: 2003, The time-triggered architecture, *Proceedings of the IEEE* **91**(1), 112–126. **DOI:** *10.1109/jproc.2002.805821*

[S36] Obermaisser, R. and Peti, P.: 2003, A framework for rapid application development of distributed embedded real-time systems, *Proceedings of the 15th IEEE International Conference on Computer as a Tool (EUROCON '03)*, Vol. 1, IEEE, pp. 80–84. **DOI:** *10.1109/EURCON.2003.1247983*

[S37] Tsai, W.-T., Yu, L., Zhu, F. and Paul, R.: 2003, Rapid verification of embedded systems using patterns, *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC '03)*, IEEE, pp. 466–471. **DOI:** *10.1109/CMPSAC.2003.1245381*

[S38] Wang, L.: 2003, Fault handling in embedded industrial measurement and control systems: issues and a case study, *Proceedings of the 2003 IEEE Systems Readiness Technology Conference (AUTOTESTCON '03)*, IEEE, pp. 713–719. **DOI:** *10.1109/AUTEST.2003.1243657*

[S39] Yang, S., Sang, N. and Xiong, G.: 2003, Integrated safety critical systems on reliable real time network, *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '03)*, IEEE, pp. 66–70. **DOI:** *10.1109/PDCAT.2003.1236260*

[S40] Zhang, Y. and Chakrabarty, K.: 2003, Fault recovery based on checkpointing for hard real-time embedded systems, *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '03)*, IEEE, pp. 320–327. **DOI:** *10.1109/DFTVS.2003.1250127*

[S41] Coyle, E., Maguire, L. and McGinnity, T.: 2004, Self-repair of embedded systems, *Engineering Applications of Artificial Intelligence* **17**(1), 1–9. **DOI:** *10.1016/j.engappai.2003.11.009*

[S42] Dion, B., Le Sergent, T., Martin, B. and Griebel, H.: 2004, Model-based development for time-triggered architectures, *Proceedings of the 23rd Digital Avionics Systems Conference (DASC '04)*, Vol. 2, IEEE, pp. 6–D. **DOI:** *10.1109/DASC.2004.1390733*

[S43] Durrieu, G., Laurent, O., Seguin, C. and Wiels, V.: 2004, Formal proof and test case generation for critical embedded systems using scade, *Building the Information Society*, Springer, pp. 499–504. **DOI:** *10.1007/978-1-4020-8157-6_44*

[S44] Gopalakrishnan, S., Sha, L. and Caccamo, M.: 2004, Hard real-time communication in bus-based networks, *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, IEEE, pp. 405–414. **DOI:** *10.1109/REAL.2004.24*

[S45] Hansson, H., Åkerholm, M., Crnkovic, I. and Törngren, M.: 2004, SaveCCM-a component model for safety-critical real-time systems, *Proceedings os the 30th Euromicro Conference*, IEEE, pp. 627–635. **DOI:** *10.1109/eurmic.2004.1333431*

[S46] Konrad, S., Cheng, B. H. and Campbell, L. A.: 2004, Object analysis patterns for embedded systems, *IEEE Transactions on Software Engineering* **30**(12), 970–992. **DOI:** *10.1109/tse.2004.102*

[S47] Lavagno, L., Di Natale, M., Ferrari, A. and Giusto, P.: 2004, SoftContract: model-based design of error-checking code and property monitors, *UML Modeling Languages and Applications*, Springer, pp. 150–162. **DOI:** *10.1007/978-3-540-31797-5_16*

[S48] Maheshwari, A., Burleson, W. and Tessier, R.: 2004, Trading off transient fault tolerance and power consumption in deep submicron (DSM) VLSI circuits, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **12**(3), 299–311. **DOI:** *10.1109/tvlsi.2004.824302*

[S49] Morris, J., Kroening, D. and Koopman, P.: 2004, Fault tolerance tradeoffs in moving from decentralized to centralized embedded systems, *Proceedings of the International Conference on Dependable Systems and Networks (DSN '93)*, IEEE, pp. 377–386. **DOI:** *10.1109/dsn.2004.1311907*

[S50] Ortmeier, F., Thums, A., Schellhorn, G. and Reif, W.: 2004, Combining formal methods and safety analysis–the ForMoSA approach, *Integration of Software Specification Techniques for Applications in Engineering*, Springer, pp. 474–493. **DOI:** *10.1007/978-3-540-27863-4_26*

[S51] Pop, P., Eles, P., Peng, Z. and Pop, T.: 2004, Analysis and optimization of distributed real-time embedded systems, *ACM Transactions on Design Automation of Electronic Systems* **11**(3), 593–625. **DOI:** *10.1145/1142980.1142984*

[S52] Schinz, I., Toben, T., Mrugalla, C. and Westphal, B.: 2004, The Rhapsody UML verification environment, *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM '04)*, IEEE, pp. 174–183. **DOI:** *10.1109/SEFM.2004.1347518*

[S53] Thiele, L. and Wilhelm, R.: 2004, Design for timing predictability, *Real-Time Systems* **28**(2-3), 157–177. **DOI:** *10.1023/b:time.0000045316.66276.6e*

[S54] Wu, B., Wu, Z. and Chen, W.: 2004, Component model optimization for distributed real-time embedded software, *Proceedings of the 2004 IEEE International Conference on Systems, Man and Cybernetics (ICSMC '04)*, Vol. 2, IEEE, pp. 1158–1163. **DOI:** *10.1109/ICSMC.2004.1399780*

[S55] Zhang, Y. and Chakrabarty, K.: 2004, Dynamic adaptation for fault tolerance and power management in embedded real-time systems, *ACM Transactions on Embedded Computing Systems* **3**(2), 336–360. **DOI:** *10.1145/993396.993402*

[S56] Caffall, D. S. and Michael, J. B.: 2005, Formal methods in a system-of-systems development, *Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics (ICSMC '05)*, Vol. 2, IEEE, pp. 1856–1863. **DOI:** *10.1109/ICSMC.2005.1571417*

[S57] de Freitas Francisco, A. L. and Rammig, F. J.: 2005, Fault-tolerant hard-real-time communication of dynamically reconfigurable, distributed embedded systems, *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05)*, IEEE, pp. 275–283. **DOI:** *10.1109/ISORC.2005.27*

[S58] Feiler, P. H., Lewis, B., Vestal, S. and Colbert, E.: 2005, An overview of the SAE architecture analysis & design language (AADL) standard: a basis for model-based architecture-driven embedded systems engineering, *Architecture Description Languages*, Springer, pp. 3–15. **DOI:** *10.1007/0-387-24590-1_1*

[S59] Guerrouat, A. and Richter, H.: 2005, A formal approach for analysis and testing of reliable embedded systems, *Electronic Notes in Theoretical Computer Science* **141**(3), 91–106. **DOI:** *10.1016/j.entcs.2005.02.050*

[S60] Hewett, R. and Seker, R.: 2005, A risk assessment model of embedded software systems, *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop (SEW '05)*, IEEE, pp. 142–149. **DOI:** *10.1109/SEW.2005.16*

[S61] Latronico, E. and Koopman, P.: 2005, Design time reliability analysis of distributed fault tolerance algorithms, *Proceedings of the International Conference on Dependable Systems and Networks (DSN '05)*, IEEE, pp. 486–495. **DOI:** *10.1109/DSN.2005.38*

[S62] Obermaisser, R.: 2005, Ordering messages in virtual CAN networks, *Proceedings of the 12th IEEE International Conference on Electronics, Circuits and Systems (ICECS '05)*, IEEE, pp. 1–4. **DOI:** *10.1109/icecs.2005.4633552*

[S63] Obermaisser, R., Peti, P. and Kopetz, H.: 2005, Virtual networks in an integrated time-triggered architecture, *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '05)*, IEEE, pp. 241–253. **DOI:** *10.1109/WORDS.2005.55*

[S64] Pai, G., Bechta-Dugan, J. and Lateef, K.: 2005, Bayesian Networks applied to

Software IV & V, *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop (SEW '05)*, IEEE, pp. 293–304. **DOI:** *10.1109/SEW.2005.20*

[S65] Peti, P., Obermaisser, R., Tagliabo, F., Marino, A. and Cerchio, S.: 2005, An integrated architecture for future car generations, *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05)*, IEEE, pp. 2–13. **DOI:** *10.1109/ISORC.2005.12*

[S66] Pop, P., Eles, P. and Peng, Z.: 2005, Analysis and optimisation of heterogeneous real-time embedded systems, *IEE Proceedings - Computers and Digital Techniques* **152**(2), 130–147. **DOI:** *10.1049/ip-cdt:20045069*

[S67] Silva, V., Marau, R., Almeida, L., Ferreira, J., Calha, M., Pedreiras, P. and Fonseca, J.: 2005, Implementing a distributed sensing and actuation system: The CAMBADA robots case study, *Proceedings of the 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA '05)*, Vol. 2, IEEE, pp. 8–pp. **DOI:** *10.1109/ETFA.2005.1612753*

[S68] Tavares, E., Maciel, P., Bessa, A., Barreto, R., Barros, L., Oliveira Jr, M. and Lima, R.: 2005, A time petri net based approach for embedded hard real-time software synthesis with multiple operational modes, *Proceedings of the 18th Symposium on Integrated Circuits and Systems Design (SBCCI '05)*, IEEE, pp. 98–103. **DOI:** *10.1145/1081081.1081110*

[S69] Benoit, E., Chovin, A., Foulloy, L., Chatenay, A. and Mauris, G.: 2006, Toward a safe design of CANopen distributed instruments, *IEEE Transactions onInstrumentation and Measurement* **55**(3), 771–777. **DOI:** *10.1109/tim.2006.873798*

[S70] Buckl, C., Knoll, A. and Schrott, G.: 2006, Model-based development of fault-tolerant embedded software, *Proccedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '06)*, IEEE, pp. 103–110. **DOI:** *10.1109/isola.2006.22*

[S71] Choi, K.-S., Jung, S.-C., Kim, H.-J., Bae, D.-H. and Lee, D.-H.: 2006, UML-based Modeling and Simulation Method for Mission-Critical Real-Time Embedded System Development., *Proceedings of the IASTED International Conference on Software Engineering*, Vol. 2006, pp. 160–165.

[S72] Goldsby, H., Cheng, B. H., Konrad, S. and Kamdoum, S.: 2006, A visualization framework for the modeling and formal analysis of high assurance systems, *Model Driven Engineering Languages and Systems*, Springer, pp. 707–721. **DOI:** *10.1007/11880240_49*

[S73] Guerrouat, A. and Richter, H.: 2006, A component-based specification approach for embedded systems using FDTs, *ACM SIGSOFT Software Engineering Notes* **31**(2), 14. **DOI:** *10.1145/1118537.1123073*

[S74] Miller, S. P., Tribble, A. C., Whalen, M. W. and Heimdahl, M. P.: 2006, Proving the shalls, *International Journal on Software Tools for Technology Transfer* **8**(4-5), 303–319. **DOI:** *10.1007/s10009-004-0173-6*

[S75] Obermaisser, R. and Peti, P.: 2006, A fault hypothesis for integrated architectures, *Proceedings of the Fourth International Workshop on Intelligent Solutions in Embedded Systems (WISES '06)*, IEEE, pp. 1–18. **DOI:** *10.1109/wises.2006.329115*

[S76] Oswald, N.: 2006, Towards a conceptual framework-based architecture for unmanned systems, *Informatics in Control, Automation and Robotics I*, Springer, pp. 167–177. **DOI:** *10.1007/1-4020-4543-3_20*

[S77] Ryan, C., Heffernan, D. and Leen, G.: 2006, Interactive consistency on a time-triggered real-time control network, *IEEE Transactions on Industrial Informatics* **2**(4), 242–254. **DOI:** *10.1109/tii.2006.885189*

[S78] Schoitsch, E., Althammer, E., Eriksson, H., Vinter, J., Gönczy, L., Pataricza, A. and Csertan, G.: 2006, Validation and Certification of Safety-Critical Embedded Systems–The DECOS Test Bench, *Computer Safety, Reliability, and Security*, Springer, pp. 372–385. **DOI:** *10.1007/11875567_28*

[S79] Su, H., Hemingway, G., Chen, K. and Koo, T. J.: 2006, Model-based toolchain infrastructure for automated analysis of embedded systems, *Automated Technology for Verification and Analysis*, Springer, pp. 523–537. **DOI:** *10.1007/11901914_38*

[S80] Villani, E., Miyagi, P. E. and Valette, R.: 2006, Landing system verification based on petri nets and a hybrid approach, *IEEE Transactions on Aerospace and Electronic Systems* **42**(4), 1420–1436. **DOI:** *10.1109/taes.2006.314582*

[S81] Yang, G., Li, H. and Wu, Z.: 2006, SmartC: A component-based hierarchical modeling language for automotive electronics, *Proceedings of the Second IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC '06)*, IEEE, pp. 203–210. **DOI:** *10.1109/dasc.2006.45*

[S82] Azevedo, J. L., Cunha, B. and Almeida, L.: 2007, Hierarchical distributed architectures for autonomous mobile robots: a case study, *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA '07)*, IEEE, pp. 973–980. **DOI:** *10.1109/efta.2007.4416889*

[S83] Banci, M., Fantechi, A., Gnesi, S. and Lombardi, G.: 2007, Experimenting with diversity in the model driven development of a railway signaling system, *Proceedings of the Second International Workshop on Engineering Fault Tolerant Systems (EFTS '07)*, ACM, p. 5. **DOI:** *10.1145/1316550.1316555*

[S84] Barboni, E., Navarre, D., Palanque, P. and Basnyat, S.: 2007, A formal description technique for interactive cockpit applications compliant with ARINC specification 661, *Proceedings of the 12th International Symposium on Industrial Embedded Systems (SIES '07)*, IEEE, pp. 250–257. **DOI:** *10.1109/sies.2007.4297342*

[S85] Buckl, C., Regensburger, M., Knoll, A. and Schrott, G.: 2007, Models for automatic generation of safety-critical real-time systems, *Proceedings of the Second International Conference on Availability, Reliability and Security (ARES '07)*, IEEE, pp. 580–587. **DOI:** *10.1109/ares.2007.106*

[S86] de las Heras, E. and Villar, E.: 2007, Specification for SystemC-AADL interoperability, *Proceedings of the Fifth Workshop on Intelligent Solutions in Embedded Systems (WISES '07)*, IEEE, pp. 76–86. **DOI:** *10.1109/wises.2007.4408490*

[S87] Gamatié, A., Gautier, T., Guernic, P. L. and Talpin, J.-P.: 2007, Polychronous design of embedded real-time applications, *ACM Transactions on Software Engineering and Methodology* **16**(2), 9. **DOI:** *10.1145/1217295.1217298*

[S88] Gu, Z., He, X. and Yuan, M.: 2007, Optimization of static task and bus access schedules for time-triggered distributed embedded systems with model-checking, *Proceedings of the 44th Annual Design Automation Conference (DAC '07)*, ACM, pp. 294–299. **DOI:** *10.1145/1278480.1278556*

[S89] Heitmeyer, C. L. and Jeffords, R. D.: 2007, Applying a formal requirements method to three NASA systems: Lessons learned, *Proceedings of the 2007 IEEE Aerospace Conference (AeroConf '07)*, IEEE, pp. 1–10. **DOI:** *10.1109/aero.2007.352764*

[S90] Islam, S. and Suri, N.: 2007, A multi variable optimization approach for the design of integrated dependable real-time embedded systems, *Embedded and Ubiquitous Computing*, Springer, pp. 517–530. **DOI:** *10.1007/978-3-540-77092-3_45*

[S91] Iwu, F., Galloway, A., McDermid, J. and Toyn, I.: 2007, Integrating safety and formal analyses using UML and PFS, *Reliability Engineering & System Safety* **92**(2), 156–170. **DOI:** *10.1016/j.ress.2005.11.060*

[S92] Natale, M. D.: 2007, Virtual platforms and timing analysis: status, challenges and future directions, *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC '07)*, IEEE, pp. 551–555. **DOI:** *10.1145/1278480.1278620*

[S93] Nguyen, K. D., Thiagarajan, P. and Wong, W.-F.: 2007, A UML-based design framework for time-triggered applications, *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS '07)*, IEEE, pp. 39–48. **DOI:** *10.1109/rtss.2007.18*

[S94] Obermaisser, R., Peti, P. and Tagliabo, F.: 2007, An integrated architecture for future car generations, *Real-Time Systems* **36**(1-2), 101–133. **DOI:** *10.1007/s11241-007-9015-4*

[S95] Obermaisser, R. and Schlager, M.: 2007, A Simulation Framework for Virtual Integration of Integrated Systems, *Proceedings of the International Conference on Computer as a Tool (EUROCON '07)*, IEEE, pp. 2208–2216. **DOI:** *10.1109/eurcon.2007.4400256*

[S96] Ponsard, C., Massonet, P., Molderez, J.-F., Rifaut, A., van Lamsweerde, A. and Van, H. T.: 2007, Early verification and validation of mission critical systems, *Formal Methods in System Design* **30**(3), 233–247. **DOI:** *10.1007/s10703-006-0028-8*

[S97] Rosset, V., Souto, P. F. and Vasques, F.: 2007, Formal verification of a group membership protocol using model checking, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, Springer, pp. 471–488. **DOI:** *10.1007/978-3-540-76848-7_34*

[S98] Shukla, S. K., Suhaib, S. M., Mathaikutty, D. A. and Talpin, J.-P.: 2007, On the Polychronous Approach to Embedded Software Design, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, Springer, pp. 261–273. **DOI:** *10.1007/978-1-4020-6254-4_20*

[S99] Squair, M. J.: 2007, Safety, software architecture and MIL-STD-1760, *Proceedings of the 11th Australian Workshop on Safety Critical Systems and Software (SCS '07)*, Australian Computer Society, Inc., pp. 93–112.

[S100] Wang, J., Liu, S., Qi, Y. and Hou, D.: 2007, Developing an insulin pump system using the SOFL method, *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC '07)*, IEEE, pp. 334–341. **DOI:** *10.1109/apsec.2007.41*

[S101] Yang, I., Kim, D., Kang, K., Lee, D. and Yoon, K.: 2007, Smart Actuator-Based Fault-Tolerant Control for Networked Safety-Critical Embedded Systems, *Em-*

bedded Software and Systems, Springer, pp. 615–626. **DOI:** *10.1007/978-3-540-72685-2_57*

[S102] Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P. and Tivoli, M.: 2007, The SAVE approach to component-based development of vehicular systems, *Journal of Systems and Software* **80**(5), 655–667. **DOI:** *10.1016/j.jss.2006.08.016*

[S103] Althammer, E., Schoitsch, E., Sonneck, G., Eriksson, H. and Vinter, J.: 2008, Modular certification support—the DECOS concept of generic safety cases, *Proceedings of the Sixth IEEE International Conference on Industrial Informatics (INDIN '08)*, IEEE, pp. 258–263. **DOI:** *10.1109/indin.2008.4618105*

[S104] Aoyama, M. and Yoshino, A.: 2008, AORE (aspect-oriented requirements engineering) methodology for automotive software product lines, *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC '08)*, IEEE, pp. 203–210. **DOI:** *10.1109/apsec.2008.59*

[S105] Armoush, A., Salewski, F. and Kowalewski, S.: 2008a, Effective pattern representation for safety critical embedded systems, *Proceedings of the 2008 International Conference on Computer Science and Software Engineering (CSSE '08)*, IEEE, pp. 91–97. **DOI:** *10.1109/csse.2008.739*

[S106] Armoush, A., Salewski, F. and Kowalewski, S.: 2008b, Recovery block with backup voting: A new pattern with extended representation for safety critical embedded systems, *Proceedings of the 11th International Conference on Information Technology (ICIT '08)*, IEEE, pp. 232–237. **DOI:** *10.1109/icit.2008.60*

[S107] Balp, H., Borde, É. and Haïk, G.: 2008, Automatic composition of AADL models for the verification of critical component-based embedded systems, *Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '08)*, IEEE, pp. 269–274. **DOI:** *10.1109/iceccs.2008.26*

[S108] Barreto, R., Maciel, P., Tavares, E., Freitas, R. D., Oliveira, M. and Lima, R. M.: 2008, A time Petri net-based method for embedded hard real-time software synthesis, *Design Automation for Embedded Systems* **12**(1-2), 31–62. **DOI:** *10.1007/s10617-007-9011-x*

[S109] Delanote, D., Van Baelen, S., Joosen, W. and Berbers, Y.: 2008, Using AADL to model a protocol stack, *Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '08)*, IEEE, pp. 277–281. **DOI:** *10.1109/iceccs.2008.12*

[S110] Giese, M., Mistrzyk, T., Pfau, A., Szwillus, G. and von Detten, M.: 2008, Amboss: A task modeling approach for safety-critical systems, *Engineering Interactive Systems*, Springer, pp. 98–109. **DOI:** *10.1007/978-3-540-85992-5_8*

[S111] Hall, B., Paulitsch, M., Benson, D. and Behbahani, A.: 2008, Jet engine control using ethernet with a BRAIN, *Proceedings of the 44th AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, Vol. 5291. **DOI:** *10.2514/6.2008-5291*

[S112] Insaurralde, C. C., Seminario, M. A., Jiménez, J. F. and Giron-Sierra, J. M.: 2008, Model-based design analysis of an Avionics Fuel Distributed Control System, *Proceedings of the IEEE/AIAA 27th Digital Avionics Systems Conference (DASC '08)*, IEEE, pp. 5–C. **DOI:** *10.1109/dasc.2008.4702856*

[S113] Liu, X., Liu, X., Li, J., Zhao, Y. and Wang, Z.: 2008, Refinement of UML Interaction for Correct Embedded System Design, *Proceedings of the Ninth International Conference for Young Computer Scientists (ICYCS '08)*, IEEE, pp. 1156–1162. **DOI:** *10.1109/icycs.2008.251*

[S114] Liu, Y. and Wong, T.: 2008, Component architecture and modeling for microkernel-based embedded system development, *Proceedings of the 19th Australian Conference on Software Engineering (ASWEC '08)*, IEEE, pp. 190–199. **DOI:** *10.1109/aswec.2008.4483207*

[S115] Ma, Y., Talpin, J.-P. and Gautier, T.: 2008, Virtual prototyping AADL architectures in a polychronous model of computation, *Proceedings of the 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '08)*, IEEE, pp. 139–148. **DOI:** *10.1109/memcod.2008.4547701*

[S116] Pascoal, E., Rufino, J., Schoofs, T. and Windsor, J.: 2008, AMOBA-ARINC 653 simulator for modular based space applications, *Proceedings of the Eurospace Data Systems in Aerospace Conference (DASIA '08)* **10**, 2.

[S117] Pinello, C., Carloni, L. P. and Sangiovanni-Vincentelli, A. L.: 2008, Fault-tolerant distributed deployment of embedded control software, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **27**(5), 906–919. **DOI:** *10.1109/tcad.2008.917971*

[S118] Sakurai, K., Bokor, P. and Suri, N.: 2008, Aiding modular design and verification of safety-critical time-triggered systems by use of executable formal specifications, *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE '08)*, IEEE, pp. 261–270. **DOI:** *10.1109/hase.2008.45*

[S119] Sveda, M. and Vrba, R.: 2008, Meta-Design Support for Safe and Secure Networked Embedded Systems, *Proceedings of the Third International Conference on*

Systems (ICONS '08), IEEE, pp. 69–74. **DOI:** *10.1109/icons.2008.52*

[S120] Yi, Z., Cai, W. and Yue, W.: 2008, Adaptive safety critical middleware for distributed and embedded safety critical system, *Proceedings of the Fourth International Conference on Networked Computing and Advanced Information Management (NCM '08)*, Vol. 1, IEEE, pp. 162–166. **DOI:** *10.1109/NCM.2008.58*

[S121] Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S. P. and Cofer, D. D.: 2009, A formal architecture pattern for real-time distributed systems, *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS '09)*, IEEE, pp. 161–170. **DOI:** *10.1109/rtss.2009.50*

[S122] Armoush, A., Beckschulze, E. and Kowalewski, S.: 2009, Safety assessment of design patterns for safety-critical embedded systems, *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '09)*, IEEE, pp. 523–527. **DOI:** *10.1109/seaa.2009.12*

[S123] Ayrault, P., Hardin, T. and Pessaux, F.: 2009, Development Life-cycle of Critical Software Under FoCaL, *Electronic Notes in Theoretical Computer Science* **243**, 15–31. **DOI:** *10.1016/j.entcs.2009.07.003*

[S124] Bak, S., Chivukula, D. K., Adekunle, O., Sun, M., Caccamo, M. and Sha, L.: 2009, The system-level simplex architecture for improved real-time embedded system safety, *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '09)*, IEEE, pp. 99–107. **DOI:** *10.1109/rtas.2009.20*

[S125] Barranco, M., Proenza, J. and Almeida, L.: 2009, Boosting the robustness of controller area networks: CANcentrate and ReCANcentrate, *Computer* **42**(5), 66–73. **DOI:** *10.1109/mc.2009.145*

[S126] Bochot, T., Virelizier, P., Waeselynck, H. and Wiels, V.: 2009, Model checking flight control systems: The Airbus experience., *Proceedings of the 31st International Conference on Software Engineering - Companion Volume (ICSE-Companion '09)*, Vol. 2009, pp. 18–27. **DOI:** *10.1109/icse-companion.2009.5070960*

[S127] Borde, E., Haïk, G. and Pautet, L.: 2009, Mode-based reconfiguration of critical software component architectures, *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '09)*, IEEE, pp. 1160–1165. **DOI:** *10.1109/date.2009.5090838*

[S128] Domis, D. and Trapp, M.: 2009, Component-based abstraction in fault tree analysis, *Computer Safety, Reliability, and Security*, Springer, pp. 297–310. **DOI:** *10.1007/978-3-642-04468-7_24*

[S129] Gustafsson, J., Altenbernd, P., Ermedahl, A. and Lisper, B.: 2009, Approximate worst-case execution time analysis for early stage embedded systems development, *Software Technologies for Embedded and Ubiquitous Systems*, Springer, pp. 308–319. **DOI:** *10.1007/978-3-642-10265-3_28*

[S130] Huber, B. and Obermaisser, R.: 2009, Platform Modeling in Safety-Critical Embedded Systems, *Intelligent Technical Systems*, Springer, pp. 145–158. **DOI:** *10.1007/978-1-4020-9823-9_11*

[S131] Islam, S., Suri, N., Balogh, A., Csertán, G. and Pataricza, A.: 2009, An optimization based design for integrated dependable real-time embedded systems, *Design Automation for Embedded Systems* **13**(4), 245–285. **DOI:** *10.1007/s10617-009-9041-7*

[S132] Izosimov, V., Polian, I., Pop, P., Eles, P. and Peng, Z.: 2009, Analysis and optimization of fault-tolerant embedded systems with hardened processors, *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '09)*, IEEE, pp. 682–687. **DOI:** *10.1109/DATE.2009.5090752*

[S133] Kim, J. E., Rogalla, O., Kramer, S. and Hamann, A.: 2009, Extracting, specifying and predicting software system properties in component based real-time embedded software development, *Proceedings of the 31st International Conference on Software Engineering - Companion Volume (ICSE-Companion '09)*, IEEE, pp. 28–38. **DOI:** *10.1109/ICSE-COMPANION.2009.5070961*

[S134] Lasnier, G., Zalila, B., Pautet, L. and Hugues, J.: 2009, Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications, *Proceedings of the 14th International Conference on Reliable Software Technologies (Ada-Europe '09)*, Springer, pp. 237–250. **DOI:** *10.1007/978-3-642-01924-1_17*

[S135] Medikonda, B. S. and Panchumarthy, S. R.: 2009b, An approach to modeling software safety in safety-critical systems, *Journal of Computer Science* **5**(4), 311. **DOI:** *10.3844/jcs.2009.311.322*

[S136] Medikonda, B. S. and Panchumarthy, S. R.: 2009a, A framework for software safety in safety-critical systems, *ACM SIGSOFT Software Engineering Notes* **34**(2), 1–9. **DOI:** *10.1145/1507195.1507207*

[S137] Miller, S. P., Cofer, D. D., Sha, L., Meseguer, J. and Al-Nayeem, A.: 2009, Implementing logical synchrony in integrated modular avionics, *Proceedings of the IEEE/AIAA 28th Digital Avionics Systems Conference (DASC '09)*, IEEE, pp. 1–A. **DOI:** *10.1109/dasc.2009.5347579*

[S138] Nanda, M. and Rao, S.: 2009, A formal method approach to analyze the design of aircraft flight control systems, *Proceedings of the Third Annual IEEE Systems Conference (SysCon '09)*, IEEE, pp. 64–69. **DOI:** *10.1109/systems.2009.4815773*

[S139] Pagano, B., Andrieu, O., Moniot, T., Canou, B., Chailloux, E., Wang, P., Manoury, P. and Colaço, J.-L.: 2009, Experience report: using Objective Caml to develop safety-critical embedded tools in a certification framework, *ACM SIGPLAN Notices* **44**(9), 215–220. **DOI:** *10.1145/1631687.1596582*

[S140] Pellizzoni, R., Meredith, P., Nam, M.-Y., Sun, M., Caccamo, M. and Sha, L.: 2009, Handling mixed-criticality in SoC-based real-time embedded systems, *Proceedings of the Seventh ACM International Conference on Embedded Software (EMSOFT '09)*, ACM, pp. 235–244. **DOI:** *10.1145/1629335.1629367*

[S141] Rosset, V., Souto, P. and Vasques, F.: 2009, Reliable communication for DuST networks, *Proceedings of the IEEE Conference on Emerging Technologies & Factory Automation (ETFA '09)*, IEEE, pp. 1–8. **DOI:** *10.1109/etfa.2009.5347122*

[S142] Selby, R. W.: 2009a, Development and Management of Large-Scale Mission-Critical Embedded Software Systems for Robotic Spacecraft, *Proceedings of the 47th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, p. 1648. **DOI:** *10.2514/6.2009-1648*

[S143] Selby, R. W.: 2009b, Synthesis, Analysis, and Modeling of Large-Scale Mission-Critical Embedded Software Systems, *Trustworthy Software Development Processes*, Springer, pp. 3–10. **DOI:** *10.1007/978-3-642-01680-6_3*

[S144] Sveda, M.: 2009, Fault Management for Secure Embedded Systems, *Proceedings of the Fourth International Conference on Systems (ICONS '09)*, IEEE, pp. 23–28. **DOI:** *10.1109/icons.2009.12*

[S145] Varona-Gomez, R. and Villar, E.: 2009, Aadl simulation and performance analysis in systemc, *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '09)*, IEEE, pp. 323–328. **DOI:** *10.1109/iceccs.2009.11*

[S146] Viehl, A., Pressler, M. and Bringmann, O.: 2009, Bottom-up performance analysis considering time slice based software scheduling at system level, *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES '09)*, ACM, pp. 423–432. **DOI:** *10.1145/1629435.1629493*

[S147] Wang, H. and Liang, N.: 2009, A software diversity model for embedded safety-critical system, *Proceedings of the International Conference on Wireless Networks and Information Systems (WNIS '09)*, IEEE, pp. 106–109. **DOI:** *10.1109/wnis.2009.52*

[S148] Yang, X., Lei, J. and Xiong, G.-z.: 2009, Inter-partition Information Flow Control for High-Assurance Embedded Systems, *Proceedings of the WRI World Congress on Computer Science and Information Engineering (CSIE '09)*, Vol. 2, IEEE, pp. 456–460. **DOI:** *10.1109/csie.2009.656*

[S149] Adler, R., Schaefer, I., Trapp, M. and Poetzsch-Heffter, A.: 2010, Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems, *ACM Transactions on Embedded Computing Systems* **10**(2), 20:1–20:39. **DOI:** *10.1145/1880050.1880056*

[S150] Adler, R., Schneider, D. and Trapp, M.: 2010, Engineering dynamic adaptation for achieving cost-efficient resilience in software-intensive embedded systems, *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '10)*, IEEE, pp. 21–30. **DOI:** *10.1109/iceccs.2010.22*

[S151] Aguiar, A., Sérgio Filho, J., Magalhães, F. G., Casagrande, T. D. and Hessel, F.: 2010, Hellfire: A design framework for critical embedded systems' applications, *Proceedings of the 11th International Symposium on Quality Electronic Design (ISQED '10)*, IEEE, pp. 730–737. **DOI:** *10.1109/isqed.2010.5450495*

[S152] Baruah, S., Li, H. and Stougie, L.: 2010, Towards the design of certifiable mixed-criticality systems, *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '10)*, IEEE, pp. 13–22. **DOI:** *10.1109/rtas.2010.10*

[S153] Correa, T., Becker, L. B., Farines, J.-M., Bodeveix, J.-P., Filali, M. and Vernadat, F.: 2010, Supporting the design of safety critical systems using AADL, *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '10)*, IEEE, pp. 331–336. **DOI:** *10.1109/iceccs.2010.56*

[S154] Farcas, C., Farcas, E., Krueger, I. H. and Menarini, M.: 2010, Addressing the Integration Challenge for Avionics and Automotive Systems—From Components to Rich Services, *Proceedings of the IEEE* **98**(4), 562–583. **DOI:** *10.1109/jproc.2009.2039630*

[S155] Feiler, P. H.: 2010, Model-based validation of safety-critical embedded systems, *Proceedings of the 2010 IEEE Aerospace Conference (AeroConf '10)*, IEEE, pp. 1–10. **DOI:** *10.1109/aero.2010.5446809*

[S156] Förster, M. and Schneider, D.: 2010, Flexible, any-time fault tree analysis with component logic models, *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10)*, IEEE, pp. 51–60. **DOI:** *10.1109/issre.2010.47*

[S157] Grießnig, G., Mader, R., Steger, C. and Weiß, R.: 2010, Design and implementation of safety functions on a novel CPLD-based fail-safe system architecture, *Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS '10)*, IEEE, pp. 206–212. **DOI:** *10.1109/ecbs.2010.29*

[S158] Insaurralde, C. C., Seminario, M. A., Jiménez, J. F. and Giron-Sierra, J. M.: 2010, Model-based development framework for distributed embedded control of aircraft fuel systems, *Proceedings of the IEEE/AIAA 29th Digital Avionics Systems Conference (DASC '10)*, IEEE, pp. 6–E. **DOI:** *10.1109/dasc.2010.5655449*

[S159] Lafaye, M., Faura, D., Gatti, M. and Pautet, L.: 2010, A new modeling approach for ima platform early validation, *Proceedings of the Seventh International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES '10)*, ACM, pp. 17–20. **DOI:** *10.1145/1865875.1865878*

[S160] Lakhani, F. and Pont, M. J.: 2010, Using design patterns to support migration between different system architectures, *Proceedings of the Fifth International Conference on System of Systems Engineering (SoSE '10)*, IEEE, pp. 1–6. **DOI:** *10.1109/sysose.2010.5544004*

[S161] Lefftz, V., Bertrand, J., Casse, H., Clienti, C., Coussy, P., Maillet-Contoz, L., Mercier, P., Moreau, P., Pierre, L. and Vaumorin, E.: 2010, A Design Flow for Critical Embedded Systems, *Proceedings of the Fifth International Symposium on Industrial Embedded Systems (SIES '10)*, pp. 229–233. **DOI:** *10.1109/sies.2010.5551393*

[S162] Lesens, D.: 2010, Using static analysis in space: why doing so?, *Static Analysis*, Springer, pp. 51–70. **DOI:** *10.1007/978-3-642-15769-1_5*

[S163] Li, C., Zhou, X. and Dong, Y.: 2010, Formal behavior specification for AADL, *Proceedings of the Second International Conference on Industrial and Information Systems (IIS '10)*, Vol. 2, IEEE, pp. 110–113. **DOI:** *10.1109/INDUSIS.2010.5565667*

[S164] Li, H. and Baruah, S.: 2010a, An algorithm for scheduling certifiable mixed-criticality sporadic task systems, *Proceedings of the IEEE 31st Real-Time Systems*

Symposium (RTSS '10), IEEE, pp. 183–192. **DOI:** *10.1109/rtss.2010.18*

[S165] Li, H. and Baruah, S.: 2010b, Load-based schedulability analysis of certifiable mixed-criticality systems, *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT '10)*, ACM, pp. 99–108. **DOI:** *10.1145/1879021.1879035*

[S166] Meseguer, J. and Ölveczky, P. C.: 2010, Formalization and correctness of the PALS architectural pattern for distributed real-time systems, *Formal Methods and Software Engineering*, Springer, pp. 303–320. **DOI:** *10.1007/978-3-642-16901-4_21*

[S167] Mitzlaff, M., Lang, M., Kapitza, R. and Schröder-Preikschat, W.: 2010, A membership service for a distributed, embedded system based on a time-triggered flexray network, *Proceedings of the Ninth European Dependable Computing Conference (EDCC '10)*, IEEE, pp. 155–162. **DOI:** *10.1109/edcc.2010.27*

[S168] Perez, J., Azkarate-Askasua, M. and Perez, A.: 2010, Codesign and simulated fault injection of safety-critical embedded systems using SystemC, *Proceedings of the Ninth European Dependable Computing Conference (EDCC '10)*, IEEE, pp. 221–229. **DOI:** *10.1109/edcc.2010.34*

[S169] Schlickling, M. and Pister, M.: 2010, Semi-automatic derivation of timing models for WCET analysis, *ACM SIGPLAN Notices* **45**(4), 67–76. **DOI:** *10.1145/1755951.1755899*

[S170] Stallbaum, H. and Rzepka, M.: 2010, Toward DO-178B-compliant Test Models, *Proceedings of the Seventh Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa '10)*, IEEE, pp. 25–30. **DOI:** *10.1109/modevva.2010.21*

[S171] Steindl, M., Mottok, J. and Meier, H.: 2010, SES-based Framework for Fault-tolerant Systems, *Proceedings of the Eight Workshop on Intelligent Solutions in Embedded Systems (WISES '10)*, IEEE, pp. 12–16. **DOI:** *10.1109/wises.2010.5548427*

[S172] Suri, N., Jhumka, A., Hiller, M., Pataricza, A., Islam, S. and Sârbu, C.: 2010, A software integration approach for designing and assessing dependable embedded systems, *Journal of Systems and Software* **83**(10), 1780–1800. **DOI:** *10.1016/j.jss.2010.04.063*

[S173] Sveda, M.: 2010, Fault Management Driven Design with Safety and Security Requirements, *Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS '10)*, IEEE, pp. 113–120. **DOI:** *10.1109/ecbs.2010.19*

[S174] Trienekens, J. J., Kusters, R. J. and Brussel, D. C.: 2010, Quality specification and metrication, results from a case-study in a mission-critical software domain, *Software Quality Journal* **18**(4), 469–490. **DOI:** *10.1007/s11219-010-9101-z*

[S175] Trindade, O., de Oliveira Neris, L., Barbosa, L. C. P. and Branco, K. R. L. J. C.: 2010, A layered approach to design autopilots, *Proceedings of the 2010 IEEE International Conference on Industrial Technology (ICIT '10)*, IEEE, pp. 1415–1420. **DOI:** *10.1109/ICIT.2010.5472499*

[S176] Varona-Gomez, R. and Villar, E.: 2010, Aads+: Aadl simulation including the behavioral annex, *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '10)*, IEEE, pp. 379–384. **DOI:** *10.1109/iceccs.2010.8*

[S177] Wasicek, A., El-Salloum, C. and Kopetz, H.: 2010, A system-on-a-chip platform for mixed-criticality applications, *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '10)*, IEEE, pp. 210–216. **DOI:** *10.1109/isorc.2010.43*

[S178] Yates, A. M., Torres-Pomales, W., Malekpour, M. R., González, O. R. and Gray, W. S.: 2010, High-Intensity Radiated Field fault-injection experiment for a fault-tolerant distributed communication system, *Proceedings of the 29th IEEE/AIAA Digital Avionics Systems Conference (DASC '10)*, IEEE, pp. 4–E. **DOI:** *10.1109/dasc.2010.5655331*

[S179] Yun, L. and Fulei, G.: 2010, Tool of scheduling simulation based on AADL models, *Proceedings of the Second World Congress on Software Engineering (WCSE '10)*, Vol. 1, IEEE, pp. 45–48. **DOI:** *10.1109/WCSE.2010.42*

[S180] Ölveczky, P. C., Boronat, A. and Meseguer, J.: 2010, Formal semantics and analysis of behavioral AADL models in Real-Time Maude, *Formal Techniques for Distributed Systems*, Springer, pp. 47–62. **DOI:** *10.1007/978-3-642-13464-7_5*

[S181] Abella, J., Cazorla, F. J., Quiñones, E., Grasset, A., Yehia, S., Bonnot, P., Gizopoulos, D., Mariani, R. and Bernat, G.: 2011, Towards improved survivability in safety-critical systems, *Proceedings of the IEEE 17th International On-Line Testing Symposium (IOLTS '11)*, IEEE, pp. 240–245. **DOI:** *10.1109/iolts.2011.5994536*

[S182] Assayad, I., Girault, A. and Kalla, H.: 2011, Tradeoff exploration between reliability, power consumption, and execution time, *Computer Safety, Reliability, and Security*, Springer, pp. 437–451. **DOI:** *10.1007/978-3-642-24270-0_32*

[S183] Belwal, C. and Cheng, A. M.: 2011, Feasibility interval for the transactional event handlers of P-FRP, *Proceedings of the IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '11)*, IEEE, pp. 966–973. **DOI:** *10.1109/trustcom.2011.133*

[S184] Björnander, S., Seceleanu, C., Lundqvist, K. and Pettersson, P.: 2011, Abv-a verifier for the architecture analysis and design language (aadl), *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '11)*, IEEE, pp. 355–360. **DOI:** *10.1109/iceccs.2011.43*

[S185] Blanquart, J.-P., Armengaud, E., Baufreton, P., Bourrouilh, Q., Griessnig, G., Krammer, M., Laurent, O., Machrouh, J., Peikenkamp, T., Schindler, C. et al.: 2011, Towards cross-domains model-based safety process, methods and tools for critical embedded systems: the CESAR approach, *Computer Safety, Reliability, and Security*, Springer, pp. 57–70. **DOI:** *10.1007/978-3-642-24270-0_5*

[S186] Bonifacio, G., Marmo, P., Orazzo, A., Petrone, I., Velardi, L. and Venticinque, A.: 2011, Improvement of processes and methods in testing activities for safety-critical embedded systems, *Computer Safety, Reliability, and Security*, Springer, pp. 369–382. **DOI:** *10.1007/978-3-642-24270-0_27*

[S187] Braga, R. T. V., Branco, K. R., Junior, O. T. and de Oliveira Neris, L.: 2011, Safe-crites: Developing safety-critical embedded systems supported by reuse techniques, *Proceedings of the IEEE International Conference on Information Reuse and Integration (IRI '11)*, IEEE, pp. 206–211. **DOI:** *10.1109/iri.2011.6009547*

[S188] Cardoso, J. M., Diniz, P. C., Petrov, Z., Bertels, K., Hübner, M., van Someren, H., Gonçalves, F., de Coutinho, J. G. F., Constantinides, G. A., Olivier, B. et al.: 2011, REFLECT: Rendering FPGAs to multi-core embedded computing, *Reconfigurable Computing*, Springer, pp. 261–289. **DOI:** *10.1007/978-1-4614-0061-5_11*

[S189] Cuenca-Asensi, S., Martínez-Álvarez, A., Restrepo-Calle, F., Palomo, F. R., Guzmán-Miranda, H. and Aguirre, M. A.: 2011, Soft core based embedded systems in critical aerospace applications, *Journal of Systems Architecture* **57**(10), 886–895. **DOI:** *10.1016/j.sysarc.2011.04.006*

[S190] Daramola, O., Stålhane, T., Sindre, G. and Omoronyia, I.: 2011, Enabling hazard identification from requirements and reuse-oriented HAZOP analysis, *Proceedings of the Fourth International Workshop on Managing Requirements Knowledge (MARK '11)*, IEEE, pp. 3–11. **DOI:** *10.1109/mark.2011.6046555*

[S191] Dias, D. M. and Iyoda, J. M.: 2011, Behavioural preservation in fault tolerant patterns, *Formal Methods, Foundations and Applications*, Springer, pp. 156–171.

DOI: *10.1007/978-3-642-25032-3_11*

[S192] El Ariss, O., Xu, D. and Wong, W. E.: 2011, Integrating safety analysis with functional modeling, *IEEE Transactions on Systems, Man and Cybernetics* **41**(4), 610–624. **DOI:** *10.1109/tsmca.2010.2093889*

[S193] Forget, J., Grolleau, E., Pagetti, C. and Richard, P.: 2011, Dynamic priority scheduling of periodic tasks with extended precedences, *Proceedings of the IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA '11)*, IEEE, pp. 1–8. **DOI:** *10.1109/etfa.2011.6059015*

[S194] Gray, I., Matragkas, N., Audsley, N. C., Indrusiak, L. S., Kolovos, D. and Paige, R.: 2011, Model-based hardware generation and programming-the MADES approach, *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW '11)*, IEEE, pp. 88–96. **DOI:** *10.1109/isorcw.2011.20*

[S195] Griessnig, G., Kundner, I., Armengaud, E., Torchiaro, S. and Karlsson, D.: 2011, Improving automotive embedded systems engineering at European level, *e & i Elektrotechnik und Informationstechnik* **128**(6), 209–214. **DOI:** *10.1007/s00502-011-0003-y*

[S196] Hilbrich, R. and Goltz, H.-J.: 2011, Model-based generation of static schedules for safety critical multi-core systems in the avionics domain, *Proceedings of the Fourth International Workshop on Multicore Software Engineering (IWMSE '11)*, ACM, pp. 9–16. **DOI:** *10.1145/1984693.1984695*

[S197] Hong, D., Gu, T. and Baik, J.: 2011, A uml model based white box reliability prediction to identify unreliable components, *Proceedings of the Fifth International Conference on Secure Software Integration & Reliability Improvement Companion (SSIRI-C '11)*, IEEE, pp. 152–159. **DOI:** *10.1109/ssiri-c.2011.30*

[S198] Hooman, J., Huis, R., Schuts, M. et al.: 2011, Experiences with a compositional model checker in the healthcare domain, *Foundations of Health Informatics Engineering and Systems*, Springer, pp. 93–110. **DOI:** *10.1007/978-3-642-32355-3_6*

[S199] Höfig, K. and Domis, D.: 2011, Failure-Dependent execution time analysis, *Proceedings of the joint ACM SIGSOFT Conference on Quality of software architectures and Architecting Critical Systems (QoSA-ISARCS '11)*, ACM, pp. 115–122. **DOI:** *10.1145/2000259.2000279*

[S200] Julien Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A. and Rival, X.: 2011, Static analysis by abstract interpretation of embedded critical software, *ACM SIGSOFT Software Engineering Notes* **36**(1), 1–8. **DOI:** *10.1145/1921532.1921553*

[S201] Kumar, S. P., Ramaiah, P. S. and Khanaa, V.: 2011, Architectural patterns to design software safety based safety-critical systems, *Proceedings of the 2011 International Conference on Communication, Computing & Security (ICCCS '11)*, ACM, pp. 620–623. **DOI:** *10.1145/1947940.1948069*

[S202] Lamy, F. and Schoofs, T.: 2011, Industry use cases for the Java environment for parallel realtime development, *Proceedings of the Ninth International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '11)*, ACM, pp. 106–115. **DOI:** *10.1145/2043910.2043928*

[S203] Leitner, A., Mader, R., Kreiner, C., Steger, C. and Weiß, R.: 2011, A development methodology for variant-rich automotive software architectures, *e & i Elektrotechnik und Informationstechnik* **128**(6), 222–227. **DOI:** *10.1007/s00502-011-0001-0*

[S204] Lévêque, T. and Sentilles, S.: 2011, Refining extra-functional property values in hierarchical component models, *Proceedings of the 14th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE '11)*, ACM, pp. 83–92. **DOI:** *10.1145/2000229.2000242*

[S205] Mader, R., Grießnig, G., Leitner, A., Kreiner, C., Bourrouilh, Q., Armengaud, E., Steger, C. and Weiss, R.: 2011, A Computer-Aided approach to preliminary hazard analysis for automotive embedded systems, *Proceedings of the 18th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS '11)*, IEEE, pp. 169–178. **DOI:** *10.1109/ecbs.2011.43*

[S206] Pagetti, C., Forget, J., Boniol, F., Cordovilla, M. and Lesens, D.: 2011, Multi-task implementation of multi-periodic synchronous programs, *Discrete Event Dynamic Systems* **21**(3), 307–338. **DOI:** *10.1007/s10626-011-0107-x*

[S207] Pedroza, G., Apvrille, L. and Knorreck, D.: 2011, Avatar: A sysml environment for the formal verification of safety and security properties, *Proceedings of the 11th Annual International Conference on New Technologies of Distributed Systems (NOTERE '11)*, IEEE, pp. 1–10. **DOI:** *10.1109/notere.2011.5957992*

[S208] Rodrigues, D., de Melo Pires, R., Estrella, J. C., Marconato, E. A., Trindade, O. and Branco, K. R. L. J. C.: 2011, Using SOA in Critical-Embedded Systems, *Proceedings of the joint 2011 International Conference on Internet of Things and Fourth International Conference on Cyber, Physical and Social Computing (iThings-CPSCom '11)*, IEEE, pp. 733–738. **DOI:** *10.1109/ithings/cpscom.2011.127*

[S209] Rodrigues, D., de Melo Pires, R., Estrella, J. C., Vieira, M., Corrêa, M., Júnior, J. B. C., Branco, K. R. L. J. C. and Júnior, O. T.: 2011, Application of SOA in safety-critical embedded systems, *Convergence and Hybrid Information Technology*, Springer, pp. 345–354. **DOI:** *10.1007/978-3-642-24106-2_45*

[S210] Sabetzadeh, M., Nejati, S., Briand, L. and Mills, A.-H. E.: 2011, Using SysML for modeling of safety-critical software-hardware interfaces: Guidelines and industry experience, *Proceedings of the IEEE 13th International Symposium on High-Assurance Systems Engineering (HASE '11)*, IEEE, pp. 193–201. **DOI:** *10.1109/hase.2011.23*

[S211] Saddem, R., Toguyeni, A. and Tagina, M.: 2011, Diagnosis of critical embedded systems: application to the control card of a railway vehicle braking systems, *Proceedings of the 2011 IEEE Conference on Automation Science and Engineering (CASE '11)*, IEEE, pp. 163–168. **DOI:** *10.1109/case.2011.6042512*

[S212] Sojer, D.: 2011, Synthesis of Fault Detection Mechanisms TRACK: Real-Time, Embedded and Physical Systems, *Proceedings of the IEEE 35th Annual Computer Software and Applications Conference (COMPSAC '11)*, IEEE, pp. 700–703. **DOI:** *10.1109/compsac.2011.108*

[S213] Varet, A. and Larrieu, N.: 2011, New methodology to develop certified safe and secure aeronautical software—An embedded router case study, *Proceedings of the IEEE/AIAA 30th Digital Avionics Systems Conference (DASC '11)*, IEEE, pp. 7C6–1. **DOI:** *10.1109/dasc.2011.6096126*

[S214] Wang, Y., Ma, D., Zhao, Y., Zou, L. and Zhao, X.: 2011, An aadl-based modeling method for arinc653-based avionics software, *Proceedings of the IEEE 35th Annual Computer Software and Applications Conference (COMPSAC '11)*, IEEE, pp. 224–229. **DOI:** *10.1109/compsac.2011.36*

[S215] Wasicek, A., El-Salloum, C. and Kopetz, H.: 2011, Authentication in time-triggered systems using time-delayed release of keys, *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '11)*, IEEE, pp. 31–39. **DOI:** *10.1109/isorc.2011.14*

[S216] Yu, H., Ma, Y., Glouche, Y., Talpin, J.-P., Besnard, L., Gautier, T., Guernic, P. L., Toom, A. and Laurent, O.: 2011, System-level co-simulation of integrated avionics using Polychrony, *Proceedings of the 26th ACM International Symposium on Applied Computing (SAC '11')*, ACM, pp. 354–359. **DOI:** *10.1145/1982185.1982263*

[S217] Zhang, Y., Li, G. and Zhang, J.: 2011, QP based framework for development and formal verification of flight control software of UAV, *Artificial Intelligence and Computational Intelligence*, Springer, pp. 1–8. **DOI:** *10.1007/978-3-642-23881-9_1*

[S218] Ziemke, C., Kuwahara, T. and Kossev, I.: 2011, An integrated development framework for rapid development of platform-independent and reusable satellite on-board software, *Acta Astronautica* **69**(7), 583–594. **DOI:** *10.1016/j.actaastro.2011.04.011*

[S219] Acharyulu, P. S. and Seetharamaiah, P.: 2012, A methodological framework for software safety in safety critical computer systems, *Journal of Computer Science* **8**(9), 1564. **DOI:** *10.3844/jcssp.2012.1564.1575*

[S220] Agrou, H., Sainrat, P., Gatti, M. and Toillon, P.: 2012, Mastering the behavior of multi-core systems to match avionics requirements, *Proceedings of the IEEE/AIAA 31st Digital Avionics Systems Conference (DASC '12)*, IEEE, pp. 6E5–1. **DOI:** *10.1109/dasc.2012.6382403*

[S221] Aliouat, Z. and Aliouat, M.: 2012, Verification of cooperative transient fault diagnosis and recovery in critical embedded systems, *The International Arab Journal of Information Technology* **9**(4), 373–381.

[S222] Andrade, H. A., Ghosal, A., Ravindran, K. and Evans, B. L.: 2012, A methodology for the design and deployment of reliable systems on heterogeneous platforms, *Proceedings of the 2012 International Conference on Reconfigurable Computing and FPGAs (ReConFig '12)*, IEEE, pp. 1–7. **DOI:** *10.1109/reconfig.2012.6416722*

[S223] Asplund, F., Biehl, M. and Loiret, F.: 2012, Towards the automated qualification of tool chain design, *Computer Safety, Reliability, and Security*, Springer, pp. 392–399. **DOI:** *10.1007/978-3-642-33675-1_36*

[S224] Barnat, J., Brim, L., Beran, J., Kratochvíla, T. and Oliveira, Í. R.: 2012, Executing model checking counterexamples in Simulink, *Proceedings of the Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE '12)*, IEEE, pp. 245–248. **DOI:** *10.1109/TASE.2012.42*

[S225] Baruah, S., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., Megow, N. and Stougie, L.: 2012, Scheduling real-time mixed-criticality jobs, *IEEE Transactions on Computers* **61**(8), 1140–1152. **DOI:** *10.1109/tc.2011.142*

[S226] Braga, R. T. V., Junior, O. T., Branco, K. R. C., Neris, L. D. O. and Lee, J.: 2012, Adapting a software product line engineering process for certifying safety critical embedded systems, *Computer Safety, Reliability, and Security*, Springer, pp. 352–363. **DOI:** *10.1007/978-3-642-33678-2_30*

[S227] Braga, R. T., Trindade Jr, O., Branco, K. R. and Lee, J.: 2012, Incorporating certification in feature modelling of an unmanned aerial vehicle product line, *Proceedings of the 16th International Software Product Line Conference (SPLC '12)*, ACM, pp. 249–258. **DOI:** *10.1145/2362536.2362570*

[S228] Cadoret, F., Borde, E., Gardoll, S. and Pautet, L.: 2012, Design patterns for rule-based refinement of safety critical embedded systems models, *Proceedings of the 17th International Conference on Engineering of Complex Computer Systems (ICECCS '12)*, IEEE, pp. 67–76.

[S229] Casola, V., Esposito, M., Mazzocca, N. and Flammini, F.: 2012, Freight train monitoring: A case-study for the pSHIELD project, *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, IEEE, pp. 597–602. **DOI:** *10.1109/imis.2012.51*

[S230] Costa, I. C. and de Oliveira, J. M. P.: 2012, Modeling Unmanned Aircraft System conflicts resolution based on a real-time services approach, *Proceedings of the IEEE/AIAA 31st Digital Avionics Systems Conference (DASC '12)*, IEEE, pp. 8A5–1. **DOI:** *10.1109/dasc.2012.6383115*

[S231] Dalpez, S., Vaccari, A., Passerone, R. and Penasa, A.: 2012, Design of an innovative proximity detection embedded-system for safety application in industrial machinery, *Proceedings of the IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA '12)*, IEEE, pp. 1–8. **DOI:** *10.1109/etfa.2012.6489582*

[S232] Diemer, J., Thiele, D. and Ernst, R.: 2012, Formal worst-case timing analysis of Ethernet topologies with strict-priority and AVB switching, *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES '12)*, IEEE, pp. 1–10. **DOI:** *10.1109/sies.2012.6356564*

[S233] Ebnenasir, A., Hajisheykhi, R. and Kulkarni, S. S.: 2012, Facilitating the design of fault tolerance in transaction level SystemC programs, *Distributed Computing and Networking*, Springer, pp. 91–105. **DOI:** *10.1016/j.tcs.2012.11.010*

[S234] Edmunds, A., Rezazadeh, A. and Butler, M.: 2012, Formal modelling for Ada implementations: Tasking event-B, *Proceedings of the 17th International Conference on Reliable Software Technologies (Ada-Europe '12)*, Springer, pp. 119–132. **DOI:** *10.1007/978-3-642-30598-6_9*

[S235] Fernandes, L. C., Souza, J. R., Shinzato, P. Y., Pessin, G., Mendes, C. C., Osório, F. S. and Wolf, D. F.: 2012, Intelligent robotic car for autonomous navigation: Platform and system architecture, *Proceedings of the Second Brazilian Conference on Critical Embedded Systems (CBSEC '12)*, IEEE, pp. 12–17. **DOI:** *10.1109/cbsec.2012.26*

[S236] Gatti, S., Aimé, F., Treuchot, S. and Jourdan, J.: 2012, Incremental functional certification for avionic functions reuse & evolution, *Proceedings of the IEEE/AIAA 31st Digital Avionics Systems Conference (DASC '12)*, IEEE, pp. 7A5–1. **DOI:** *10.1109/dasc.2012.6382409*

[S237] Gezgin, T., Henkler, S., Rettberg, A. and Stierand, I.: 2012, Abstraction techniques for compositional state-based scheduling analysis, *Proceedings of the Second Brazilian Symposium on Computing System Engineering (SBESC '12)*, IEEE, pp. 166–171. **DOI:** *10.1109/sbesc.2012.40*

[S238] Hazra, A., Ghosh, P. and Dasgupta, P.: 2012, Reliability annotations to formal specifications of context-sensitive safety properties in embedded systems, *Proceedings of the Forum on Specification and Design Languages (FDL '12)*, IEEE, pp. 36–43.

[S239] Jo, H.-C., Han, S., Lee, S.-H. and Jin, H.-W.: 2012, Implementing control and mission software of UAV by exploiting open source software-based arinc 653, *Proceedings of the 31st IEEE/AIAA Digital Avionics Systems Conference (DASC '12)*, IEEE, pp. 8B2–1. **DOI:** *10.1109/DASC.2012.6382436*

[S240] Koskinen, J., Vuori, M. and Katara, M.: 2012, Safety Process Patterns: Demystifying Safety Standards, *Proceedings of the IEEE International Conference on Software Science, Technology and Engineering (SWSTE '12)*, IEEE, pp. 63–71. **DOI:** *10.1109/swste.2012.10*

[S241] Lafaye, M., Pautet, L., Borde, E., Gatti, M. and Faura, D.: 2012, Model driven resource usage simulation for critical embedded systems, *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '12)*, EDA Consortium, pp. 312–315. **DOI:** *10.1109/date.2012.6176486*

[S242] Lee, S.-H., Han, S. and Jin, H.-W.: 2012, A Configurable, Extensible Implementation of Inter-Partition Communication for Integrated Modular Avionics, *Proceedings of the IEEE 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '12)*, IEEE, pp. 453–458. **DOI:** *10.1109/rtcsa.2012.44*

[S243] Lefftz, V. and Lachaize, J.: 2012, SoCKET: A HW/SW Co-Design Flow: Presentation & feedbacks from aeronautic and space application domains, *Proceedings of the Eurospace Data Systems in Aerospace Conference (DASIA '12)*.

[S244] Mader, R., Armengaud, E., Leitner, A. and Steger, C.: 2012, Automatic and optimal allocation of safety integrity levels, *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS '12)*, IEEE, pp. 1–6. **DOI:** *10.1109/rams.2012.6175431*

[S245] Mader, R., Grießnig, G., Armengaud, E., Leitner, A., Kreiner, C., Bourrouilh, Q., Steger, C. and Weiss, R.: 2012, A bridge from system to software development for safety-critical automotive embedded systems, *Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA '12)*, IEEE, pp. 75–79. **DOI:** *10.1109/seaa.2012.61*

[S246] Marrone, S., Nardone, R., Orazzo, A., Petrone, I. and Velardi, L.: 2012, Improving verification process in driverless metro systems: the MBAT project, *Proceedings of the Fifth International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '12)*, Springer, pp. 231–245. **DOI:** *10.1007/978-3-642-34032-1_23*

[S247] Méry, D. and Singh, N. K.: 2012, Critical systems development methodology using formal techniques, *Proceedings of the Third Symposium on Information and Communication Technology (SoICT '12)*, ACM, pp. 3–12. **DOI:** *10.1145/2350716.2350720*

[S248] Nejati, S., Di Alesio, S., Sabetzadeh, M. and Briand, L.: 2012, Modeling and analysis of CPU usage in safety-critical embedded systems to support stress testing, *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS '12)*, Springer. **DOI:** *10.1007/978-3-642-33666-9_48*

[S249] Novak, T. and Stoegerer, C.: 2012, Software architecture of a safety-related actuator in traffic management systems, *Computer Safety, Reliability, and Security*, Springer, pp. 268–278. **DOI:** *10.1007/978-3-642-33678-2_23*

[S250] Perez, J., Nicolas, C. F., Obermaisser, R. and El Salloum, C.: 2012, Modeling Time-Triggered Architecture Based Real-Time Systems Using SystemC, *System Specification and Design Languages*, Springer, pp. 123–141. **DOI:** *10.1007/978-1-4614-1427-8_8*

[S251] Saadatmand, M. and Leveque, T.: 2012, Modeling security aspects in distributed real-time component-based embedded systems, *Proceedings of the Ninth International Conference on Information Technology: New Generations (ITNG '12)*, IEEE, pp. 437–444. **DOI:** *10.1109/itng.2012.103*

[S252] Skopik, F., Treytl, A., Geven, A., Hirschler, B., Bleier, T., Eckel, A., El-Salloum,

C. and Wasicek, A.: 2012, Towards secure time-triggered systems, *Computer Safety, Reliability, and Security*, Springer, pp. 365–372. **DOI:** *10.1007/978-3-642-33675-1_33*

[S253] Soderberg, A. and Vedder, B.: 2012, Composable safety-critical systems based on pre-certified software components, *Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW '12)*, IEEE, pp. 343–348. **DOI:** *10.1109/issrew.2012.83*

[S254] Xu, T., Liu, Z., Tang, T., Zheng, W. and Zhao, L.: 2012, Component based design of fault tolerant devices in cyber physical system, *Proceedings of the 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW '12)*, IEEE, pp. 37–42. **DOI:** *10.1109/isorcw.2012.17*

[S255] Assayad, I., Girault, A. and Kalla, H.: 2013, Tradeoff exploration between reliability, power consumption, and execution time for embedded systems, *International Journal on Software Tools for Technology Transfer* **15**(3), 229–245. **DOI:** *10.1007/s10009-012-0263-9*

[S256] Ben Atitallah, R., Senn, E., Chillet, D., Lanoe, M. and Blouin, D.: 2013, An efficient framework for power-aware design of heterogeneous MPSoC, *IEEE Transactions on Industrial Informatics* **9**(1), 487–501. **DOI:** *10.1109/tii.2012.2198657*

[S257] Bolchini, C. and Miele, A.: 2013, Reliability-driven system-level synthesis for mixed-critical embedded systems, *IEEE Transactions on Computers* **62**(12), 2489–2502. **DOI:** *10.1109/tc.2012.226*

[S258] Boniol, F., Lauer, M., Pagetti, C. and Ermont, J.: 2013, Freshness and Reactivity Analysis in Globally Asynchronous Locally Time-Triggered Systems, *NASA Formal Methods*, Springer, pp. 93–107. **DOI:** *10.1007/978-3-642-38088-4_7*

[S259] Castellanos, C., Vergnaud, T., Borde, E., Derive, T. and Pautet, L.: 2013, Formalization of design patterns for security anddependability, *Proceedings of the 4th International ACM SIGSOFT Symposium on Architecting Critical Systems (ISARCS '13)*, ACM, pp. 17–26. **DOI:** *10.1145/2465470.2465476*

[S260] Hu, W., Oberg, J., Barrientos, J., Mu, D. and Kastner, R.: 2013, Expanding Gate Level Information Flow Tracking for Multilevel Security, *IEEE Embedded Systems Letters* **5**(2), 25–28. **DOI:** *10.1109/LES.2013.2261572*

[S261] Jiang, K., Eles, P. and Peng, Z.: 2013, Optimization of secure embedded systems with dynamic task sets, *Proceedings of the 2013 Design, Automation &*

Test in Europe Conference & Exhibition (DATE '13), IEEE, pp. 1765–1770. **DOI:** *10.7873/date.2013.355*

[S262] Martin, L. K., Schatalov, M., Hagner, M., Goltz, U. and Maibaum, O.: 2013, A methodology for model-based development and automated verification of software for aerospace systems, *Proceedings of the 2013 IEEE Aerospace Conference (AeroConf '13)*, IEEE, pp. 1–19. **DOI:** *10.1109/aero.2013.6496950*

[S263] Min, H.-S., Chung, S.-M. and Choi, J.-Y.: 2013, Deriving System Behavior from UML State Machine Diagram: Applied to Missile Project., *Journal of Universal Computer Science* **19**(1), 53–77. **DOI:** *10.3217/jucs-019-01-0053*

[S264] Notander, J. P., Runeson, P. and Höst, M.: 2013, A model-based framework for flexible safety-critical software development: a design study, *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*, ACM, pp. 1137–1144. **DOI:** *10.1145/2480362.2480575*

[S265] Osaiweran, A., Schuts, M., Hooman, J. and Wesselius, J.: 2013, Incorporating formal techniques into industrial practice: an experience report, *Electronic Notes in Theoretical Computer Science* **295**, 49–63. **DOI:** *10.1016/j.entcs.2013.04.005*

[S266] Rossignol, A.: 2013, *The Reference Technology Platform*, Springer, pp. 213–236. **DOI:** *10.1007/978-3-7091-1387-5_6*

[S267] Tamas-Selicean, D., Keymeulen, D., Berisford, D., Carlson, R., Hand, K., Pop, P., Wadsworth, W. and Levy, R.: 2013, Fourier transform spectrometer controller for partitioned architectures, *Proceedings of the 2013 IEEE Aerospace Conference (AeroConf '13)*, IEEE, pp. 1–11. **DOI:** *10.1109/aero.2013.6496969*

[S268] Wang, Y. and Ma, D.: 2013, An automatic development process for integrated modular avionics software, *Journal of Networks* **8**(5), 1088–1095. **DOI:** *10.4304/jnw.8.5.1088-1095*

[S269] Yoon, M.-K., Mohan, S., Choi, J., Kim, J.-E. and Sha, L.: 2013, SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems, *Proceedings of the IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS '13)*, IEEE, pp. 21–32. **DOI:** *10.1109/RTAS.2013.6531076*

Appendix B

B.1 Supplementary Tables to Chapter 4

Table B.2: Top three most recurrent violations per class type

QA	Class Type	Violation	Rank	Frequency
Security	SPP	Exposed inner representation by returning mutable object	RC4	37%
		Exposed inner representation by incorporating mutable object	RC4	32%
		Field should be final	RC4	18%
	CPP	Exposed inner representation by returning mutable object	RC4	38%
		Exposed inner representation by incorporating mutable object	RC4	25%
		Field should be final	RC4	13%
	NPP	Exposed inner representation by incorporating mutable object	RC4	35%
		Exposed inner representation by returning mutable object	RC4	31%
		Field should be final	RC4	20%
Correctness	SPP	Unsafe call for multithreading	RC2	14%
		Inconsistent synchronization	RC4	11%
		Incorrect initialization of static lazy field	RC4	10%
	CPP	Inconsistent synchronization	RC4	17%
		Nullcheck on dereferenced variable	RC3	9%
		Possible null pointer dereference	RC2	8%
	NPP	Nullcheck on dereferenced variable	RC3	10%
		Possible null pointer dereference	RC2	9%
		Possible null pointer dereference on exception	RC3	7%
Performance	SPP	Inner class should be static	RC4	27%
		Invoke of inefficient constructor	RC4	14%
		Class should be static	RC4	7%
	CPP	Invoke of inefficient constructor	RC4	23%
		Inner class should be static	RC4	22%
		Class should be static	RC4	6%
	NPP	Inner class should be static	RC4	23%
		Invoke of inefficient constructor	RC4	12%
		Inefficient conversion to array	RC4	10%

FindBugs' rank categories: RC1—Scariest; RC2—Scary; RC3—Troubling; RC4—Of Concern

Table B.3: Top three most recurrent violations per pattern category

QA	Pattern Category	Violation	Rank	Frequency
Security	Creational	Field should be final	RC4	41%
		Exposed inner representation by returning mutable object	RC4	22%
		Exposed inner representation by incorporating mutable object	RC4	16%
	Behavioral	Exposed inner representation by returning mutable object	RC4	36%
		Exposed inner representation by incorporating mutable object	RC4	29%
		Field should be final	RC4	19%
	Structural	Exposed inner representation by incorporating mutable object	RC4	57%
		Field should be final	RC4	24%
		Exposed inner representation by returning mutable object	RC4	14%
Correctness	Creational	Incorrect initialization of static lazy field	RC4	37%
		Unsafe call for multithreading	RC2	26%
		Nullcheck on dereferenced variable	RC3	11%
	Behavioral	Inconsistent synchronization	RC4	16%
		Possible null pointer dereference on exception	RC3	13%
		Unsafe call for multithreading	RC2	10%
	Structural	Possible null value is passed to a method that cannot handle it	RC2	60%
		Potential disposal of necessary return value	RC1	20%
		Field always return default value	RC3	20%
Performance	Creational	Inner class should be static	RC4	29%
		Invoke of inefficient constructor	RC4	13%
		Slow parsing of primitive value	RC4	9%
	Behavioral	Inner class should be static	RC4	24%
		Invoke of inefficient constructor	RC4	15%
		Class should be static	RC4	11%
	Structural	Inefficient conversion to array	RC4	56%
		Inefficient search for first occurrence	RC4	13%
		Inefficient search for last occurrence	RC4	13%

FindBugs' rank categories: RC1—Scariest; RC2—Scary; RC3—Troubling; RC4—Of Concern

Table B.4: Top three most recurrent violations per pattern

QA	Pattern	Violation	Rank	Frequency
Security	Factory Method	Exposed inner representation by returning mutable object	RC4	36%
		Exposed inner representation by incorporating mutable object	RC4	34%
		Field should be final	RC4	27%
	Prototype	Field should be final	RC4	71%
		Exposed inner representation by returning mutable object	RC4	18%
		Opportunity for SQL injection	RC4	6%
	Singleton	Field should be final	RC4	37%
		Exposed inner representation by returning mutable object	RC4	11%
		Field should be package protected	RC4	11%
	Adapter/Command	Exposed inner representation by returning mutable object	RC4	48%
		Exposed inner representation by incorporating mutable object	RC4	42%
		Field should be final	RC4	5%
	Decorator	Exposed inner representation by incorporating mutable object	RC4	55%
		Field should be final	RC4	25%
		Exposed inner representation by returning mutable object	RC4	15%
	State/Strategy	Exposed inner representation by returning mutable object	RC4	44%
		Exposed inner representation by incorporating mutable object	RC4	35%
		Field should be final	RC4	8%
	Template Method	Field should be final	RC4	34%
		Exposed inner representation by returning mutable object	RC4	26%
		Exposed inner representation by incorporating mutable object	RC4	22%
Correctness	Factory Method	Unsafe call for multithreading	RC2	70%
		Nullcheck on dereferenced variable	RC3	16%
		Lack of appropriate hashing method	RC1	5%
	Prototype	Inconsistent synchronization	RC4	17%
		Possible null pointer dereference	RC2	17%
		Repetition of conditional tests	RC3	17%
	Singleton	Incorrect initialization of static lazy field	RC4	71%
		Nullcheck on dereferenced variable	RC3	8%
		Check of instance type is always false	RC2	4%
	Adapter/Command	Inconsistent synchronization	RC4	14%
		Null value is passed to a method that cannot handle it	RC2	11%
		Unsafe call for multithreading	RC2	8%
	Decorator	Possible null value is passed to a method that cannot handle it	RC2	60%
		Potential disposal of necessary return value	RC1	20%
		Field always return default value	RC3	20%
	State/Strategy	Inconsistent synchronization	RC4	23%
		Possible null pointer dereference on exception	RC3	20%
		Inconsistent synchronization	RC4	8%
	Template Method	Unsafe call for multithreading	RC2	30%
		Nullcheck on dereferenced variable	RC3	10%
		Possible null pointer dereference	RC2	9%
Performance	Factory Method	Slow parsing of primitive value	RC4	19%
		Unnecessary value unboxing	RC4	18%
		Inefficient search for last occurrence	RC4	12%
	Prototype	Invoke of inefficient constructor	RC4	26%
		Inner class should be static	RC4	13%
		Inefficient concatenation of string in a loop	RC4	12%
	Singleton	Inner class should be static	RC4	58%
		Inefficient search for first occurrence	RC4	7%
		Invoke of inefficient constructor	RC4	6%
	Adapter/Command	Inner class should be static	RC4	36%
		Invoke of inefficient constructor	RC4	14%
		Inefficient search for first occurrence	RC4	10%
	Decorator	Inefficient conversion to array	RC4	56%
		Inefficient search for first occurrence	RC4	13%
		Inefficient search for last occurrence	RC4	13%
	State/Strategy	Inner class should be static	RC4	31%
		Class should be static	RC4	9%
		Invoke of inefficient constructor	RC4	8%
	Template Method	Inner class should be static	RC4	18%
		Class should be static	RC4	15%
		Invoke of inefficient constructor	RC4	14%

FindBugs' rank categories: RC1—Scariest; RC2—Scary; RC3—Troubling; RC4—Of Concern

Table B.5: Top three most recurrent violations per meta-role

QA	Meta-role	Violation	Rank	Frequency
Security	Client	Exposed inner representation by returning mutable object	RC4	41%
		Exposed inner representation by incorporating mutable object	RC4	34%
		Field should be final	RC4	13%
	Superclass	Field should be final	RC4	29%
		Exposed inner representation by returning mutable object	RC4	26%
		Exposed inner representation by incorporating mutable object	RC4	18%
	Subclass	Exposed inner representation by returning mutable object	RC4	35%
		Exposed inner representation by incorporating mutable object	RC4	25%
		Field should be final	RC4	24%
	Container	Exposed inner representation by incorporating mutable object	RC4	38%
		Exposed inner representation by returning mutable object	RC4	23%
		Opportunity for SQL injection	RC4	19%
	Containee	Exposed inner representation by returning mutable object	RC4	47%
		Exposed inner representation by incorporating mutable object	RC4	40%
		Field should be final	RC4	8%
	Both	Exposed inner representation by incorporating mutable object	RC4	63%
		Exposed inner representation by returning mutable object	RC4	25%
		Field should be final	RC4	13%
Correctness	Client	Inconsistent synchronization	RC4	21%
		Possible null pointer dereference on exception	RC3	9%
		Mistaken read of uninitialized field in constructor	RC1	9%
	Superclass	Inconsistent synchronization	RC4	62%
		Possible null pointer dereference	RC2	10%
		Possible race condition in servlet field	RC4	8%
	Subclass	Possible null pointer dereference on exception	RC3	17%
		Unsafe call for multithreading	RC2	15%
		Should notify all threads	RC4	7%
	Container	Unsafe call for multithreading	RC2	19%
		Suspicious use of inappropriate comparison	RC2	17%
		Potential dereferencing a field with null value	RC3	14%
	Containee	Inconsistent synchronization	RC4	23%
		Null value is passed to a method that cannot handle it	RC2	19%
		Inconsistent synchronization	RC4	9%
	Both	Unsafe call for multithreading	RC2	72%
		Nullcheck on dereferenced variable	RC3	14%
		Lack of appropriate hashing method	RC1	6%
Performance	Client	Inner class should be static	RC4	27%
		Class should be static	RC4	10%
		Private method is never called	RC4	10%
	Superclass	Invoke of inefficient constructor	RC4	35%
		Inefficient conversion to array	RC4	13%
		Class should be static	RC4	11%
	Subclass	Inner class should be static	RC4	21%
		Invoke of inefficient constructor	RC4	15%
		Class should be static	RC4	10%
	Container	Inner class should be static	RC4	36%
		Invoke of inefficient constructor	RC4	14%
		Private method is never called	RC4	12%
	Containee	Inner class should be static	RC4	30%
		Inefficient search for first occurrence	RC4	14%
		Invoke of inefficient constructor	RC4	13%
	Both	Slow parsing of primitive value	RC4	24%
		Unnecessary value unboxing	RC4	22%
		Field should be static	RC4	11%

FindBugs' rank categories: RC1—Scariest; RC2—Scary; RC3—Troubling; RC4—Of Concern

Table B.6: Average number of violations per version per KLOC

RQ	QA	Group	New				Removed				Total			
			RC1	RC2	RC3	RC4	RC1	RC2	RC3	RC4	RC1	RC2	RC3	RC4
Pattern Participation	Sec.	SPP	0.00	1.36	0.07	29.25	0.00	0.62	0.00	14.01	0.00	37.32	2.74	651.34
		CPP	0.00	0.00	0.00	12.66	0.00	0.00	0.00	6.82	0.00	0.00	0.00	276.64
		NPP	0.00	1.54	0.13	77.95	0.00	1.10	0.00	34.29	0.00	20.07	1.70	1957.39
	Cor.	SPP	1.18	4.04	2.81	1.52	0.77	1.18	0.84	0.98	20.38	90.31	63.44	39.70
		CPP	0.63	1.14	1.46	0.91	0.53	0.56	0.83	0.56	8.24	19.38	26.74	29.70
		NPP	2.37	11.02	9.63	1.31	1.77	5.86	4.53	1.04	45.12	167.35	173.37	31.71
	Per.	SPP	0.00	0.00	0.00	16.52	0.00	0.00	0.00	7.13	0.00	0.00	0.00	334.03
		CPP	0.00	0.00	0.00	9.74	0.00	0.00	0.00	5.37	0.00	0.00	0.00	221.75
		NPP	0.00	0.00	0.00	48.91	0.00	0.00	0.00	28.87	0.00	0.00	0.00	1058.03
Pattern Category	Sec.	Creational	0.00	0.00	0.00	5.37	0.00	0.00	0.00	2.06	0.00	0.00	0.00	111.50
		Behavioral	0.00	1.36	0.07	16.66	0.00	0.62	0.00	8.93	0.00	37.32	2.74	339.21
		Structural	0.00	0.00	0.00	1.41	0.00	0.00	0.00	0.65	0.00	0.00	0.00	23.59
	Cor.	Creational	0.10	0.60	0.22	0.17	0.09	0.11	0.07	0.05	1.12	12.75	3.47	5.02
		Behavioral	0.83	2.72	2.01	0.83	0.58	0.62	0.32	0.59	12.21	64.46	48.76	22.80
		Structural	0.01	0.06	0.13	0.00	0.01	0.06	0.13	0.00	0.14	0.18	5.18	0.00
	Per.	Creational	0.00	0.00	0.00	3.54	0.00	0.00	0.00	1.73	0.00	0.00	0.00	77.33
		Behavioral	0.00	0.00	0.00	11.38	0.00	0.00	0.00	4.69	0.00	0.00	0.00	224.75
		Structural	0.00	0.00	0.00	0.17	0.00	0.00	0.00	0.08	0.00	0.00	0.00	2.80
Pattern	Security	FactoryMethod	0.00	0.00	0.00	0.81	0.00	0.00	0.00	0.02	0.00	0.00	0.00	12.33
		Prototype	0.00	0.00	0.00	0.66	0.00	0.00	0.00	0.56	0.00	0.00	0.00	16.93
		Singleton	0.00	0.00	0.00	3.90	0.00	0.00	0.00	1.48	0.00	0.00	0.00	82.25
		AdapterCommand	0.00	0.00	0.00	5.81	0.00	0.00	0.00	2.36	0.00	0.00	0.00	177.04
		Decorator	0.00	0.00	0.00	1.34	0.00	0.00	0.00	0.65	0.00	0.00	0.00	23.02
		Observer	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		StateStrategy	0.00	0.00	0.00	8.41	0.00	0.00	0.00	4.71	0.00	0.00	0.00	183.84
		TemplateMethod	0.00	1.36	0.07	8.25	0.00	0.62	0.00	4.22	0.00	37.32	2.74	155.37
	Correctness	FactoryMethod	0.02	0.20	0.04	0.00	0.02	0.08	0.03	0.00	0.39	4.66	0.66	0.00
		Prototype	0.00	0.08	0.03	0.07	0.00	0.02	0.00	0.00	0.00	3.01	1.07	2.64
		Singleton	0.07	0.33	0.15	0.10	0.06	0.01	0.04	0.05	0.72	5.08	1.73	2.38
		AdapterCommand	0.24	0.66	0.45	0.52	0.09	0.39	0.32	0.33	6.91	12.93	6.04	11.88
		Decorator	0.01	0.06	0.13	0.00	0.01	0.06	0.13	0.00	0.14	0.18	5.18	0.00
		Observer	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.22	0.00	0.00
		StateStrategy	0.38	0.62	1.16	0.75	0.29	0.25	0.15	0.51	5.60	12.94	29.66	20.86
		TemplateMethod	0.45	2.05	0.85	0.08	0.28	0.37	0.17	0.08	6.62	49.29	19.10	1.94
	Performance	FactoryMethod	0.00	0.00	0.00	1.15	0.00	0.00	0.00	0.98	0.00	0.00	0.00	17.91
		Prototype	0.00	0.00	0.00	1.32	0.00	0.00	0.00	0.19	0.00	0.00	0.00	43.43
		Singleton	0.00	0.00	0.00	1.07	0.00	0.00	0.00	0.55	0.00	0.00	0.00	15.99
		AdapterCommand	0.00	0.00	0.00	1.43	0.00	0.00	0.00	0.63	0.00	0.00	0.00	29.15
		Decorator	0.00	0.00	0.00	0.17	0.00	0.00	0.00	0.08	0.00	0.00	0.00	2.80
		Observer	0.00	0.00	0.00	0.22	0.00	0.00	0.00	0.22	0.00	0.00	0.00	2.40
		StateStrategy	0.00	0.00	0.00	3.97	0.00	0.00	0.00	1.68	0.00	0.00	0.00	76.25
		TemplateMethod	0.00	0.00	0.00	7.19	0.00	0.00	0.00	2.79	0.00	0.00	0.00	146.10
Meta-role	Security	Client	0.00	0.00	0.00	3.06	0.00	0.00	0.00	2.05	0.00	0.00	0.00	54.75
		Superclass	0.00	0.11	0.00	0.58	0.00	0.11	0.00	0.26	0.00	0.11	0.00	9.61
		Subclass	0.00	1.24	0.07	14.50	0.00	0.51	0.00	7.70	0.00	37.21	2.74	306.73
		Container	0.00	0.00	0.00	0.35	0.00	0.00	0.00	0.25	0.00	0.00	0.00	9.84
		Containee	0.00	0.00	0.00	5.89	0.00	0.00	0.00	2.13	0.00	0.00	0.00	176.28
		Both	0.00	0.00	0.00	0.97	0.00	0.00	0.00	0.14	0.00	0.00	0.00	11.88
	Correctness	Client	0.32	0.34	0.68	0.70	0.28	0.12	0.10	0.47	3.86	6.24	11.42	19.82
		Superclass	0.10	0.05	0.30	0.00	0.10	0.05	0.00	0.00	0.21	0.14	13.38	0.00
		Subclass	0.41	2.41	1.19	0.20	0.20	0.52	0.34	0.12	8.28	59.05	30.20	5.62
		Container	0.09	0.35	0.45	0.00	0.06	0.28	0.32	0.00	1.59	4.02	6.04	0.00
		Containee	0.15	0.37	0.00	0.52	0.03	0.11	0.00	0.33	5.32	11.13	0.00	11.88
		Both	0.02	0.20	0.04	0.00	0.02	0.08	0.03	0.00	0.39	4.66	0.66	0.00
	Performance	Client	0.00	0.00	0.00	2.43	0.00	0.00	0.00	0.85	0.00	0.00	0.00	46.11
		Superclass	0.00	0.00	0.00	0.68	0.00	0.00	0.00	0.33	0.00	0.00	0.00	14.26
		Subclass	0.00	0.00	0.00	9.55	0.00	0.00	0.00	3.56	0.00	0.00	0.00	208.20
		Container	0.00	0.00	0.00	0.90	0.00	0.00	0.00	0.46	0.00	0.00	0.00	15.63
		Containee	0.00	0.00	0.00	0.85	0.00	0.00	0.00	0.39	0.00	0.00	0.00	20.30
		Both	0.00	0.00	0.00	1.04	0.00	0.00	0.00	0.98	0.00	0.00	0.00	13.54

FindBugs' rank categories: RC1—Scariest; RC2—Scary; RC3—Troubling; RC4—Of Concern

Bibliography

Adamczyk, P.: 2003, The anthology of the finite state machine design patterns, *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP '03)*, Irsee, Germany.

Adamczyk, P.: 2004, Selected patterns for implementing finite state machines, *Proceedings of the 11th Conference on Pattern Languages of Programs (PLoP '04)*, Irsee, Germany.

Afacan, T.: 2011, State design pattern implementation of a DSP processor: A case study of Tms5416C, *Proceedings of the Sixth International Symposium on Industrial and Embedded Systems (SIES '11)*, IEEE, Vasteras, Sweden, pp. 67–70.
DOI: *10.1109/SIES.2011.5953682*

Aguiar, A., Filho, S. J., Magalhães, F. G., Casagrande, T. D., Hessel, F., Magalhaes, F. G., Casagrande, T. D. and Hessel, F.: 2010, Hellfire: A design framework for critical embedded systems' applications, *Proceedings of the 11th International Symposium on Quality Electronic Design (ISQED '10)*, IEEE, San Jose, CA, USA, pp. 730–737.
DOI: *10.1109/ISQED.2010.5450495*

Ahlgren, R. and Markkula, J.: 2005, Design patterns and organisational memory in mobile application development, *Proceedings of the Sixth International Conference on Product Focused Software Process Improvement (PROFES '05)*, Oulu, Finland, pp. 143–156.
DOI: *10.1007/11497455_13*

AlBreiki, H. H. and Mahmoud, Q. H.: 2014, Evaluation of static analysis tools for software security, *Proceedings of the 10th International Conference on Innovations in Information Technology (IIT '14)*, IEEE, Al Ain, United Arab Emirates, pp. 93–98.
DOI: *10.1109/INNOVATIONS.2014.6987569*

Aleksy, M., Korthaus, A. and Seifried, C.: 2006, Design patterns usage in peer-to-peer systems – An empirical analysis, *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology Workshops (WI-IATW '06)*, IEEE Computer Society, Hong Kong, China, pp. 459–462.
DOI: *10.1109/WI-IATW.2006.57*

Alhusain, S., Coupland, S., John, R. and Kavanagh, M.: 2013, Towards machine learning based design pattern recognition, *Proceedings of the 13th UK Workshop on Computational Intelligence (UKCI '13)*, IEEE, Guildford, UK, pp. 244–251.
DOI: *10.1109/UKCI.2013.6651312*

Ali, M. and Elish, M. O.: 2013, A comparative literature survey of design patterns impact on software quality, *Proceedings of the Fourth International Conference on Information Science and Applications (ICISA '13)*, IEEE, Suwon, South Korea, pp. 1–7.
DOI: *10.1109/ICISA.2013.6579460*

Alshammari, B., Fidge, C. and Corney, D.: 2010, Security metrics for object-oriented designs, *Proceedings of the 21st Australian Software Engineering Conference (ASWEC '10)*, IEEE, Auckland, New Zealand, pp. 55–64.
DOI: *10.1109/ASWEC.2010.34*

Alves, V., Niu, N., Alves, C. and Valença, G.: 2010, Requirements engineering for software product lines: A systematic literature review, *Information and Software Technology* **52**(8), 806–820.
DOI: *10.1016/j.infsof.2010.03.014*

Amanatidis, T., Chatzigeorgiou, A., Ampatzoglou, A. and Stamelos, I.: 2017, Who is producing more technical debt?, *Proceedings of the Nineth International Workshop on Managing Technical Debt (MTD '17)*, ACM Press, Cologne, Germany, pp. 4:1–4:8.
DOI: *10.1145/3120459.3120464*

Ampatzoglou, A., Charalampidou, S. and Stamelos, I.: 2011, Investigating the use of object-oriented design patterns in open-source software: A case study, *in* L. A. Maciaszek and P. Loucopoulos (eds), *Evaluation of Novel Approaches to Software Engineering*, Springer Berlin Heidelberg, pp. 106–120.
DOI: *10.1007/978-3-642-23391-3_8*

Ampatzoglou, A., Charalampidou, S. and Stamelos, I.: 2013a, Design pattern alternatives, *Proceedings of the 17th Panhellenic Conference on Informatics (PCI '13)*, ACM, Thessaloniki, Greece, pp. 122–127.
DOI: *10.1145/2491845.2491857*

Ampatzoglou, A., Charalampidou, S. and Stamelos, I.: 2013b, Research state of the art on GoF design patterns: A mapping study, *Journal of Systems and Software* **86**(7), 1945–1964.
DOI: *10.1016/j.jss.2013.03.063*

Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S. and Avgeriou, P.: 2015, The effect of GoF design patterns on stability: A case study, *IEEE Transactions on Software Engineering* **41**(8), 781–802.
DOI: *10.1109/TSE.2015.2414917*

Ampatzoglou, A., Frantzeskou, G. and Stamelos, I.: 2012, A methodology to assess the impact of design patterns on software quality, *Information and Software Technology* **54**(4), 331–346.
DOI: *10.1016/j.infsof.2011.10.006*

Ampatzoglou, A., Gkortzis, A., Charalampidou, S. and Avgeriou, P.: 2013, An embedded multiple-case study on oss design quality assessment across domains, *Proceedings of the Seventh ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '13)*, IEEE, pp. 255–258.
DOI: *10.1109/ESEM.2013.48*

Ampatzoglou, A., Kritikos, A., Arvanitou, E.-M., Gortzis, A., Chatziasimidis, F. and Stamelos, I.: 2011, An empirical investigation on the impact of design pattern application on computer game defects, *Proceedings of the 15th International Academic MindTrek Conference on Envisioning Future Media Environments (MindTrek '11)*, ACM, Tampere, Finland, pp. 214–221.
DOI: *10.1145/2181037.2181074*

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), 131–142.
DOI: *10.1016/j.entcom.2012.10.002*

Antonio, E. A., Ferrari, F. C. and Ferraz Fabbri, S. C. P.: 2012, A systematic mapping of architectures for embedded software, *Proceedings of the Second Brazilian Conference on Critical Embedded Systems (CBSEC '12)*, Campinas, Brazil, pp. 18–23.
DOI: *10.1109/CBSEC.2012.22*

Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C. and Di Penta, M.: 2007, An empirical study on the evolution of design patterns, *Proceedings of the the Sixth joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07)*, ACM Press, Dubrovnik, Croatia, p. 385.
DOI: *10.1145/1287624.1287680*

Aversano, L., Cerulo, L. and Di Penta, M.: 2009, Relationship between design patterns defects and crosscutting concern scattering degree: an empirical study, *IET Software* **3**(5), 395.
DOI: *10.1049/iet-sen.2008.0105*

Aversano, L., Cerulo, L. and Penta, M. D.: 2007, Relating the evolution of design patterns and crosscutting concerns, *Proceedings of the Seventh International Working Conference on Source Code Analysis and Manipulation (SCAM '07)*, IEEE, Paris, France, pp. 180–192.
DOI: *10.1109/SCAM.2007.21*

Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J. and Pugh, W.: 2008, Using static analysis to find bugs, *IEEE Software* **25**(5), 22–29.
DOI: *10.1109/MS.2008.130*

Ayewah, N. and Pugh, W.: 2010, The Google FindBugs fixit, *Proceedings of the 19th international symposium on Software testing and analysis (ISSTA '10)*, ACM, Trento, Italy, pp. 241–252.
DOI: *10.1145/1831708.1831738*

Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J. and Zhou, Y.: 2007, Evaluating static analysis defect warnings on production software, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '07)*, ACM Press, San Diego, California, USA, pp. 1–8.
DOI: *10.1145/1251535.1251536*

Azadmanesh, M. R., Hauswirth, M. and Van De Vanter, M. L.: 2017, Language-independent information flow tracking engine for program comprehension tools, *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*, IEEE Press, pp. 346–355.
DOI: *10.1109/ICPC.2017.5*

Bafandeh Mayvan, B., Rasoolzadegan, A. and Ghavidel Yazdi, Z.: 2017, The state of the art on design patterns: A systematic mapping of the literature, *Journal of Systems and Software* **125**, 93–118.
DOI: *10.1016/j.jss.2016.11.030*

Balmas, F., Bergel, A., Denier, S., Ducasse, S., Laval, J., K., M.-M. and H. Abdeen, F. B.: 2010, SQualE - Software metric for Java and C++ practices, *Technical report*, INRIA, Paris.
URL: *http://www.squale.org/quality-models-site/research-deliverables/WP1.1_Software-metrics-for-Java-and-Cpp-practices_v2.pdf*

Bansiya, J. and Davis, C.: 2002, A hierarchical model for object-oriented design quality assessment, *IEEE Transactions on Software Engineering* **28**(1), 4–17.
DOI: *10.1109/32.979986*

Barbosa, J. R., Delamaro, M. E., Maldonado, J. C. and Vincenzi, A. M. R.: 2011, Software testing in critical embedded systems: a systematic review of adherence to the Do-178B standard, *Proceedings of the Third International Conference on Advances in System Testing and Validation Lifecycle*, Barcelona, Spain, pp. 126–130.
URL: *https://www.thinkmind.org/index.php?view=article&articleid=valid_2011_5_40_40074*

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), 651–662.
DOI: *10.1016/j.infsof.2012.01.008*

Barros, M. d. O., Farzat, F. d. A. and Travassos, G. H.: 2014, Learning from optimization: A case study with Apache Ant, *Information and Software Technology* **57**(1), 684–704.
DOI: *10.1016/j.infsof.2014.07.015*

Bartelt, C., Bauer, O., Beneken, G., Bergner, K., Birowicz, U., Bliß, T., Cordes, N., Cruz, D., Dohrmann, P., Friedrich, J., Gnatz, M., Hammerschall, U., Hidvegi-Barstorfer, I., Hummel, H., Israel, D., Klingenberg, T., Klugseder, K., Küffer, I., Kuhrmann, M., Kranz, M., Kranz, W., Meinhardt, H.-J., Meisinger, M., Mittrach, S., Neußer, H.-J., Niebuhr, D., Plögert, K., Rauh, D., Rausch, A., Rittel, T., Rösch, W., Saas, E., Schramm, J., Sihling, M., Ternité, T., Vogel, S. and Wittmann, M.: 2010, V-modell Xt Gesamt 1.3, *Technical report*, Bundesverwaltungsamt, Freistaat Bayern, 4Soft GmbH, Airbus Defence & Space AG/GmbH, IABG

mbH, Siemens AG, Technische Universität Clausthal, Technische Universität München.
URL: *http://v-modell.iabg.de/XThtmleng/index.html*

Bass, L., Clements, P. and Kazman, R.: 2012, *Software Architecture in Practice*, 3 edn, Addison-Wesley Professional.

Bass, L., Nord, R., Wood, W., Zubrow, D. and Ozkaya, I.: 2008, Analysis of architecture evaluation data, *Journal of Systems and Software* **81**(9), 1443–1455.
DOI: *10.1016/j.jss.2008.02.021*

Bate, I.: 2008, Systematic approaches to understanding and evaluating design trade-offs, *Journal of Systems and Software* **81**(8), 1253–1271.
DOI: *10.1016/j.jss.2007.10.032*

Boehm, B. and In, H.: 1996, Identifying quality-requirement conflicts, *IEEE Software* **13**(2), 25–35.
DOI: *10.1109/52.506460*

Bourque, P. and Fairley, R. E.: 2014, *Guide to the Software Engineering Body of Knowledge*, 3rd edn, IEEE Computer Society Press.

Brooks Jr., F. P.: 1987, No silver bullet essence and accidents of software engineering, *Computer* **20**(4), 10–19.
DOI: *10.1109/MC.1987.1663532*

Bunse, S. S. C., Schwedenschanze, Z. and Stiemer, S.: 2013, On the energy consumption of design patterns, *Proceedings of the Seconda Workshop EASED BUIS Energy Aware Software-Engineering and Development*, pp. 7–8.

Buschmann, F., Henney, K. and Schmidt, D. C.: 2007, *Pattern-Oriented Software Architecture, On Patterns and Pattern Languages*, Vol. 5 of *Pattern-Oriented Software Architecture*, Wiley.

Buyens, K., Scandariato, R. and Joosen, W.: 2009, Measuring the interplay of security principles in software architectures, *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*, IEEE, Lake Buena Vista, FL, USA, pp. 554–563.
DOI: *10.1109/ESEM.2009.5315968*

Canovas, J. and Molina, J.: 2010, An architecture-driven modernization tool for calculating metrics, *IEEE Software* **27**(4), 37–43.
DOI: *10.1109/MS.2010.61*

Cawley, O., Wang, X. and Richardson, I.: 2010, Lean/agile software development methodologies in regulated environments - State of the art, *First International Conference on Lean Enterprise Software and Systems (LESS '10)*, Helsinki, Finland, pp. 31–36.
DOI: *10.1007/978-3-642-16416-3_4*

Chantarasathaporn, K. and Srisa-an, C.: 2006, Energy conscious factory method design pattern for mobile devices with C# and intermediate language, *Proceedings of the Third international conference on Mobile technology, applications & systems (Mobility '06)*, ACM, Bangkok, Thailand, pp. 29:1–29:8.
DOI: *10.1145/1292331.1292364*

Charalampidou, S., Ampatzoglou, A., Avgeriou, P., Sencer, S., Arvanitou, E.-M. and Stamelos, I.: 2017, A theoretical model for capturing the impact of design patterns on quality, *Proceedings of 32nd ACM SIGAPP Symposium On Applied Computing (SAC '17)*, ACM Press, Marrakech, Morocco, pp. 1231–1238.
DOI: *10.1145/3019612.3019781*

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), 303–324.
DOI: *10.1002/smr.584*

Chen, H., Li, Y. and Shi, W.: 2012, Fine-grained power management using process-level profiling, *Sustainable Computing: Informatics and Systems* **2**(1), 33–42.

Chidamber, S. and Kemerer, C.: 1994, A metrics suite for object oriented design, *IEEE Transactions on Software Engineering* **20**(6), 476–493.
DOI: *10.1109/32.295895*

Cohen, F.: 2007, Identifying and avoiding SOA performance problems, *FastSOA*, Elsevier, pp. 75–101.
DOI: *10.1016/B978-012369513-0/50005-7*

Dale, M. R. and Izurieta, C.: 2014, Impacts of design pattern decay on system quality, *Proceedings of the Eighth ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*, ACM Press, Torino, Italy, pp. 37:1–37:4.
DOI: *10.1145/2652524.2652560*

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), 1–19.
DOI: *10.1016/j.jss.2007.07.006*

Di Penta, M., Cerulo, L., Guéhéneuc, Y. G. and Antoniol, G.: 2008, An empirical study of the relationships between design pattern roles and class change proneness, *Proceedings of the 24th International Conference on Software Maintenance (ICSM '08)*, IEEE, Beijing, China, pp. 217–226.
DOI: *10.1109/ICSM.2008.4658070*

Díaz, G. and Bermejo, J. R.: 2013, Static analysis of source code security: Assessment of tools against SAMATE tests, *Information and Software Technology* **55**(8), 1462–1476.
DOI: *10.1016/j.infsof.2013.02.005*

Dieste, O., Grimán, A. and Juristo, N.: 2009, Developing search strategies for detecting relevant experiments, *Empirical Software Engineering* **14**(5), 513–539.
DOI: *10.1007/s10664-008-9091-7*

Diouri, M. E. M., Dolz, M. F., Glück, O., Lefèvre, L., Alonso, P., Catalán, S., Mayo, R. and Quintana-Ortí, E. S.: 2014, Assessing power monitoring approaches for energy and power analysis of computers, *Sustainable Computing: Informatics and Systems* **4**(2), 68–82.
DOI: *10.1016/j.suscom.2014.03.006*

Do, T., Rawshdeh, S. and Shi, W.: 2009, ptop: A process-level power profiling tool, *Proceedings of the 2nd workshop on power aware computing and systems (HotPower'09)*.

Dromey, R.: 1995, A model for software product quality, *IEEE Transactions on Software Engineering* **21**(2), 146–162.
DOI: *10.1109/32.345830*

Dybå, T. and Dingsøyr, T.: 2008, Empirical studies of agile software development: A systematic review, *Information and Software Technology* **50**(9-10), 833–859.

Dybå, T., Dingsøyr, T. and Hanssen, G. K.: 2007, Applying systematic reviews to diverse study types: An experience report, *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM '07)*, Madrid, Spain, pp. 225–234.
DOI: *10.1109/ESEM.2007.21*

Eklund, U. and Bosch, J.: 2013, Archetypical approaches of fast software development and slow embedded projects, *Proceedings og the 39th Euromicro Conference Series on Software Engineering and Advanced Applications (SEAA '13)*, Santander, Spain, pp. 276–283.
DOI: *10.1109/SEAA.2013.38*

Elberzhager, F., Rosbach, A. and Bauer, T.: 2013, Analysis and testing of matlab simulink models: a systematic mapping study, *Proceedings og the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation (JAMAICA '13)*, Lugano, Switzerland, pp. 29–34.
DOI: *10.1145/2489280.2489285*

Evans, J. D.: 1996, *Straightforward statistics for the behavioral sciences*, Brooks/Cole Pub. Co., Pacific Grove.

Feitosa, D., Alders, R., Ampatzoglou, A., Avgeriou, P. and Nakagawa, E. Y.: 2017, Investigating the effect of design patterns on energy consumption, *Journal of Software: Evolution and Process* **29**(2), e1851.
DOI: *10.1002/smr.1851*

Feitosa, D., Ampatzoglou, A., Avgeriou, P., Affonso, F. J., Andrade, H., Felizardo, K. R. and Nakagawa, E. Y.: 2017, Supplementary Material: "Design approaches for critical embedded system: A systematic mapping study".
DOI: *10.5281/zenodo.996480*

Feitosa, D., Ampatzoglou, A., Avgeriou, P., Affonso, F. J., Andrade, H., Felizardo, K. R. and Nakagawa, E. Y.: 2018, Design Approaches for Critical Embedded Systems: A Systematic Mapping Study, *Evaluation of Novel Approaches to Software Engineering (ENASE '17)*, Springer, Cham, pp. 243–274.
DOI: *10.1007/978-3-319-94135-6_12*

Feitosa, D., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A. and Nakagawa, E.: 2018, What can violations of good practices tell about the relationship between GoF patterns and runtime quality attributes?, *Information and Software Technology* .
DOI: *10.1016/j.infsof.2018.07.014*

Feitosa, D., Ampatzoglou, A., Avgeriou, P. and Nakagawa, E. Y.: 2015, Investigating quality trade-offs in open source critical embedded systems, *Proceedings of the 11th International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA '15)*, ACM, Montreal, QC, Canada, pp. 113–122.
DOI: *10.1145/2737182.2737190*

Feitosa, D., Ampatzoglou, A., Avgeriou, P. and Nakagawa, E. Y.: 2018, Correlating Pattern Grime and Quality Attributes, *IEEE Access* **6**, 23065–23078.
DOI: *10.1109/ACCESS.2018.2829895*

Feitosa, D., Avgeriou, P., Ampatzoglou, A. and Nakagawa, E. Y.: 2017a, Supplementary Material: "The Evolution of Design Pattern Grime: An Industrial Case Study".
DOI: *10.5281/zenodo.806800*

Feitosa, D., Avgeriou, P., Ampatzoglou, A. and Nakagawa, E. Y.: 2017b, The evolution of design pattern grime: An industrial case study, *Proceedings of the 18th International Conference on Product-Focused Software Process Improvement (PROFES '17)*, Innsbruck, Austria, pp. 165–181.
DOI: *10.1007/978-3-319-69926-4_13*

Fernandez-Buglioni, E.: 2013, *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*, Wiley Software Patterns Series, Wiley.

Ferraz, F. S., Assad, R. E. and Lemos Meira, S. R.: 2009, Relating security requirements and design patterns: Reducing security requirements implementation impacts with design patterns, *Proceedings of the Fourth International Conference on Software Engineering Advances (ICSEA '09)*, IEEE, Porto, Portugal, pp. 9–14.
DOI: *10.1109/ICSEA.2009.10*

Ferreira, L. L. and Rubira, C. M. F.: 1998, The reflective state pattern, *Proceedings of the Pattern Languages of Program Design Monticello, Illinois-USA* .

Field, A.: 2013, *Discovering Statistics Using IBM SPSS Statistics*, 4th edn, SAGE Publications Ltd.

Firesmith, D. G.: 2003, Engineering security requirements.
DOI: *10.5381/jot.2003.2.1.c6*

Fleck, G., Kirchmayr, W., Moser, M., Nocke, L., Pichler, J., Tober, R. and Witlatschil, M.: 2016, Experience report on building astm based tools for multi-language reverse engineering, *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*, pp. 683–687.
DOI: *10.1109/SANER.2016.33*

Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D.: 1999, Refactoring: improving the design of existing code, *ISBN: 0-201-48567-2* .

Gajski, D. D., Zhu, J., Dömer, R., Gerstlauer, A. and Zhao, S.: 2000, *SPECC: Specification Language and Methodology*, Springer US, Boston, MA.
DOI: *10.1007/978-1-4615-4515-6*

Gamma, E., Helm, R., Johnson, R. E. and Vlissides, J.: 1995, *Design patterns: elements of reusable object-oriented software*, Vol. 206, Addison-Wesley Longman Publishing Co., Inc.

Gatrell, M. and Counsell, S.: 2011, Design patterns and fault-proneness a study of commercial C# software, *Proceedings of the Fifth International Conference on Research Challenges in Information Science (RCIS '11)*, IEEE, Gosier, France, pp. 1–8.
DOI: *10.1109/RCIS.2011.6006827*

Goseva-Popstojanova, K. and Perhinschi, A.: 2015, On the capability of static code analysis to detect security vulnerabilities, *Information and Software Technology* **68**, 18–33.
DOI: *10.1016/j.infsof.2015.08.002*

Grady, R. B.: 1992, *Practical software metrics for project management and process improvement*, Prentice-Hall, Inc.

Griffith, I. and Izurieta, C.: 2014, Design pattern decay: the case for class grime, *Proceedings of the Eighth ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*, ACM Press, Torino, Italy, pp. 39:1–39:4.
DOI: *10.1145/2652524.2652570*

Grimmer, M., Seaton, C., Schatz, R., Würthinger, T. and Mössenböck, H.: 2015, High-performance cross-language interoperability in a multi-language runtime, *ACM SIGPLAN Notices* **51**(2), 78–90.
DOI: *10.1145/2936313.2816714*

Guessi, M., Nakagawa, E. Y., Oquendo, F. and Maldonado, J. C.: 2012, Architectural description of embedded systems: A systematic review, *Proceedings of the Third International ACM SIGSOFT Symposium on Architecting Critical Systems (ISARCS '12)*, ACM, Bertinoro, Italy, pp. 31–40.
DOI: *10.1145/2304656.2304661*

Hafiz, M., Adamczyk, P. and Johnson, R. E.: 2007, Organizing security patterns, *IEEE Software* **24**(4), 52–60.
DOI: *10.1109/MS.2007.114*

Haghighatkhah, A., Oivo, M., Banijamali, A. and Kuvaja, P.: 2017, Improving the state of automotive software engineering, *IEEE Software* **34**(5), 82–86.
DOI: *10.1109/MS.2017.3571571*

Hammadi, A. and Mhamdi, L.: 2014, A survey on architectures and energy efficiency in data center networks, *Computer Communications* **40**, 1–21.
DOI: *10.1016/j.comcom.2013.11.005*

Harper, R. and Morrisett, G.: 1995, Compiling polymorphism using intensional type analysis, *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*, ACM Press, San Francisco, CA, USA, pp. 130–141.
DOI: *10.1145/199448.199475*

Hastie, T., Tibshirani, R. and Friedman, J.: 2009, *The Elements of Statistical Learning*, Springer Series in Statistics, Springer New York, New York, NY.
DOI: *10.1007/978-0-387-84858-7*

Heath, S.: 2002, *Embedded Systems Design*, Elsevier.
DOI: *10.1016/B978-0-7506-5546-0.X5000-2*

Helmerich, A., Koch, N., Mandel, L., Braun, P., Dornbusch, P., Gruler, A., Keil, P., Leisibach, R., Romberg, J., Schätz, B., Wild, T. and Wimmel, G.: 2005, Study of worldwide trends and R&D programmes in embedded systems in siew of maximising the impact of a technology platform in the area, *Technical report*, Information Society Technologies.
DOI: *KK-04-14-422-EN-N*

Henney, K.: 1999, Collections for States, *Proceedings of the Fourth European Conference on Pattern Languages of Programms (EuroPLoP '99)*, Irsee, Germany, pp. 57–64.

Henney, K.: 2002, Methods for States, *Proceedings of the First Nordic Conference on Pattern Languages of Programming (VikingPLoP '02)*, Højstrupgård, Denmark.

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A. and America, P.: 2007, A general model of software architecture design derived from five industrial approaches, *Journal of Systems and Software* **80**(1), 106–126.
DOI: *10.1016/j.jss.2006.05.024*

Hora, A., Anquetil, N., Ducasse, S. and Allier, S.: 2012, Domain specific warnings: Are they any better?, *Proceedings of the 28th International Conference on Software Maintenance (ICSM '12)*, IEEE, Trento, Italy, pp. 441–450.
DOI: *10.1109/ICSM.2012.6405305*

Hovemeyer, D. and Pugh, W.: 2004, Finding bugs is easy, *ACM SIGPLAN Notices* **39**(12), 92–106.
DOI: *10.1145/1052883.1052895*

Hsueh, N.-L., Chu, P.-H. and Chu, W.: 2008, A quantitative approach for evaluating the quality of design patterns, *Journal of Systems and Software* **81**(8), 1430–1439.
DOI: *10.1016/j.jss.2007.11.724*

Huston, B.: 2001, The effects of design pattern application on metric scores, *Journal of Systems and Software* **58**(3), 261–269.
DOI: *10.1016/S0164-1212(01)00043-7*

ISO/IEC: 2011, ISO/IEC 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, *Technical report*, ISO/IEC.
URL: *http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733*

ISO/IEC: 2015, ISO/IEC 15026-3:2015 Systems and software engineering – Systems and software assurance – Part 3: System integrity levels, *Technical report*, ISO/IEC.
URL: *http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64842*

ISO/IEC/IEEE: 2010, ISO/IEC/IEEE 24765-2010 - Systems and software engineering – Vocabulary, *Technical report*, ISO/IEC/IEEE.
DOI: *10.1109/IEEESTD.2010.5733835*

Izurieta, C. and Bieman, J. M.: 2007, How software designs decay: A pilot study of pattern evolution, *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM '07)*, IEEE, Madrid, Spain, pp. 449–451.
DOI: *10.1109/ESEM.2007.55*

Izurieta, C. and Bieman, J. M.: 2008, Testing consequences of grime buildup in object oriented design patterns, *Proceedings of the First International Conference on Software Testing, Verification, and Validation (ICST '08)*, IEEE, Lillehammer, Norway, pp. 171–179.
DOI: *10.1109/ICST.2008.27*

Izurieta, C. and Bieman, J. M.: 2013, A multiple case study of design pattern decay, grime, and rot in evolving software systems, *Software Quality Journal* **21**(2), 289–323.
DOI: *10.1007/s11219-012-9175-x*

Jain, R., Molnar, D. and Ramzan, Z.: 2005, Towards understanding algorithmic factors affecting energy consumption: switching complexity, randomness, and preliminary experiments, *Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing (DIALM-POMC '05)*, ACM, Cologne, Germany, pp. 70–79.

Jedlitschka, A., Ciolkowski, M. and Pfahl, D.: 2008, Reporting experiments in software engineering, *Guide to Advanced Empirical Software Engineering*, Springer London, London, pp. 201–228.
DOI: *10.1007/978-1-84800-044-5_8*

Johann, T., Dick, M., Naumann, S. and Kern, E.: 2012, How to measure energy-efficiency of software: Metrics and measurement results, *Proceedings of the First International Workshop on Green and Sustainable Software (GREENS '12)*, IEEE, Zurich, Switzerland, pp. 51–54.

Kacimi, O., Ellen, C., Oertel, M. and Sojka, D.: 2014, Creating a reference technology platform - Performing model-based safety analysis in a heterogeneous development environment,

Second International Conference on Model-Driven Engineering and Software Development (MODELSWARD '14), Lisbon, Portugal, pp. 645–652.
DOI: *10.5220/0004875306450652*

Karlström, D. and Runeson, P.: 2006, Integrating agile software development into stage-gate managed product development, *Empirical Software Engineering* **11**(2), 203–225.
DOI: *10.1007/s10664-006-6402-8*

Khalid, H., Nagappan, M. and Hassan, A. E.: 2016, Examining the relationship between FindBugs warnings and app ratings, *IEEE Software* **33**(4), 34–39.
DOI: *10.1109/MS.2015.29*

Khomh, F., Gueheneuc, Y.-G. and Antoniol, G.: 2009, Playing roles in design patterns: An empirical descriptive and analytic study, *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09)*, IEEE, Edmonton, AB, Canada, pp. 83–92.
DOI: *10.1109/ICSM.2009.5306327*

Kitchenham, B. A., Dyba, T. and Jorgensen, M.: 2004, Evidence-based software engineering, *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, IEEE Comput. Soc, Edinburgh, UK, pp. 23–28.
DOI: *10.1109/icse.2004.1317449*

Kitchenham, B. and Charters, S.: 2007, Guidelines for performing systematic literature reviews in software engineering, *Engineering* **2**, 1051.

Kitchenham, B. and Pfleeger, S.: 1996, Software quality: the elusive target [special issues section], *IEEE Software* **13**(1), 12–21.
DOI: *10.1109/52.476281*

Kniesel, G. and Binun, A.: 2009, Standing on the shoulders of giants - A data fusion approach to design pattern detection, *Proceedings of the 17th International Conference on Program Comprehension (ICPC '09)*, IEEE, Vancouver, BC, Canada, pp. 208–217.
DOI: *10.1109/ICPC.2009.5090044*

Li, W. and Henry, S.: 1993, Object-oriented metrics that predict maintainability, *Journal of Systems and Software* **23**(2), 111–122.
DOI: *10.1016/0164-1212(93)90077-B*

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 antipatterns, application domains, and quality-related metrics in Java mobile apps, *Proceedings of the 22nd International Conference on Program Comprehension (ICPC '14)*, ACM Press, Hyderabad, India, pp. 232–243.
DOI: *10.1145/2597008.2597144*

Litke, A., Zotos, K., Chatzigeorgiou, A. and Stephanides, G.: 2007, Energy consumption analysis of design patterns, *International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering* **1**(11), 1663–1667.

Liu, Y. D.: 2012, Energy-efficient synchronization through program patterns, *Proceedings of the First International Workshop on Green and Sustainable Software (GREENS '12)*, IEEE, Zurich, Switzerland, pp. 35–40.

Lyardet, F. D.: 1997, The dynamic template pattern, *Proceedings of the Conference on Pattern Languages of Design (PLoP '97)*.

Manotas, I., Pollock, L. and Clause, J.: 2014, SEedS: a software engineer's energy-optimization decision support framework, *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, ACM Press, Hyderabad, India, pp. 503–514.
DOI: *10.1145/2568225.2568297*

March, S. T. and Smith, G. F.: 1995, Design and natural science research on information technology, *Decision Support Systems* **15**(4), 251–266.
DOI: *10.1016/0167-9236(94)00041-2*

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

Marwedel, P.: 2011, *Embedded System Design*, Embedded Systems, 2 edn, Springer Netherlands, Dordrecht.
DOI: *10.1007/978-94-007-0257-8*

McCall, J. A.: 1977, Factors in software quality, *Technical report*, US Rome Air development center reports.

McNatt, W. B. and Bieman, J. M.: 2001, Coupling of design patterns: common practices and their benefits, *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC '01)*, IEEE, Chicago, IL, USA, pp. 574–579.
DOI: *10.1109/CMPSAC.2001.960670*

Medikonda, B. S. and Panchumarthy, S. R.: 2009, A framework for software safety in safety-critical systems, *ACM SIGSOFT Software Engineering Notes* **34**(2), 1.
DOI: *10.1145/1507195.1507207*

Misra, S. C. and Bhavsar, V. C.: 2003, Relationships Between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality, *in* V. Kumar, M. L. Gavrilova, C. J. K. Tan and P. L'Ecuyer (eds), *Computational Science and Its Applications (ICCSA 2003)*, Vol. 2667 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 724–732.
DOI: *10.1007/3-540-44839-X*

Miyashiro, M. A. S., Ferreira, M. G. V. and Sant'Anna, N.: 2015, CMmi-Dev process areas modeled on a process for critical embedded systems development, *Proceedings of the 2015 Science and Information Conference (SAI '15)*, IEEE, London, UK, pp. 870–878.
DOI: *10.1109/SAI.2015.7237245*

Muraki, T. and Saeki, M.: 2002, Metrics for applying GoF design patterns in refactoring processes, *Proceedings of the Fourth International workshop on Principles of software evolution (IW-PSE '01)*, ACM Press, Vienna, Austria, pp. 27–36.
DOI: *10.1145/602461.602466*

Nakagawa, E. Y., Gonçalves, M., Guessi, M., Oliveira, L. B. R. and Oquendo, F.: 2013, The state of the art and future perspectives in systems of systems software architectures, *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems (SESoS '13)*, Montpellier, France, pp. 13–20.
DOI: *10.1145/2489850.2489853*

Noureddine, A., Bourdon, A., Rouvoy, R. and Seinturier, L.: 2012a, A preliminary study of the impact of software engineering on greenit, *Proceedings of the First International Workshop on Green and Sustainable Software (GREENS '12)*, IEEE, Zurich, Switzerland, pp. 21–27.

Noureddine, A., Bourdon, A., Rouvoy, R. and Seinturier, L.: 2012b, Runtime monitoring of software energy hotspots, *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE '12)*, IEEE, Essen, Germany, pp. 160–169.

Noureddine, A. and Rajan, A.: 2015, Optimising energy consumption of design patterns, *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, Florence, Italy.

Noureddine, A., Rouvoy, R. and Seinturier, L.: 2013, A review of energy measurement approaches, *ACM SIGOPS Operating Systems Review* **47**(3), 42–49.

Noureddine, A., Rouvoy, R. and Seinturier, L.: 2014, Unit testing of energy consumption of software libraries, *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC '14)*, ACM, ACM Press, Gyeongju, Korea, pp. 1200–1205.
DOI: *10.1145/2554850.2554932*

Noureddine, A., Rouvoy, R. and Seinturier, L.: 2015, Monitoring energy hotspots in software, *Automated Software Engineering* pp. 1–42.

Oliveira, L. B. R., Guessi, M., Feitosa, D., Manteuffel, C., Galster, M., Oquendo, F. and Nakagawa, E. Y.: 2013, An Investigation on Quality Models and Quality Attributes for Embedded Systems, *Proceedings of the Eighth International Conference on Software Engineering Advances (ICSEA '13)*, IARIA XPS Press, Venice, Italy, pp. 523–528.

Oliveira, M. F., Redin, R. M., Carro, L., Lamb, L. and Wagner, F.: 2008, Software quality metrics and their impact on embedded software, *Proceedings of the Fifth International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2008)*, IEEE, Budapest, Hungary, pp. 68–77.
DOI: *10.1109/MOMPES.2008.11*

Ollila, J., Sangiovanni, A., Vincentelli, Kleisterlee, G., Dais, S., Pistorio, P., Levin, D., Baksaas, J. F., Ranque, D., Skalicky, P. and Petit, A.: 2004, Building ArtEmiS: Report by the High-Level Group on Embedded Systems, *Technical report*, Information Society Technologies and

European Commission.
DOI: *KK-60-04-296-EN-C*

Patton, M. Q.: 2014, *Qualitative Research & Evaluation Methods: Integrating Theory and Practice*, SAGE Publications.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C. and Seinturier, L.: 2016, SPooN: A library for implementing analyses and transformations of Java source code, *Software: Practice and Experience* **46**(9), 1155–1179.
DOI: *10.1002/spe.2346*

Pelliccione, P., Knauss, E., Heldal, R., Magnus Ågren, S., Mallozzi, P., Alminger, A. and Borgentun, D.: 2017, Automotive Architecture Framework: The experience of Volvo Cars, *Journal of Systems Architecture* **77**, 83–100.
DOI: *10.1016/j.sysarc.2017.02.005*

Penta, M. D., Cerulo, L. and Aversano, L.: 2009, The life and death of statically detected vulnerabilities: An empirical study, *Information and Software Technology* **51**(10), 1469–1484.
DOI: *10.1016/j.infsof.2009.04.013*

Perez-Castillo, R. and Piattini, M.: 2014, Analyzing the harmful effect of god class refactoring on power consumption, *Software, IEEE* **31**(3), 48–54.

Perry, D. E. and Wolf, A. L.: 1992, Foundations for the study of software architecture, *ACM SIGSOFT Software Engineering Notes* **17**(4), 40–52.
DOI: *10.1145/141874.141884*

Petersen, K., Feldt, R., Mujtaba, S. and Mattsson, M.: 2008, Systematic mapping studies in software engineering, *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE '08)*, Bari, Italy, pp. 68–77.
DOI: *10.1142/S0218194007003112*

Pétrissans, A., Krawczyk, S., Cattaneo, G., Feeney, N., Veronesi, L. and Meunier, C.: 2012, Final Study Report: Design of Future Embedded Systems (SmaRt 2009 / 0063), *Technical report*, International Data Corporation.
DOI: *KK-04-13-201-EN-N*

Pettersson, N., Löwe, W. and Nivre, J.: 2010, Evaluation of accuracy in design pattern occurrence detection, *IEEE Transactions on Software Engineering* **36**(4), 575–590.
DOI: *10.1109/TSE.2009.92*

Pinto, G., Castor, F. and Liu, Y. D.: 2014, Understanding energy behaviors of thread management constructs, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, ACM, Portland, OR, USA, pp. 345–360.
DOI: *10.1145/2714064.2660235*

Procaccianti, G., Lago, P. and Bevini, S.: 2015, A systematic literature review on energy efficiency in cloud software architectures, *Sustainable Computing: Informatics and Systems* **7**, 2–10.
DOI: *10.1016/j.suscom.2014.11.004*

Riaz, M., Breaux, T. and Williams, L.: 2015, How Have We Evaluated Software Pattern Application? A systematic Mapping Study of Research Design Practices, *Information and Software Technology* **65**, 14–38.
DOI: *10.1016/j.infsof.2015.04.002*

Riehle, D.: 2011, Lessons Learned from Using Design Patterns in Industry Projects, *Transactions on Pattern Languages of Programming II*, Vol. 6510 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 1–15.
DOI: *10.1007/978-3-642-19432-0_1*

Romano, D., Raila, P., Pinzger, M. and Khomh, F.: 2012, Analyzing the impact of antipatterns on change-proneness using uine-grained source code changes, *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE '12)*, IEEE, Kingston, ON, Canada, pp. 437–446.
DOI: *10.1109/WCRE.2012.53*

Rudzki, J.: 2004, How design patterns affect application performance – a case of a multi-tier J2eE application, *Proceedings of the Fourth International Workshop on Scientific Engineering of Distributed Java Applications (FIDJI '04)*, Springer-Verlag, Luxembourg-Kirchberg, Luxembourg, pp. 12–23.
DOI: *10.1007/978-3-540-31869-9_2*

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

Sadowski, C., van Gogh, J., Jaspan, C., Soderberg, E. and Winter, C.: 2015, Tricorder: Building a program analysis ecosystem, *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE '15)*, IEEE, Florence, Italy, pp. 598–608.
DOI: *10.1109/ICSE.2015.76*

Sahin, C., Cayci, F., Gutiérrez, I. L. M., Clause, J., Kiamilev, F., Pollock, L. and Winbladh, K.: 2012, Initial explorations on design pattern energy usage, *Proceedings of the First International Workshop on Green and Sustainable Software (GREENS '12)*, IEEE, Zurich, Switzerland, pp. 55–61.
DOI: *10.1109/GREENS.2012.6224257*

Sahin, C., Pollock, L. and Clause, J.: 2014, How do code refactorings affect energy usage?, *Proceedings of the Eighth ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*, ACM, ACM Press, Torino, Italy, pp. 36:1–36:10.
DOI: *10.1145/2652524.2652538*

Schanz, T. and Izurieta, C.: 2010, Object oriented design pattern decay, *Proceedings of the Fourth ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*, ACM Press, Bolzano-Bozen, Italy, pp. 7:1–7:8.
DOI: *10.1145/1852786.1852796*

Schilling, W. and Alam, M.: 2008, A methodology for quantitative evaluation of software reliability using static analysis, *Proceedings of the 2008 Annual Reliability and Maintainability Symposium (RAMS '08)*, IEEE, Las Vegas, NV, USA, pp. 399–404.
DOI: *10.1109/RAMS.2008.4925829*

Selim, G. M. K., Wang, S., Cordy, J. R. and Dingel, J.: 2012, Model transformations for migrating legacy models: An industrial case study, *Proceedings of the Eighth European Conference on Modelling Foundations and Applications (ECMFA '12)*, Springer, Kgs. Lyngby, Denmark, pp. 90–101.
DOI: *10.1007/978-3-642-31491-9_9*

Seng, O., Stammel, J. and Burkhart, D.: 2006, Search-based determination of refactorings for improving the class structure of object-oriented systems, *Proceedings of the Eighth Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, ACM Press, Seattle, WA, USA, pp. 1909–1916.
DOI: *10.1145/1143997.1144315*

Sobajic, O., Moussavi, M. and Far, B.: 2010, Extending the strategy pattern for parameterized algorithms, *Proceedings of the 17th Conference on Pattern Languages of Programs (PLOP'10)*, Reno/Tahoe, NV, USA.

Sommerville, I.: 2000, *Software Engineering*, 6 edn, Addison Wesley.

Suryn, W.: 2014, *Software Quality Engineering*, Wiley Blackwell.
DOI: *10.1002/9781118830208*

Tamrabet, Z., Marir, T. and Mokhati, F.: 2018, A survey on quality attributes and quality models for embedded software, *International Journal of Embedded and Real-Time Communication Systems* **9**(2), 1–17.
DOI: *10.4018/IJERTCS.2018070101*

Thompson, H., Lora-Tamayo, E., Damm, W., Dormoy, J.-L., Hobbs, L., Jansz, M., Kosmider, T. and Pribyl, W.: 2017a, Final evaluation of the ARTEMIS and ENIAC joint undertaking (2008-2013) operating under Fp7, *Technical report*, European Comission.
DOI: *10.2759/271765*

Thompson, H., Lora-Tamayo, E., Damm, W., Dormoy, J.-L., Hobbs, L., Jansz, M., Kosmider, T. and Pribyl, W.: 2017b, Interim evaluation of the ECSEL joint undertaking (2014-2016) operating under Horizon 2020, *Technical report*, European Comission.
DOI: *10.2759/614017*

Tiwari, V., Malik, S., Wolfe, A. and Lee, M. T.-C.: 1996, Instruction level power analysis and optimization of software, *Technologies for wireless computing*, Springer, pp. 139–154.

Tripathi, A. K. and Gupta, A.: 2014, A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for Java programs, *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14)*, ACM, London, UK, pp. 23:1–23:4.
DOI: *10.1145/2601248.2601288*

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. and Halkidis, S.: 2006, Design pattern detection using similarity scoring, *IEEE Transactions on Software Engineering* **32**(11), 896–909.
DOI: *10.1109/TSE.2006.112*

van Solingen, R., Basili, V., Caldiera, G. and Rombach, H. D.: 2002, Goal Question Metric (GQM) approach, *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc., Hoboken, NJ, USA, pp. 528–532.
DOI: *10.1002/0471028959.sof142*

VanHilst, M. and Fernandez, E. B.: 2007, Reverse engineering to detect security patterns in code, *Proceedings of 1st International Workshop on Software Patterns and Quality*, Information Processing Society of Japan, pp. 1–6.

Victório, R. A. S. S., Coutinho, G. C. A. and Others: 2010, Persistent state pattern, *Proceedings of the 17th Conference on Pattern Languages of Programs (PLoP '10)*, ACM, Reno/Tahoe, NV, USA, pp. 5:1–5:16.
DOI: *10.1145/2493288.2493293*

Vokac, M.: 2004, Defect frequency and design patterns: an empirical study of industrial code, *IEEE Transactions on Software Engineering* **30**(12), 904–917.
DOI: *10.1109/TSE.2004.99*

Weisfeld, M.: 2013, *The Object-Oriented Thought Process*, 4 edn, Addison-Wesley Professional.

Wieringa, R.: 2014, *Design Science Methodology for Information Systems and Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg.
DOI: *10.1007/978-3-662-43839-8*

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. and Wesslén, A.: 2012, *Experimentation in Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg.
DOI: *10.1007/978-3-642-29044-2*

Zaman, S., Adams, B. and Hassan, A. E.: 2011, Security versus performance bugs, *Proceedings of the Eighth Working Conference on Mining Software Repositories (MSR '11)*, ACM Press, Honolulu , HI, USA, pp. 93–102.
DOI: *10.1145/1985441.1985457*

Zhang, C. and Budgen, D.: 2012, What do we know about the effectiveness of software design patterns?, *IEEE Transactions on Software Engineering* **38**(5), 1213–1231.
DOI: *10.1109/TSE.2011.79*

Zhang, H. and Babar, M. A.: 2010, On searching relevant studies in software engineering, *Proceedings of the 14th international conference on Evaluation and Assessment in Software Engineering (EASE '10)*, British Computer Society, Keele, UK, pp. 111–120.

Zhang, Z., Cai, Y.-Y., Zhang, Y., Gu, D.-J. and Liu, Y.-F.: 2016, A distributed architecture based on microbank modules with self-reconfiguration control to improve the energy efficiency in the battery energy storage system, *IEEE Transactions on Power Electronics* **31**(1), 304–317.
DOI: *10.1109/TPEL.2015.2406773*

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P. and on Vouk, M. A. S. E. I. T.: 2006, On the value of static analysis for fault detection in software, *Software Engineering, IEEE Transactions on* **32**(4), 240–253.
DOI: *10.1109/TSE.2006.38*