Universidade Federal da Bahia

Universidade Estadual de Feira de Santana

DISSERTAÇÃO DE MESTRADO

Understanding Architectural Bad Smells in Software Product Lines

Hugo Sica de Andrade

Mestrado Multiinstitucional em Ciência da Computação

MMCC

Salvador – BA

2014

Federal University of Bahia
Mathematics Institute - Computer Science Department
Master's Degree in Computer Science

Hugo Sica de Andrade

"Understanding Architectural Bad Smells in Software Product Lines"

A M.Sc. Dissertation presented to the Mathematics Institute - Computer Science Department of Federal University of Bahia in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: *Eduardo Santana de Almeida*

SALVADOR, AUGUST/2014

Sistema de Bibliotecas da UFBA

Andrade, Hugo Sica de.
"Understanding architectural bad smells in software product lines" / Hugo Sica de Andrade. - 2014.
163 f.: il.

Inclui apêndices.
Advisor: Eduardo Santana de Almeida.
Dissertação (mestrado) - Universidade Federal da Bahia, Instituto de Matemática, Salvador, 2014.

1. Software. 2. Software - Produto. 3. Software - Arquitetura. 4. Engenharia de software. 5. Computação. I. Almeida, Eduardo Santana de. II. Universidade Federal da Bahia. Instituto de Matemática. III. Título.

CDD - 005.1
CDU - 004.4

I dedicate this dissertation to my family and girlfriend, who kindly provided me with patience and knowledge to help me achieve my goals.

Acknowledgements

First and foremost I would like to thank God for providing me with the patience and the knowledge required to accomplish my goals so far.

Thank you very much to my parents Hélio and Gina, and my sister Débora for being such a great motivation and the reason why I am who I am.

Special thanks to my girlfriend Mariana who has trusted, loved and supported me in every tough decision that I have made since the moment we have met. I am with you, ris!

Furthermore, I would like to thank professors Eduardo Almeida and Ivica Crnkovic for clarifying things and helping me resolve many of the issues I have encountered while working on my master degree.

Thanks to all my colleagues, in special my bachelor's advisor Luanna Lobato, for having me initiated in academia and for supporting me while I pursue more and more achievements in this field.

Last but not least, I would like to thank my friends from many different countries, colleagues at UFBA and MDH (Sweden), and all members of RiSE (Reuse in Software Engineering) research group for sharing their knowledge over discussions and presentations, thus contributing to enrich this work.

Resumo

O paradigma de Linhas de Produto de Software (LPS) tem provado ser um meio efetivo para se obter reuso de grande escala em diferentes domínios. A abordagem tira proveito de aspectos comuns entre diferentes produtos, enquanto também considera propriedades específicas dos mesmos. A arquitetura tem um papel importante na engenharia de LPS, provendo meios para melhor entender e manter o ambiente de derivação de produtos. No entanto, é difícil evoluir tal arquitetura, pois nem sempre é claro onde e como refatorar.

A arquitetura de uma LPS contém um modelo que irá resultar na arquitetura de produtos, e muitas vezes inclui soluções que indicam um *design* (arquitetural) inadequado. Uma forma de avaliar tais decisões de *design* é através da identificação de *bad smells* de arquitetura, ou seja, propriedades que prejudicam a qualidade do software, mas não são necessariamente errôneas ou representam falhas.

Nesse sentido, o objetivo desta dissertação é obter um melhor entendimento de *bad smells* de arquitetura em LPSs. Primeiramente, o estado-da-arte atual em Arquiteturas de Linhas de Produto de software (ALP) é investigado através de um estudo de mapeamento sistemático. Este apresenta uma visão geral da área através de análise e categorização de evidências. O estudo idenfitica gaps, tendências, e provê direções futuras para pesquisa.

Ademais, esta dissertação trata do fenômeno de *bad smells* de arquitetura no contexto de LPSs através de dois estudos exploratórios em domínios diferentes. O primeiro estudo exploratório conduz uma investigação sobre as implicações de propriedades estruturais em uma LPS no domínio de editores de texto, enquanto o segundo estudo foca em uma LPS no domínio mobile. Antes da busca pelos *smells* em ambos os estudos, informações relevantes para a arquitetura foram recuperadas do código fonte para que as arquiteturas fossem definidas.

Palavras-chave: Linhas de Produto de Software, Arquitetura de Software, Estudo de Mapeamento, *Bad Smells* de Arquitetura

Abstract

The Software Product Line (SPL) paradigm has proven to be an effective way to achieve large scale reuse in different domains. It takes advantage of common aspects between different products, while also considering product specific features. The architecture plays an important role in SPL engineering, by providing means to better understand and maintain the product-derivation environment. However, it is difficult to evolve such architecture because it is not always clear where and how to refactor.

The architecture of a SPL comprises a model that will result in product architectures, and may include solutions leading to bad (architectural) design. One way to assess such design decisions is through the identification of architectural bad smells, which are properties that prejudice the overall software quality, but are not necessarily faulty or errant.

In this sense, the goal of this dissertation is to obtain a better understanding of architectural bad smells in SPLs. First, the current state of the art of software Product Line Architectures (PLAs) is investigated through a systematic mapping study. It provides an overview of the field through the analysis, and categorization of evidence. The study identifies gaps, trends and provides future directions for research.

Furthermore, this dissertation addresses the phenomenon of architectural bad smells in the context of SPLs through two exploratory studies in different domains. The first exploratory case study provides an investigation on the implications of such structural properties in a text editor domain SPL, while the second study aims at a SPL in the mobile domain. Prior to the search for smells in both studies, architecturally relevant information was recovered from the source code in order to define their architectures.

Keywords: Software Product Lines, Software Architecture, Mapping Study, Architectural Bad Smells

Table of Contents

List of Figures xi

List of Tables xiii

List of Acronyms xiv

1 Introduction 1

1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Related Work . . . 3
1.3.1 Literature Reviews . . . 3
1.3.2 Architecture Evaluation . . . 4
1.4 Out of Scope . . . 6
1.5 Statement of the Contributions . . . 6
1.6 Research Design . . . 7
1.7 Dissertation Structure . . . 8

2 An Overview on Software Product Lines, Software Architecture and Architectural Bad Smells 10

2.1 Software Product Lines . . . 10
2.1.1 Benefits of SPL Engineering . . . 11
2.1.2 SPL Engineering Essential Activities . . . 12
2.2 Software Architecture . . . 15
2.2.1 The Architecture of a Software Product Line . . . 16
2.2.2 Terminologies . . . 17
2.3 Architectural Bad Smells . . . 17
2.3.1 Definition . . . 18
2.3.2 Representative Smells . . . 19
Connector Envy . . . 20
Scattered Parasitic Functionality . . . 22
Ambiguous Interfaces . . . 23
Extraneous Adjacent Connector . . . 24
2.4 Chapter Summary . . . 25

3 A Systematic Mapping Study on Software Product Lines Architecture 26
3.1 Motivation 26
3.2 Review Method 27
3.3 Review Process 28
3.4 Planning 29
3.4.1 Research Questions 30
3.4.2 Viewpoints 31
3.4.3 Search Strategy 31
3.5 Conducting 33
3.5.1 Screening of Papers 34
3.5.2 Classification Scheme 35
3.5.3 Data Extraction 37
3.6 Outcomes 38
3.6.1 An Overview of the PLA field 38
3.6.2 Findings 40
RQ1 - Are architectural patterns (or styles) used in SPL? 43
RQ2 - How is variability handled in the architecture level of SPLs? 47
RQ3 - How are the SPL architectures documented? 52
RQ4 - How are the SPL architectures evaluated? 56
3.7 Discussion 59
3.7.1 Main Findings 60
Patterns 60
Variability 61
Documentation 62
Evaluation 62
3.8 Threats to Validity 63
3.9 Chapter Summary 63

4 Architectural Bad Smells in Software Product Lines: An Exploratory Study 65
4.1 Study Setup 65
4.2 Notepad SPL 66
4.2.1 Feature Model 67
4.2.2 Variability Management 69
4.2.3 Product Map 69
4.3 Architecture Recovery 70
4.3.1 Recovery Process 72

4.3.2 Automated Analysis 72
4.3.3 Manual Analysis 74
4.3.4 Merging Product Architectures 75
4.4 Identifying Architectural Bad Smells 76
4.4.1 Connector Envy 77
4.4.2 Scattered Parasitic Functionality 78
4.4.3 Ambiguous Interfaces 78
4.4.4 Extraneous Adjacent Connector 79
4.4.5 Feature Concentration 80
4.5 Threats to Validity 81
4.6 Chapter Summary 81

5 A Replicated Study on Architectural Bad Smells in Software Product Lines 83
5.1 Study Setup 83
5.2 RescueMe SPL 84
5.2.1 Feature Model 86
5.2.2 Variability Management 86
5.2.3 Product Map 88
5.3 Architecture Recovery 89
5.3.1 Recovery Process 91
5.3.2 Automated Analysis 91
5.3.3 Manual Analysis 95
5.3.4 Merging Product Architectures 96
5.4 Identifying Architectural Bad Smells 98
5.4.1 Connector Envy 98
5.4.2 Scattered Parasitic Functionality 99
5.4.3 Ambiguous Interfaces 100
5.4.4 Extraneous Adjacent Connector 101
5.4.5 Feature Concentration 101
5.5 Comparative Analysis 102
5.5.1 Features 102
5.5.2 Domain 104
5.5.3 Lines of code and complexity 105
5.5.4 Variability technique 105
5.5.5 Patterns 105
5.6 Main Findings 106

5.7 Threats to Validity 107
5.8 Chapter Summary 108

6 Conclusions **109**
6.1 Published Work 110
6.2 Future Work 110
6.3 Concluding Remarks 111

References **112**

Appendix **131**

A Mapping Study **132**
A.1 List of Journals Manually Searched 132
A.2 List of Conferences Manually Searched 133
A.3 Data Extraction Report 134
A.4 Primary Studies Mapped According to Aspects in PLA 135

B Exploratory Study **140**
B.1 Notepad SPL Architecture Specification 140

C Replicated Study **144**
C.1 RescueMe SPL Architecture Specification 144
C.1.1 Model 144
C.1.2 View 145
C.1.3 Controller 146
C.1.4 3rd Party 149

List of Figures

1.1 Workflow of the Research conducted in this work. 8

2.1 SPL Activities (Clements and Northrop, 2001). 13
2.2 Core asset development (Clements and Northrop, 2001). 13
2.3 Product development (Clements and Northrop, 2001). 14
2.4 SPL Engineering Framework proposed by Pohl *et al.* (2005). 15
2.5 Connector Envy smell depicted involving communication and facilitation services (a) and conversion service (b) (Garcia *et al.*, 2009). 21
2.6 Scattered Parasitic Functionality smell occurring across three components (Garcia *et al.*, 2009). 22
2.7 An Ambiguous Interface implemented using a single public method with a generic type as a parameter (Garcia *et al.*, 2009). 23
2.8 The Extraneous Adjacent Connector smell is shown. The connector SoftwareEventBus is accompanied by a direct method invocation between ComponentA and ComponentB (Garcia *et al.*, 2009). 24

3.1 Review process proposed by da Mota Silveira Neto *et al.* (2011). 28
3.2 Covered topics in PLA research. 29
3.3 Screening of papers. 36
3.4 Distribution of studies in publication years. 40
3.5 Overview of the area through a bubble plot. 41
3.6 Number of studies addressing research topics. 42
3.7 Artifacts affected by variability. 51
3.8 Architectural views addressed in the studies. 56
3.9 Evaluation methods addressed. 58

4.1 Feature Model for the Notepad SPL. 68
4.2 Architecture Recovery Process adapted from (Taylor *et al.*, 2009). . . . 73
4.3 Decomposition View of Notepad SPL. 73
4.4 Dependency View of Notepad SPL. 74
4.5 Notepad SPL Component Model. 76

5.1 Screenshot of the RescueMe app main screen. 85
5.2 Feature Model for the RescueMe SPL. 87
5.3 Deployment View for the RescueMe SPL. 89

5.4 Modules View for the RescueMe SPL. 90
5.5 Internal Dependencies Classes of RescueMe SPL. 92
5.6 Cluster Call Graph of RescueMe SPL. 93
5.7 Hierarchy Internal Dependencies of RescueMe SPL. 94
5.8 RescueMe SPL Component Model. 96

List of Tables

3.1 Search String 33
3.2 Research Type Facet 38
3.3 Top 11 contributing publication sources 39
3.4 Patterns in PLAs 44
3.5 How PLAs are documented in different studies 54

4.1 Notepad SPL Product Map. 71
4.2 Notepad SPL Variability Points. 75

5.1 RescueMe SPL Product Map 88
5.2 RescueMe SPL Variability Points 96
5.3 Comparative table considering the two objects of study. 103

A.1 Data extraction report 134
A.2 Primary Studies addressing Patterns 136
A.3 Primary Studies addressing Variability 137
A.4 Primary Studies addressing Documentation 138
A.5 Primary Studies addressing Evaluation 139

List of Acronyms

API Application Process Management

FM Feature Model

IDE Integrated Development Environment

MS Mapping Study

MVC Model-View-Controller

PLA Product Line Architecture

PM Product Map

RiSE Reuse in Software Engineering group

SOA Service-Oriented Architecture

SPL Software Product Line

XML Extensible Markup Language

1 Introduction

1.1 Motivation

Due to the critical dependence of society on software, increasing responsibility is attributed to the software engineering community. The effects of the failure of a system may spread beyond the system itself, because of the integration between systems that earlier were stand-alone. The result is a significant amount of scientific research that aims to either gather evidence or propose new ways to successfully plan and maintain those environments.

One way to handle the needs from these previous remarks while achieving faster development processes is through software reuse. According to Krueger (1992), software reuse consists in the process of creating software systems from existing software, rather than building them from scratch. In this sense, a number of approaches appeared in the software engineering scenario. One of such relevant approaches is **Software Product Lines** (SPLs) – an approach for exploiting variability, taking advantage of the common aspects of different software systems. Software product lines aim to attend to challenges through a set of core assets developed for reuse in the products that constitute the product line with up-front analysis, design, implementation, and so on (Clements and Northrop, 2001).

Since the development of an SPL involves the (often complex) implementation of different structures, processes, interfaces and activities, it is relevant for product line practitioners to pay sufficient attention to its architecture. The reuse of software artifacts is a key element in SPL practice, thus it is important to build a favorable environment that supports such practice, as well as a solid architecture related to it. The implementation of core assets – along with their uses – should obey a set of organization rules in order to successfully achieve faster time-to-market and efficient management goals.

According to Bosch (2000), the core element in successful SPLs is the software Product Line Architecture (PLA), which should maximize the benefits of the commonalities between the products in the family while providing sufficient variability for each family member.

Further, it is important to properly design and evaluate the PLA aiming at its adequate and consistent evolution. PLAs have a longer life span and should support a range of products, being responsible for their quality attributes (Etxeberria and Mendieta, 2005). The interactions of product quality attributes requirements may lead to architectural conflicts (Olumofin and Misic, 2007) and consequently interfere in the assessment of PLAs. In addition, organizational issues may affect the evaluation of PLAs, due to their larger scope and commonly high number of stakeholders involved.

In this sense, the aim of this dissertation is twofold. First, we review the current state-of-the-art of PLAs. Then, we discuss one way to evaluate PLAs with respect to lifecycle properties: attributes related to maintainability (e.g. understandability and testability).

1.2 Problem Statement

Despite the fact that the number of publications discussing PLA-related issues have grown significantly in the latest years (more details in Chapter 3), none aimed at gathering evidence for synthesizing knowledge in PLA research. For this purpose, we conducted a systematic mapping study, which aims at providing an overview of the research field through classifications.

From the systematic review, we found that it is valuable to further investigate proper means to achieve higher quality of PLAs. Thus, we aimed at discussing one way to improve the quality of PLAs through the identification of *architectural bad smells* (Lippert and Roock, 2006). The term refers to architectural decisions that negatively impact system lifecycle properties, such as understandability, testability, and extensibility. It was originally proposed for the assessment of single systems, thus, our goal is to characterize architectural bad smells in the context of PLAs. Two case studies focusing on lifecycle properties in the architecture were performed using two SPL research projects in different domains.

In summary, the goals of this dissertation can be stated as follows:

> *This work investigates (i) current literature within the topic of PLAs; and (ii) the problem of architectural bad smells in software product lines, which*

> *is approached through two empirical studies. Evidence gathered in these studies are presented to provide a better understanding of the phenomenon - initially proposed to single systems - in the context of software product line engineering.*

In order to achieve the afore mentioned goals, we performed two empirical studies from which the results were compared and discussed through a comparative analysis. First, a systematic review was undertaken to map out the available literature evidence in the field of PLA. Then, we performed two case studies considering research projects in different contexts: one in the text editor and the other in the mobile domain.

1.3 Related Work

The literature on SPL and architecture provides a large number of studies, regarding both general and specific issues. Amongst them, we have identified literature reviews that aimed at either broader scopes (domain design approaches) or different aspects in SPL engineering (e.g. Requirements). Further, a number of papers discussed means to assess PLAs and/or single systems architectures. The following studies were considered related for having similar ideas to this work. They are presented in two categories: literature reviews, and architecture evaluation.

1.3.1 Literature Reviews

Filho *et al.* (2008) performed a systematic review on domain design approaches. They studied methods to domain design that have been developed for or that can be applied to SPL, laying emphasis on activities, guidelines, views, and good practices adopted by the approaches. Moreover, Murugesupillai *et al.* (2011) undertook a preliminary study of approaches that aim at bridging SPL and Service-Oriented Architectures (SOAs). They provided a brief overview of recent studies, classification of approaches and type of research available in the area. Through a systematic mapping of the studies, they reasoned that the majority (61%) of the studies contributed with methodologies, and the main motivation factor for utilizing the SPL-SOA approach was to properly manage variability. For structuring the field of measures and quality attributes for SPLs, two studies (Montagud and Abrahão, 2009; Montagud *et al.*, 2012) identified the current lack of proper means for measuring several quality attributes, as well as the wide use of inadequade methods for validating the existing measures.

In the context of single systems, de Oliveira *et al.* (2010) performed a systematic review focused on reference models and reference architectures based on the service-oriented approach. The authors presented an overview about the application of such models, highlighting their uses and reporting that there is not a consensus about how to represent such architectures. In addition, systematic reviews were undertaken in the field of software Architecture Reconstruction (Ducasse and Pollet, 2009) and Evolution (Breivold *et al.*, 2012). Both studies revealed that the activity of evolving architectures has been widely discussed in the last years.

Existing approaches highlight the importance of utilizing different viewpoints and the need of economic and technical planning. However, to the best of our knowledge, no work has been conducted in the PLA field aiming at summarizing current knowledge through a systematic mapping study. This work attempts to fill this gap, while contributing to structure the field in addition to other systematic literature reviews focused on SPL disciplines: Scoping (Moraes *et al.*, 2009), Requirements (Alves *et al.*, 2010), and Testing (Engström and Runeson, 2011; da Mota Silveira Neto *et al.*, 2011).

1.3.2 Architecture Evaluation

The issues involving both architectural refactoring and assessment techniques have been extensively discussed over the last years. In the context of single systems, methods (e.g., ATAM (Kazman *et al.*, 2000)) aim at assessing the consequences of architectural decisions in terms of quality attributes requirements. The practice of evaluation often use metrics that provide means for measurements and calculations regarding the potential risk within a complex architecture, as the conformances to business drivers are also taken into consideration.

In this work, we are guided by the method presented by Garcia *et al.* (2009). The authors discuss the identification of four representative architectural smells, i.e., design attributes that negatively impact the system's maintainability. Instead of focusing on refactoring implementation artifacts, the work addresses design decisions that are not necessarily faulty or errant, but still present a negative impact in software quality. The introduced smells were proposed based on the experience of two large industrial systems and case studies in the literature. We selected this work to serve as a guidance to this dissertation because it proposes an interesting approach to evaluate architectures. The proposed evaluation method considers identifying smells in the architecture of single systems. Thus, with the present work, we intended to apply this approach in the context of PLAs.

A recent paper (Oliveira Junior *et al.*, 2013) reported on a novel method to systematically evaluate UML-based PLAs: the SystEM-PLA. For the evaluation, it considers metrics obtained directly from UML models to perform both qualitative and quantitative analysis. With the approach, architects are able to also perform a trade-off analysis to prioritize quality attributes. The ATAM (and its extensions) principles serve as guidelines to the initial phases of the method for identifying and defining business drivers. Despite the fact that this method also considers the class and components models of the PLA to be evaluated, there are a number of differences when compared to the smells approach. The main difference is related to the goals and the requirements to perform the evaluation. Since it is based in ATAM, it takes into consideration the specification of scenarios, thus requiring considerable knowledge on the business to which the PLA is implemented. On the other hand, the approach used in the present work considers the identification of smells, which in turn does not require a deep understanding of the business drivers to be undertaken.

To the best of our knowledge, no studies have been undertaken from the viewpoint of architectural smells in SPLs. However, several studies discussed approaches that are as well used for dealing with evaluation of PLAs. For instance, the scenario-based assessment methods SBAR (Scenario-Based Architecture Re-engineering) (Del Rosso, 2006) and SAAM (Software Architecture Analysis Method) (Kazman *et al.*, 1994; Lutz and Gannod, 2003; Olumofin and Misic, 2007; Silva de Oliveira and Rosa, 2010b) which takes into consideration the description of a software design in three perspectives: *functionality*, which refers to the features of the systems; *structure*, which contains a collection of components and interfaces; and *allocation*, which makes explicit the way the features are implemented.

The aforementioned evaluation methods are not specifically interested in architectural smells. Instead, they discuss software anti-patterns (Brown *et al.*, 1998), which take into consideration general concerns related to project management and process difficulties rather than design problems. Despite the range of work discussing the impact of *code smells* (Fowler, 1999) in software architecture (Arcoverde *et al.*, 2012; Macia *et al.*, 2012a,b), we argue that they differentiate from architecture smells in the abstraction level. While architecture smells are related to design problems, code smells are obtained through an evaluation of source code. That is, different stakeholders are able to assess the orchestration of features and products infrastructure as soon as in the design level, instead of having the code artifact as the starting point.

1.4 Out of Scope

The following topics are not considered in the scope of this dissertation:

- **Other disciplines of the SPL process.** In this dissertation, we focus on the architecture stage of development. Although the decisions made in other disciplines can represent direct influence in the architectural design of an SPL, we decided to only consider issues related to its structure and organization of components/modules/-subsystems (smaller parts);
- **Single systems.** Our investigation is guided by the properties contained in SPL projects, that is, software environments where variability is present, and there is a clear management of common and variable assets. We limit our review to PLAs and only consider SPLs in the case studies;
- **Architectural Erosion/Drift.** In this work, we do not discuss conformance violations (e.g., architectural drift Rosik *et al.* (2011) and erosion van Gurp and Bosch (2002)). Further, in the case studies, we do not consider the "intented" architecture, but always the implemented architecture which has been recovered from the project assets;
- **Novel methods to resolve architectural bad smells.** This work is concerned with the identification of smells. In addition, we discuss the kinds of impacts that such phenomena would bring to the quality of PLAs. We have not proposed new solutions to the smells.

1.5 Statement of the Contributions

As a result of the work presented in this dissertation, the following contributions can be highlighted:

- **An overview of the literature regarding SPL architectures.** We conducted a systematic mapping study presenting the current state-of-the-art regarding the PLA topic, aiming at obtaining a broad overview of the area.
- **Architectural especification and modeling of two different SPLs.** We recovered the architecture of two different SPLs through both manual and automated analysis of architecturally relevant information, obtained from the avaliable artifacts.

- **A characterization of the architectural bad smells phenomenon in the context of PLAs.** An exploratory case study was conducted in order to understand the effects of architectural bad smells to the lifecycle of SPL projects. We also proposed a new smell, specifically observed in PLAs.

- **A replication of the exploratory study in a different domain.** We replicated the exploratory study using a different SPL and compared results in order to draw conclusions.

- **A comparative analysis of the obtained results.** We discuss the outcomes of the empirical studies and contribute with knowledge regarding architectural bad smells in SPLs.

In addition to the contribution mentioned, a paper presenting part of the findings of this dissertation has been accepted for publication:

- Andrade, H. S., Almeida, E. S., and Crnkovic, I. Architectural Bad Smells in Software Product Lines: An Exploratory Study. In *3rd International Workshop on Variability in Software Architecture (VARSA), Sydney, Australia*. Proceedings of the 11th Working IEEE/IFIP Conference on Software Architecture (WICSA 2014). New York: ACM Press, 2014.

Also, we are currently submitting papers to report the remaining results.

1.6 Research Design

The first step of our work was to strengthen our knowledge regarding software product lines and software architecture. We carried out an informal literature review, which aided us in having an overview of the research field, considering the terminologies, common practices and general guidelines. Next, we performed the systematic mapping study to structure the existing knowledge, while also contributing to the field with a report. Such systematic review allowed us to categorize the available evidence on current PLA practices and also identify gaps and trends in research.

With the review, we identified a gap that no previous work had been conducted with respect to architectural bad smells in software product lines. Thus, we performed an exploratory study to characterize the problem in the context of PLAs, since those smells were initially proposed to single systems. Then, we conducted a replicated study in the

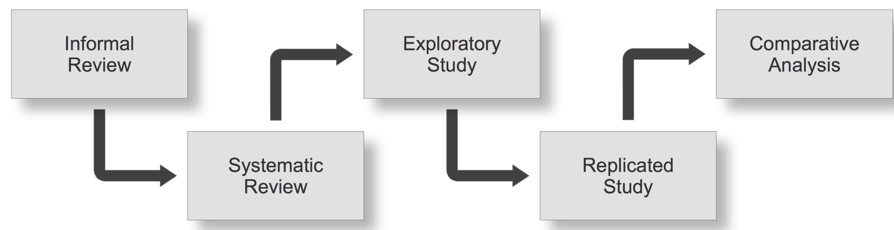


Figure 1.1 Workflow of the Research conducted in this work.

same conditions, but using an SPL in a different domain. With the replicated study, we aimed at investigating whether the occurrence (or effects) of different smells could be related to the domain in which the SPL was implemented. Finally, we discussed the differences and similarities of the results obtained from the two case studies through a comparative analysis. The workflow of the conducted research is shown in Figure 1.1.

1.7 Dissertation Structure

The remainder of this dissertation is organized as follows:

- **Chapter 2** presents a brief overview on the background of this work: fundamentals of software product lines, software architecture, and architectural bad smells;
- **Chapter 3** provides details on the planning and conduction phases, as well as the oucomes of the literature review performed;
- **Chapter 4** discusses the exploratory study used to characterize the architectural bad smells phenomenon in the context of SPLs;
- **Chapter 5** presents the replicated study using a SPL in a different domain, as well as the comparative analysis;
- **Chapter 6** provides a set of conclusions based on this work, discussing the limitations and directions for future work.
- **Appendix A** presents the list of journals and conferences manually searched in the mapping study. It also shows the data extraction report, and the lists of primary studies addressing different aspects in PLA;

- **Appendix B** provides the architecture specification of Notepad SPL: used in the exploratory study;
- **Appendix C** presents the architecture specification of RescueMe SPL in the MVC pattern: used in the replicated study;

2

An Overview on Software Product Lines, Software Architecture and Architectural Bad Smells

This chapter presents fundamental information for the understanding of three topics that are relevant to this work: software product lines, software architecture, and architectural bad smells. Section 2.1 discusses the motivation, benefits, and essential activities of SPL engineering. Section 2.2 presents the definitions, characteristics and terminologies of software architectures. Section 2.3 presents the definition, examples and impacts on quality attributes of architectural bad smells. Finally, Section 2.4 presents a summary of this chapter.

2.1 Software Product Lines

It is always desirable that software is developed with minimum time-to-market and maximum quality. Nowadays, all different kinds of business are highly dependent on software systems, thus the importance of delivering high quality software is inherent. Such systems are then required to excel with respect to both functional and non-functional requirements. On the other hand, a lower time-to-market is essential to keep business companies competitive. The different products to be developed are expected to be delivered as soon as possible without losing quality with respect to functionalities, maintainability, trustability, and so on.

In order to address the aforementioned concerns, one relevant approach appeared in the software engineering scenario. Software Product Line Engineering is a software development approach that focuses in reuse, combining concepts of platforms and mass

customization (Pohl *et al.*, 2005). The use of Software Product Lines (SPLs) aims to optimize the software development process, describing product families and exploring their variability and commonalities. In other words, the approach takes advantage of the common and variable aspects present in a range of products to be developed to systematically achieve large-scale reuse.

The initial costs for implementing an SPL are higher due to the time and investments required for design and implementation of core assets, as well as mechanisms that enable variability. In a reactive SPL approach, each product has its components analyzed in order to determine which features must be part of the domain of the product line (commonalities) and which features are singular on the application (variability). As more products are implemented over time, it becomes more valuable for one to utilize the SPL approach due to the ease of developing new applications from domain assets, and also because the maintenance effort costs are drastically reduced.

One of the most widely accepted and complete definitions of SPL is given by Clements and Northrop (2001) as "*a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*".

2.1.1 Benefits of SPL Engineering

Several benefits can be achieved from developing software using the SPL approach. For example, it allows companies to reduce costs through the development of core assets that will be reused across several products. Although the initial costs of developing an SPL are higher when compared to single systems development, because of the development platform, the costs are soon reduced. It is estimated that from the third system developed, the cost of developing each new product is considerably lower than if it were produced in the classical individual manner (all new requirements, all new scope, all new implementation, and so on) (Pohl *et al.*, 2005).

Further, the quality of the products derived are increased due to the several testing phases through which the core asset are submitted. The core assets and their functionalities are reviewed and tested in many products and under several circumstances.

Some other benefits of SPL engineering can be achieved, such as the ones described in the following topics (Clements and McGregor, 2012):

- **SPL engineering fuels innovation.** Since the products are developed in a prescribed way, companies no longer have to work long hours to meet the next

product's delivery deadlines. Engineers then have time to think about new features their customers would find valuable, as well as to innovate inside their own organization;

- **SPL engineering increases agility.** The products to be developed from the SPL are planned ahead and built from the core assets. The scope is explicitly defined using market predictions and technology forecasts, thus giving the product line built-in agility across the development of all products;

- **SPL engineering supports strategic product planning and portfolio optimization.** It is suggested that market share and profitability increase with broader product lines (Kekre and Srinivasan, 1990). This approach allows the company to carefully select product opportunities based on market considerations and the scope of their production capability;

- **SPL engineering supports mass customization and a customer-centric strategy.** In SPL engineering, mass customization is supported commonality/variability analysis, the effective reuse of assets, and variation mechanisms that support configuration at assorted times in the product life cycle;

- **SPL engineering invigorates and empowers the workforce.** In organizations that employ SPL engineering, developers who are entitled to general-purpose solutions and developers who work on products-specific requirements are both highly valued and useful.

2.1.2 SPL Engineering Essential Activities

When implementing an SPL, software platforms are used. Software platforms are software subsystem sets and interfaces that form a common structure from which a number of derived products can be developed and efficiently produced (Meyer and Lehnerd, 1997). The platform consists in the assets developed based on the product family, which are going to be reused for developing other products.

Clements and Northrop (2001) proposed a set of activities to create a software product line. The approach considers two stages of development: *domain engineering* and *application engineering*, which are realized through a set of three essential activities, as shown in Figure 2.1.

The figure illustrates three rotating circles, each of them representing one essential activity. The circles are linked together and in perpetual motion, showing that these three

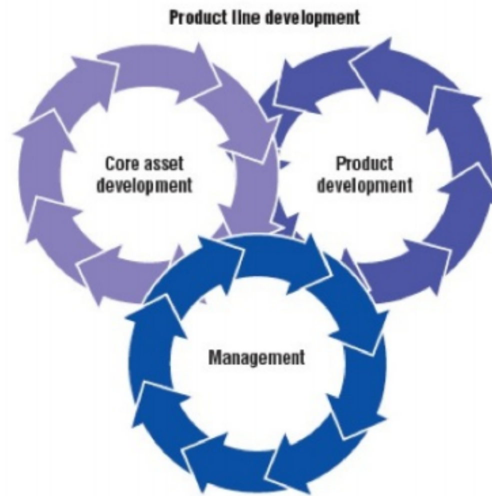


Figure 2.1 SPL Activities (Clements and Northrop, 2001).

activities are essential, inextricably linked, can occur in any order, and are highly iterative (Clements and Northrop, 2001).

The *core asset development* activity establishes a production capability for products. This is where business rules and some solution design decisions are taken into consideration, such as: product and production constraints, architectural styles, design patterns, application frameworks, production strategy and legacy systems. As a result from this process, the core assets are produced, and the SPL scope is defined. The outputs are entitled to provide means for development and details of the products that will be developed in the next phase. The core asset development activity is depicted in Figure 2.2.

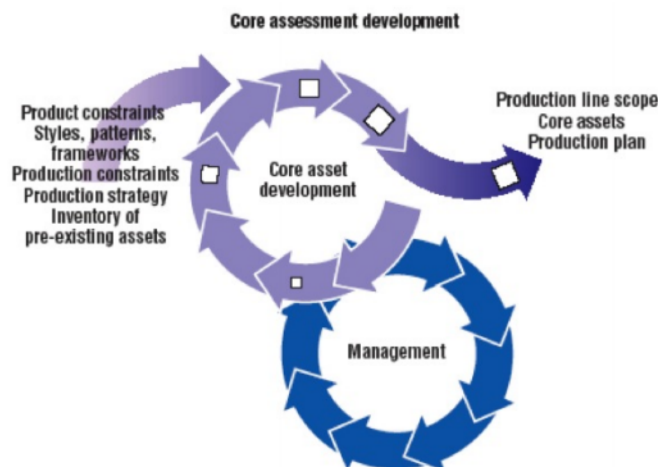


Figure 2.2 Core asset development (Clements and Northrop, 2001).

Next, the *product development* activity is responsible for building the products in a

SPL. The artifacts produced in the previous phase are considered to build the products according to the requirements specification for individual products. These requirements represent a variation of the generic product description contained in the SPL scope. The outputs of this phase are the products. The product development activity is depicted in Figure 2.3.

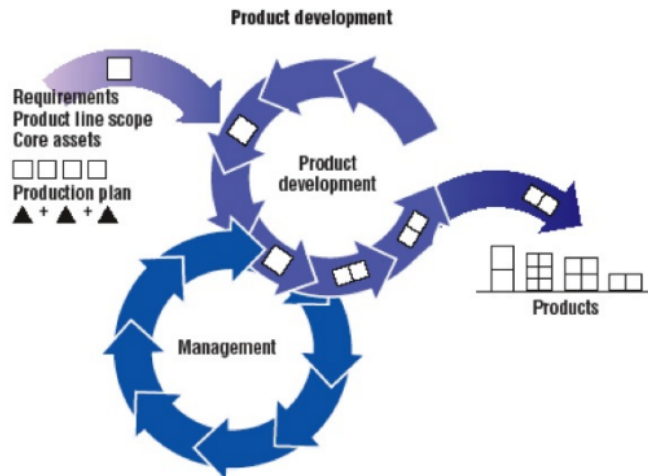


Figure 2.3 Product development (Clements and Northrop, 2001).

In the meantime, the *management* activity is responsible for performing the administration of both technical and organizational issues. The technical management ensures the definition of the processes for the SPL, the engagement of the groups in the required activities, and the progress tracking. The organizational management is entitled to optimize the organizational structure and ensure that the organizational units receive the right resources in sufficient amounts.

Another approach to perform SPL engineering is proposed by Pohl *et al.* (2005). It also takes into account domain engineering and product engineering phases, as depicted in Figure 2.4.

The framework defines the activities in developing applications using SPL, explicitly describing the processes in domain engineering and application engineering. During the domain engineering phase, the SPL properties, management solutions, applications commonalities and variation points are defined. This is when the scope of the SPL is defined, and the commonalities and variability of all products are explored. In the domain engineering phase, the core assets are developed: that is, pieces of software that will be reused among a number of products throughout the realization of the SPL. Gathering reusable software components (assets) is the main goal of this stage.

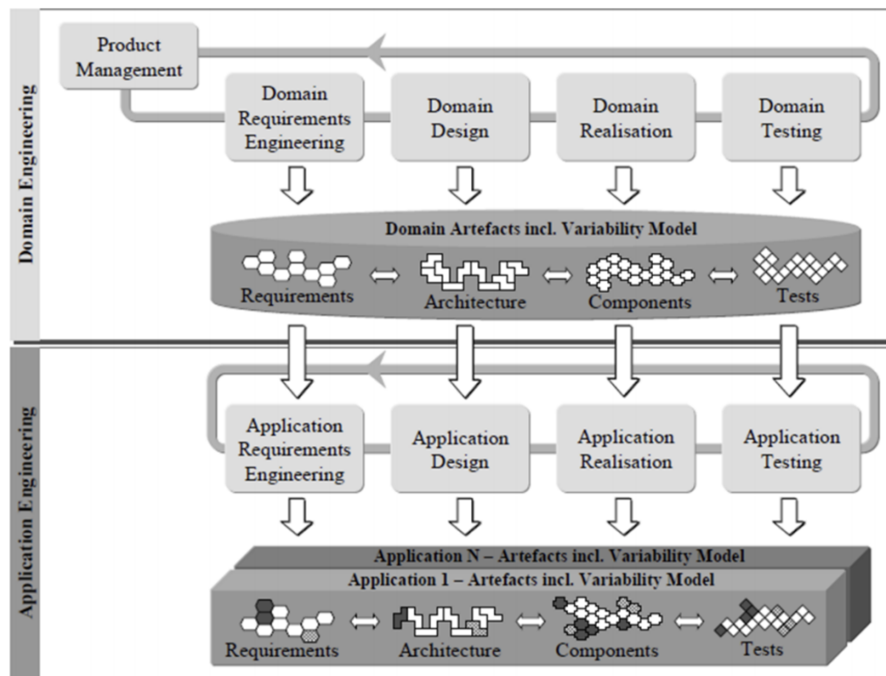


Figure 2.4 SPL Engineering Framework proposed by Pohl *et al.* (2005).

On the other hand, in the application engineering phase, the applications are developed and customized. The assets defined are implemented, and the particular functionality of the applications are incorporated into the assets to form the final product, aiming to satisfy the needs of a customer. By the end of the domain and application engineering phases, the artifacts related to requirements, architecture, components and tests should be produced.

2.2 Software Architecture

Architecture has emerged as a crucial part of software development. Through the architecture, the business rules are expressed, and the requirements are projected to be satisfied. If we consider software development as a process consisting of two stages - the problem space and the solution space - the architectural design of the system would be the first step in the solution space. In this stage, the structures of the system are defined through an abstract description. It does not consider details of implementation or algorithms, but rather addresses the interactions and the behavior of components.

One well-known definition to the term *architecture* is that it refers to the structure of a system, consisting of software elements, externally visible properties, and the

relationships among elements (Bass *et al.*, 2003a). Another explanation of the concept is provided by the Software Engineering Institute (SEI), in which the architecture of a software program is defined as "a depiction of the system that aids in the understanding of how the system will behave". Even further, the software architecture of a system can also be stated as "the set of principal design decisions made during its development and any subsequent evolution", being the stakeholders responsible for determining which aspects are considered 'principal' (Taylor *et al.*, 2009).

In fact, a widely accepted consensus on the definition of software architecture does not seem to exist (Kruchten *et al.*, 2006). In fact, several other definitions can be found in literature. Thus, researchers and practitioners agree that an explicit and clear definition of what architecture is, to the context in which the term is used, is essential to maintain the consistency of its semantics.

In this work, we maintain the definition given by Taylor *et al.* (2009), who stated that "software architecture is the set of principal design decisions governing a system". The stakeholders determine which aspects are deemed to be principal. These decisions usually include how the system is organized into subsystems and components, how functionality is allocated to components, and how components interact with each other and their execution environment.

2.2.1 The Architecture of a Software Product Line

The aforementioned diversity of concepts is extended to the definition of PLA, thus the key concepts are required to be explicitly stated in order to define the scope of a study. According to Gomaa (2004), "a product line architecture is an architecture for a family of products, which describes the kernel, optional, and variable components in the SPL, and their interconnections". Moreover, the PLA can also be defined as the "core architecture that captures the high level design for the products of the SPL, including the variation points and variants documented in the variability model (or feature model)" (Pohl *et al.*, 2005).

The architecture is a fundamental element in SPL engineering, because it is a key artifact for structuring and managing the SPL. One difference between the design of an architecture for a SPL and an architecture for an individual product is that the first requires product-specific features to be considered (Bosch, 2000). The PLA should handle the diversity of contexts, which may be present in different products, in terms of underlying hardware, communication to external products, and user interface. Common issues that SPL architects need to address is the decision whether product-specific aspects

of products contexts must be addressed by the SPL architecture, or whether these aspects can be added modularly to a product architecture.

We consider the PLA to be a key aspect in SPL engineering, through which the complexity of a variability-based environment can be managed. It is used as a platform to a range of products, being responsible for describing the common and variable aspects between them. In this sense, the design decisions in a PLA are expected to support different sets of requirements (i.e., features) for products to be instantiated. During the Domain Engineering phase, the common parts of the PLA are defined. The architecture of an individual product is composed during Application Engineering, by combining the common parts of the architecture with the variable aspects that must be present in that specific product.

2.2.2 Terminologies

Different terminologies referring to architecture in SPL have been used, such as *domain architecture*, *platform architecture* and *configuration architecture*. In many studies, such terminologies carry similar meanings, and describe the processes and artifacts aforementioned in this section. For these cases, we standardized the term product line architecture (PLA).

Moreover, several recent discussions address the definitions of *PLAs* and *reference architectures*. While a reference architecture covers the knowledge in a broader domain, a PLA is more specialized and focuses on a specific set of software systems in a narrower domain (Nakagawa *et al.*, 2011; Angelov *et al.*, 2012; Galster *et al.*, 2013). Nevertheless, there seems to be a misconception in publications around the differences regarding these concepts. Since we performed a literature review that aims at helping in the understanding of the field, studies using both terminologies were considered because they are many times used interchangeably.

2.3 Architectural Bad Smells

Not rarely, software systems are subject to changes. As lower time-to-market is one of the most important aspects in today's competitive business, a lot of effort is put into optimizing the development processes to meet new requirements. Instead of building new software from scratch, the practice of evolving and adapting existing systems is common. In the context of SPLs, for example, it is not uncommon that the development platform is maintained and upgraded for several years. In this sense, the evolution and adaptation

of software systems are required to be carefully managed and executed. The refactoring should be consistent to attend to new requirements without representing a negative impact to other qualities previously achieved.

One very popular way to determine how to refactor systems is through the identification of *code bad smells* (Fowler, 1999). The term refers to implementation structures that negatively affect system lifecycle properties. In other words, the concept of code smells is often used to indicate properties that are not necessarily faulty or errant, but present negative effect in a system's maintainability. For example, in an object-oriented system one entity can be designated as *God Class* if it assumes too much responsibility when compared to the remaining classes. This implementation decision can make it difficult to maintain and evolve such class, due to its increased relationship level with the other entities. Other examples of code smells can be mentioned, such as *Feature Envy*, which means a case where one method is too interested in other classes, and *Code Clones*, which represents duplicated code.

Nevertheless, those smells are restricted to implementation level constructs, such as methods, classes, statements and parameters. When poor structuring of architecture-level constructs (i.e., components, connectors and interfaces) causes a reduction in the system maintainability, these properties are called architectural bad smells.

2.3.1 Definition

The term *architectural bad smell* was originally used by Lippert and Roock (2006), for describing an indication of an underlying problem that occurs at a higher level of a system's abstraction than a code smell. Architectural smells are structural attributes that mainly affect lifecycle properties - such as understandability, testability and reusability - but can also affect quality properties, such as performance and reliability.

In other words, architectural smells can be defined as combinations of architectural constructs that induce reductions in system maintainability. These smells are identified in consideration to architecture-level abstractions, such as components, connectors and styles, rather than implementation constructs.

According to the discussion in (Garcia *et al.*, 2009), such a phenomenon may be caused by:

- applying a design solution in an inappropriate context;
- mixing combinations of design abstractions that have undesirable emergent behaviors; or

- applying design abstractions at the wrong level of granularity.

The occurrence of smells in a design may represent a justification in different concerns. However, we argue that the trade-offs should be assessed to also allow means to adequately maintain the system at architecture level. Software architects should evaluate whether actions to change the identified properties will result in the actual benefits when accounting the change impact as a whole. In the context of SPLs, the importance of properly evaluating such impact is increased, due to the key role of the PLAs and the often widespread effect of changes in design.

Architectural smells are remedied by changing the structure and the behaviors of the internal system elements without changing the external behavior of the system. That is, products within the scope of the SPL should not be affected in terms of functionality by architectural changes. Further, the PLA is still required to support the derivation of those products.

In this work, we do not address the issues related to differences between the intended and implemented architectures. Since the intended architecture is often outdated and/or poorly specified, we focus the analysis on the actual architecture. The disadvantage of such decision is that a recovery process is required prior to the identification of smells. On the other hand, the implemented architecture always exists, and represents actual constructs with respect to the system organization.

Further, we consider human organizations and processes to be orthogonal to the definition and impact of a specific architectural smell. In other words, the identification and correction of architectural smells are not dependant on an understanding of the history of the analyzed system. An independent analyst should be able to indicate possible smells without knowing details about the development organization, management or processes.

2.3.2 Representative Smells

This section describes the four architectural smells proposed in literature in the context of single systems. Prior to the identification of smells, the architecture is assumed to be defined in terms of Components, Connectors, Interfaces and Configurations. Those aspects are considered because they are widely accepted and can effectively inform the structure of a system through their definition.

The concepts used to scope each of these aspects defined by Taylor *et al.* (2009) and described as follows:

- **Component**: An architectural entity that (1) encapsulates a subset of the system's

functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context. A component can be as simple as a single operation or as complex as an entire system, depending on the architecture, the perspective taken by the designers, and the needs of the given system. The key aspect of any component is that it can be "seen" by its users, whether human or software, from the outside only, and only via the interface it has chosen to make public. Otherwise, it appears as a "black box".

- **Connector**: An architectural element tasked with effecting and regulating interactions among components. The simplest and most widely used type of connector is procedure call. Procedure calls are directly implemented in programming languages, where they typically enable synchronous exchange of data and control between pairs of components: The invoking component (the caller) passes the thread of control, as well as data in the form of invocation parameters, to the invoked component (the callee); after it completes the requested operation, the callee returns the control, as well as any results of the operation, to the caller.
- **Interface**: The point at which components and connectors interact with the outside world – in general, other components and connectors.

Next, we present the definition of the representative smells proposed in (Garcia *et al.*, 2009). In order to properly illustrate the smells, we show graphical examples of each of them considering UML specifications, as well as what type of impact the smells may cause to the overall system's maintainability. The diagrams shown in this section can be used to guide future inspections in different projects.

Connector Envy

Basically, components with Connector Envy cover too much functionality with regard to connections. Instead of having the interaction facilities delegated to a connector, the components encompass, to a great extent, one or more of the following types of interaction services:

- *Communication:* concerns the transfer of data between architectural elements. “Data” in this context is described as messages, computational results, etc;

- *Coordination:* concerns the transfer of control between architectural elements. “Control” in this context is described as, for example, the passing of a thread execution;

- *Conversion:* concerns with the translation of differing interaction services between architectural elements. “Services” in this context means data formats, types, protocols, etc;

- *Facilitation:* concerns the mediation, optimization, and streamlining of interaction. For example: load balancing, monitoring and fault tolerance.

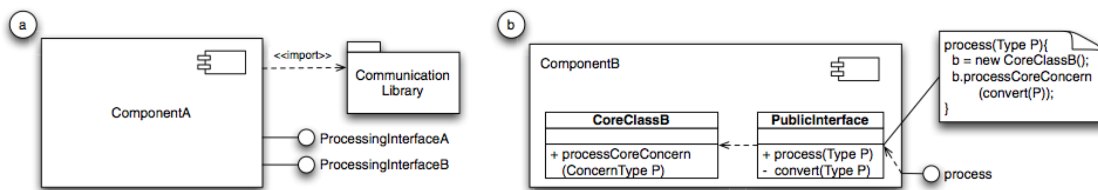


Figure 2.5 Connector Envy smell depicted involving communication and facilitation services (a) and conversion service (b) (Garcia *et al.*, 2009).

Figure 2.5 (a) shows an occurrence of Connector Envy smell, where *ComponentA* implements communication and facilitation services. *ComponentA* imports a communication library, which implies that it manages the low-level networking facilities used to implement remote communication.

Figure 2.5 (b) shows another Connector Envy smell, where *ComponentB* performs a conversion as part of its processing. The interface of *ComponentB* called *process* is implemented by the *PublicInterface* class of *ComponentB*. *PublicInterface* implements its process method by calling a conversion method that transforms a parameter of type *Type* into a *ConcernType*.

The quality attributes affected with the occurrence of this smell are:

- **Reusability:** dependencies are created between interaction services and application-specific services, making it difficult to reuse either type of service without including the other;

- **Understandability:** disparate concerns are commingled - the component carries functionality and connection responsibilities;

- **Testability:** application functionality and interaction functionality cannot be separately tested.

The occurrence of Connector Envy smell may be acceptable when performance is of higher priority than maintainability. Creating two separate entities for functionality and interaction may demand an extra level of indirection. However, the maintainability implications related to this smell can cause a cumulative effect, as multiple incompatible connector types are placed within multiple components that are used in the same system.

Scattered Parasitic Functionality

This smell is characterized by a system where multiple components are responsible for realizing the same high-level concern. Also, some of those components are responsible for orthogonal concerns. When this smell occurs, a single concern is established across multiple components, and at least one component addresses multiple orthogonal concerns. The Scattered Parasitic Functionality smell may be caused by crosscutting concerns that are not addressed properly.

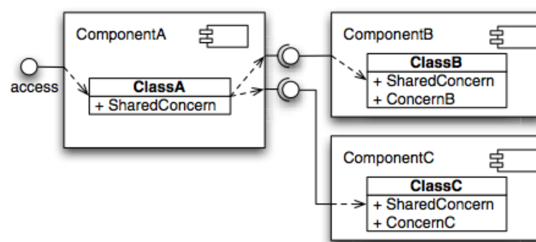


Figure 2.6 Scattered Parasitic Functionality smell occurring across three components (Garcia *et al.*, 2009).

Figure 2.6 shows an occurrence of Scattered Parasitic Functionality smell, where a concern is shared across three different components. The example also shows that each component contains its own concerns.

The quality attributes affected with the occurrence of this smell are:

- **Modifiability:** when a functionality related to a concern needs to be modified, several different components may be updated;
- **Understandability:** components affected with a scattered concern also contain their own orthogonal concerns;
- **Testability:** tracking and properly testing all functions related to a concern may be difficult depending on the complexity of the system;
- **Reusability:** reusing a component with a scattered concern requires all other related components to also be used.

The occurrence of Scattered Parasitic Functionality smell may be acceptable when, for example, the shared concern needs to be provided by multiple off-the-shelf (OTS) components whose internals are not available for modification.

Ambiguous Interfaces

Ambiguous Interfaces offer only a single, general entry-point into a component. This smell appears in systems where componentes use general types, such as strings or integers to perform dynamic dispatch. Another scenario is in event-based publish-subscribe systems, in which interactions are not explicitly modeled and multiple components exchange event messages via a shared event bus. Ambiguous Interfaces differ from function pointers and polymorphism in reducing static analyzability, because they are realized at the architectural level. Thus, they can occur independently of the implementation-level constructs.

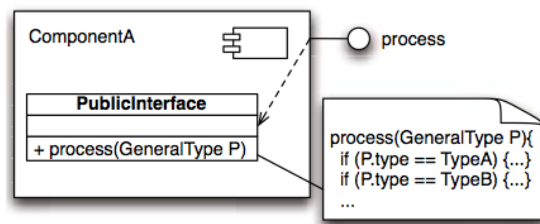


Figure 2.7 An Ambiguous Interface implemented using a single public method with a generic type as a parameter (Garcia *et al.*, 2009).

Figure 2.7 shows an occurrence of Ambiguous Interface smell, where two aspects are relevant. First, the interface offers only one public service or method, even though its component offers and processes multiple services. The component accepts all invocation requests through this single entry-point and internally dispatches to other services or methods. Second, since the interface only offers one entry-point, the accepted type is consequently overly general. In other words, the component claims to handle more types of parameters than it will actually process, by accepting the parameter *P* of generic type *GeneralType*.

The quality attributes affected with the occurrence of this smell are:

- **Analyzability:** an Ambiguous Interface does not reveal which services a component is offering. Thus, in order for its services to be used, the related component's implementation must be inspected;

- **Understandability:** it is not clear exactly what type of message is being passed through the interface. For instance, in event-based systems, the components involved become dependable, and the services invoked may not be recoverable if the interface only contains one entry point to multiple services.

Extraneous Adjacent Connector

This smell occurs when two connectors of different types are used to link a pair of components. In the context in which this smell is proposed, only two types of connectors are considered: procedure call and event connectors, although this smell applies to other connector types as well.

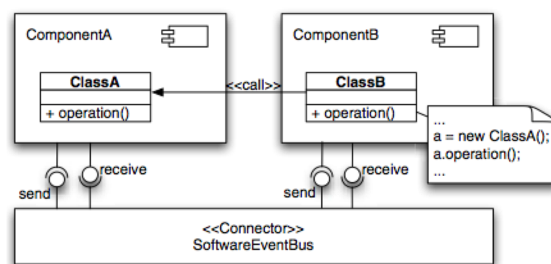


Figure 2.8 The Extraneous Adjacent Connector smell is shown. The connector SoftwareEventBus is accompanied by a direct method invocation between ComponentA and ComponentB (Garcia *et al.*, 2009).

Figure 2.8 shows an occurrence of Extraneous Adjacent Connector smell, where an event-based communication model is established through an event bus. The events are transmitted asynchronously and possibly anonymously. The passing of events is managed by the bus, while procedure calls transfer data and control through the direct invocation of a service interface provided by a component. In the figure, *ClassB* in *ComponentB* communicates with *ComponentA* using a direct method call.

The occurrence of Extraneous Adjacent Connector is peculiar because along with the decision of deploying different connectors, different quality attributes may affect each other. For example, procedure calls have a positive affect on understandability, since direct method invocations make the transfer of control explicit and, as a result, control dependencies became easily traceable. On the other hand, event connectors increase reusability and adaptability because senders and receivers of events are usually unaware of each other and, therefore, can more easily be replaced or updated. However, having two architectural elements that communicate over different connector types in parallel represents the danger that the beneficial effects of each individual connector may cancel

each other out.

The quality attributes affected with the occurrence of this smell are:

- **Adaptability:** in this condition, senders and receivers of events are unaware of each other. However, if there is a procedure call, the involved components may be difficult to adapt;
- **Understandability:** using an additional event-based connector reduces understandability because it is unclear whether and under what circumstances additional communication occurs between the two components;
- **Reusability:** having two different links between components, thus having a strong dependency on each other, make them difficult to be reused in different contexts.

The occurrence of Extraneous Adjacent Connector smell may be acceptable when, for example, a standalone desktop application uses both connector types to handle user input via a GUI. In such case, event connectors are not used for adaptability benefits, but to enable asynchronous handling of GUI events from the user.

2.4 Chapter Summary

In this chapter, we presented an overview concerning the main topics of this work: Software Product Lines, Software Architecture and Architectural Bad Smells. We described the benefits achieved when using the SPL approach, such as lower time-to-market and higher quality of products. We discussed the approaches and activities that can be performed to realize SPL.

Further, we presented definitions to the term *software architecture*, and explained the differences between the architecture of single systems and the architecture of an SPL.

Then, we presented the definition of architectural bad smells, described the representative smells, and presented their impact to the architecture's lifecycle properties.

Next chapter presents a mapping study that was performed to investigate SPL and architecture concerns. The goal of the review was to understand the current knowledge in the field of PLA, identifying gaps and trends, and providing guidance for future research.

3

A Systematic Mapping Study on Software Product Lines Architecture

As mentioned in the last chapter, software architecture represents a key aspect in SPL engineering. Through the PLA, the business processes are modeled and the solution is projected to satisfy the requirements of every product to be derived. The goal of the mapping study presented in this chapter is to categorize the current knowledge in the field of PLAs to identify gaps, trends, and contribute to future research.

This chapter is organized as follows: Section 3.1 presents the reasons why this mapping study is performed. Section 3.2 presents the review method and how it differs from classic systematic literature reviews. Section 3.3 describes the process followed to perform this review. Section 3.4 presents information related to the planning of the review, including the topics covered, research questions, search strategy, the search string and the studies selection criteria that was adopted. Section 3.5 describes the conduction process, including the data sources, conduction workflow, screening of papers, classification scheme, and details on the data extraction phase. Section 3.6 presents an overview of the included studies, as well as the findings of the review: the distribution of studies according to the aforementioned topics of interest, and the answers to the research questions. Section 3.7 presents a discussion regarding the findings related to those topics. Section 3.8 presents the threats to the validity of this review. Finally, Section 3.9 presents a summary of this chapter.

3.1 Motivation

Many studies reported solutions regarding different aspects of design and using different methods, but none aimed at gathering relevant information for synthesizing knowledge

in PLA research. In this sense, the purpose of the work reported in this chapter is to select and review the current literature publications in a systematic way, providing a categorization of the existing studies discussing the following architecture aspects of SPLs: patterns, variability management, documentation and evaluation. We chose these topics due to both a previous informal research and by discussing with research experts in the field. By presenting an up-to-date state of research, we help both practitioners in understanding the phenomenon and researchers in identifying gaps, trends and current challenges in this field.

3.2 Review Method

Given the broadness of the research area and the intention to provide an overview of the publications related to PLA, we aimed at systematically reviewing the literature through a method that resulted in broad coverage, instead of a narrow focused analysis. Our purpose with this work is to contribute by indicating the quantity of evidence in the field, identifying research trends and gaps, and categorizing studies.

In this sense, we performed a Systematic Mapping Study (MS) based on the guidelines proposed by Petersen *et al.* (2008). This type of literature review aims at systematically and objectively examining the extent and range of research activity of an area. A MS summarizes and categorizes information extracted from studies retrieved from different sources based on a set of inclusion and exclusion criteria.

Mapping Studies do not usually answer specific questions, as they are more concerned with the classification of studies and identification of research gaps (Budgen *et al.*, 2008; Kitchenham and Charters, 2007). They differ from Systematic Reviews in their broadness and depth. Instead of rigorously searching, analyzing and assessing studies, we gather relevant information in order to draw conclusions regarding the current state-of-the-art of PLA research. In many cases, a MS is performed prior to executing a full Systematic Review for identifying the value of such effort.

In summary, Mapping Studies:

- ask multiple (and often broad) research questions;
- are more concerned with a broad focus instead of a narrow focus;
- are likely to return a very large number of studies;

- are unlikely to include in depth analysis techniques such as meta-analysis and narrative synthesis; and
- aim at influencing the future direction of primary research.

It is important to notice that identifying gaps in literature through a MS will not necessarily identify gaps in study reviews, since the focus of a MS is not to consider and evaluate the quality of the studies. Our goal is to summarize evidence obtained from acknowledged sources regardless of study design.

In this work, we considered some concepts of Systematic Reviews, such as the protocol definition, for a better planning and formalization of the process.

3.3 Review Process

da Mota Silveira Neto *et al.* (2011) proposed an adaptation of the workflow reported in (Petersen *et al.*, 2008) by including the definition of a protocol. Even though the existence of a protocol is not mandatory, we decided to include this artifact because through its establishment, researchers are able to document the research directives, strategies and annotate decisions for calibrating the mapping study process. The protocol contains detailed control information on the search terms, search strategy, expectations and criteria for selecting studies.

The review process, from planning to reporting, was carried out in 11 months by 3 researchers in software engineering: one master student, one PhD student, and one professor with expertise in SPL and architecture. All participants had experience in SPL projects in both industry and academia. Figure 3.1 presents the performed steps for running the review.

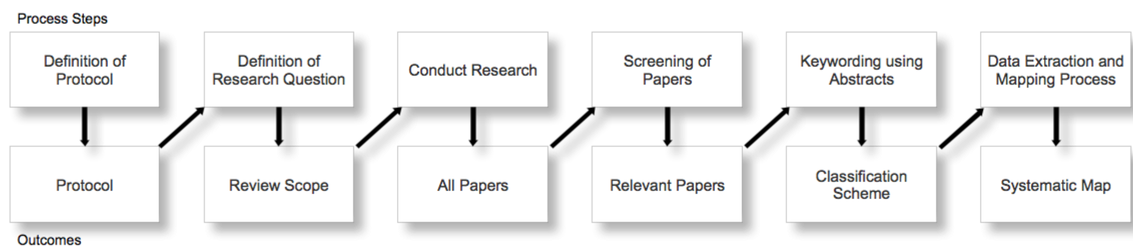


Figure 3.1 Review process proposed by da Mota Silveira Neto *et al.* (2011).

3.4 Planning

As previously stated, the focus of this work is to summarize evidence and research issues in PLA. In other words, we would like to obtain an understanding of the area through the main question: "***How is architecture dealt with in SPL?***".

Prior to defining the research protocol, we performed an informal literature review in order to better understand the terminology, main practices and challenges of the area. Based on the review, we proposed a set of subareas that were then validated through meetings with experts, who provided their opinion and contributed to improve the scope of this study. The subareas are described separately in Figure 3.2, although several studies address more than one topic simultaneously.

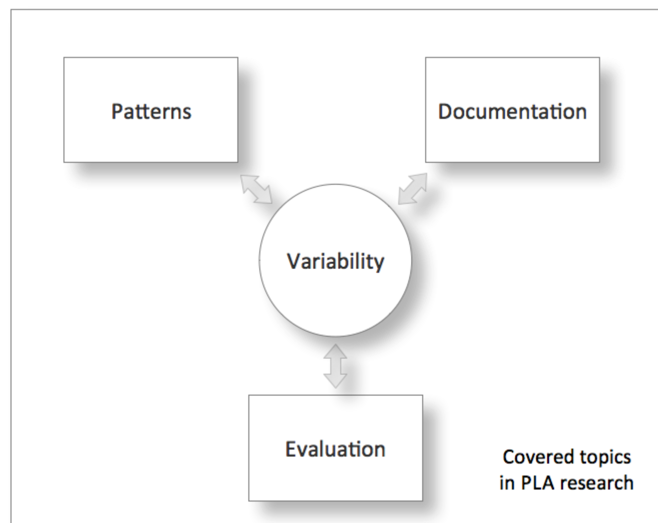


Figure 3.2 Covered topics in PLA research.

Any practice towards structuring and formalizing architecture design in SPL must consider variability since the notion of variable (and common) aspects of products plays a fundamental role throughout every SPL discipline. For instance, the choice of using a design pattern should also consider that SPLs are variability-based environments, which means that certain characteristics of a pattern could be harmed or beneficed by the need to orchestrate different features to be composed and derive products. Also, the practices to evaluate a PLA, for example, should take into consideration that different features and different compositions are supported by this architecture. Such scenarios make explicit the difference between the architecture of a single system and a PLA by a key aspect: variability, i.e., the requirement to implement different products from the same architecture.

3.4.1 Research Questions

Four research questions were derived as an attempt to cover the following topics of interest within PLA: *Patterns*, *Variability (handling)*, *Documentation*, and *Evaluation*. These topics were selected after an informal review, several meetings with experts both from industry and academia, and also as an attempt to cover the gaps discussed in the related work section. We then define the overall goal of the study, according to the Goal-Question-Metric (GQM) template, (Zhang *et al.*, 2011) as: "Analyzing and characterizing the current situation of Product Line Architecture patterns, variability handling, documentation, and evaluation with respect to SPLs architectural information from the perspective of researchers and practitioners as the basis for gathering relevant evidence to make research work more efficient and effective".

The following questions and their rationale were structured following the steps suggested by Easterbrook *et al.* (2008). They take into consideration an exploratory fashion that allows us to obtain rich qualitative data. By asking existence and descriptive-comparative questions, we provide the reader with a clear understanding of the phenomena, as well as more precise definitions of theoretical terms.

Q1. Are architectural patterns (styles) used in SPL? With this question we intend to find out whether common architectural solutions are used in SPL. For instance, we investigate if patterns commonly used in single system development - such as pipes and filters - are also part of the architecture definition of SPLs (van der Linden *et al.*, 2007). Moreover, we aim at discussing the implications of applying particular patterns in quality attributes.

Q2. How is variability handled in the architecture level of SPLs? It is known that variability is a key aspect in software product line engineering (Pohl *et al.*, 2005). Through this question, we intend to understand how variability is present in the architecture design level. We investigate how variability is represented, i.e., how variable requirements are considered. Also, we are interested in the concepts used, and the artifacts involved when representing variability, since defining proper aspects of variability are of high importance in SPL engineering.

Q3. How are the SPL architectures documented? Based on (Pohl *et al.*, 2005), there are different ways to document the architecture of a product line. First, it is necessary to decide what information to document, and then build an architecture through guidelines,

so others can successfully implement, use and maintain systems from it (Clements *et al.*, 2010). With this question, we aim to find out whether PLAs are documented at all. In addition, we are interested in the techniques used for representing different aspects of architecture, for example, views and architectural knowledge. This questions differs from the previous one in purpose. While RQ2 focuses on architectural artifacts and how they specifically express variability, this questions focuses on different views, ADLs and frameworks used to document PLAs as a whole.

Q4. How are the SPL architectures evaluated? Through this question we intend to identify the strategies to evaluate the design of a PLA. In addition, we question whether metrics or tools are used, or all validation procedures are based on expertise and subjectivity. The goal is to investigate how to measure systems properties based on an abstract specification, in this case, an architectural design (Bosch, 2000).

3.4.2 Viewpoints

The search string reflects the construction of the research questions, which were structured from three viewpoints:

- **Population:** published scientific literature about architecture of software product lines or architecture-related software product family approaches;
- **Intervention:** approaches or issues about methods, values, principles or practices involving architectures of software product lines or software product families.
- **Outcomes:** results involving particular architectural models and methods for designing software product line architectures.

3.4.3 Search Strategy

The search strategy used to construct the search terms follows the approaches presented by Easterbrook *et al.* (2008) and Kitchenham and Charters (2007) because they are systematized in essence. The defined steps were used to derive the search string from the questions and their viewpoints, as well as through the opinion of experts and information in relevant papers.

Search String: From the identified topics of research, we structured the search string by also considering for each term the synonyms frequently used by the community. The idea was to obtain targeted studies through matching the combination of software product lines (or synonyms), architecture (or synonyms) and at least one of the keywords that represent the subtopics of interest within PLA.

After a number of discussions towards an agreement, we ran a set of pilot searches in the digital libraries. In order to calibrate the search string, we established a "quasi-gold" standard (QGS), as proposed in (Zhang *et al.*, 2011). The results from the manual search were defined as a QGS by crosschecking the results obtained from the automatic search. If the QGS were not found by the automatic search, we included more terms into the search string and re-executed the automatic search.

The search process was validated with experts in both the mapping process and the field of PLA. The search string is presented in Table 3.1. Its terms were linked using Boolean OR and AND operators.

Selection Criteria: In order to identify the relevant primary studies of architecture and software product line engineering approaches, the following inclusion/exclusion criteria were defined.

- **Inclusion criteria:** Studies that explore issues related to patterns, variability, documentation, and/or evaluation in PLA were considered. In addition, we consider unique studies, i.e., when a study has been published in more than one venue, the most complete version was used. We consider full papers published in conferences, journals and workshops published up to (and including) 2013, all written in English.

- **Exclusion criteria:** Studies that do not address architecture in SPL were excluded. Moreover, primary studies or studies that mention architecture in SPL, but do not discuss any type of method, activity, experience, or approach concerning at least one of the topics of interest in this mapping study were also excluded. We do not consider feature models as part of PLA. Studies that were only available as abstracts, PowerPoint presentations, tutorials, panels or demonstrations were also excluded from the process. Finally, short papers (with three pages or less) and studies that were not published in the period between 2000 and 2013 were excluded. We decided to consider the year 2000 as a starting point due to the release of the SPLC/PFE conference in that year. We believe that a 14-year time frame is enough for a reasonable acknowledgement of the area to be obtained and evaluated.

Table 3.1 Search String

architecture **OR** architectures **OR** architectural **OR** architecting **OR** model **OR** models **OR** modeling **OR** structure **OR** structures **OR** structural **OR** structuring
AND
pattern **OR** patterns **OR** style **OR** styles **OR** variability **OR** variable **OR** variation **OR** document **OR** documenting **OR** documentation **OR** evaluate **OR** evaluating **OR** evaluation **OR** assess **OR** assessing **OR** assessment
AND
"product line" **OR** "product lines" **OR** "product-line" **OR** "product-lines" **OR** "product family" **OR** "product families" **OR** "product-family" **OR** "product-families" **OR** SPL **OR** SPLs

3.5 Conducting

Data Sources: In this work, we concentrate the automated search in scientific databases and the manual search directly in the selected sources rather than considering technical reports or books. We assume that most of the approaches and methods reported in books and technical reports are also described or referenced in research papers. We decided to use both automated and manual search techniques combined with the QGS for reducing bias by lowering the chance of overlooking relevant studies.

The main journals and conferences (see Appendices A and B) that cover topics in software architecture, software product lines, software quality and software engineering in general were manually searched for defining the QGS and having as result high quality studies. The elimination process is shown in Figure 3.3.

Regarding the automated search, we adapted the search string to satisfy each search engine syntax requirements. When available, we selected search parameters to return studies that satisfied the selection criteria, i.e., search filters for only considering studies under the computer science topic, and within the 2000-2013 publication time frame.

The following electronic databases were searched: ACM Digital Library[1], Elsevier – Engineering Village – Compendex[2], IEEE Xplore[3], ISI Web of Knowledge[4], and SciVerse ScienceDirect[5].

Conduction workflow: The selection of studies included the following activities:

[1]http://portal.acm.org/dl.cfm
[2]http://www.engineeringvillage.com
[3]http://ieeexplore.ieee.org
[4]http://www.isiknowledge.com
[5]http://www.sciencedirect.com

1. Run the manual searches from DBLP Computer Science Bibliography[6] and conferences websites, considering the key terms and all studies published within the defined time frame;

2. Define the "quasi-gold" standard (QGS) based on manual searches;

3. Run the automated searches using the search terms;

4. Identify the QGS by crosschecking the automatic searches results. If necessary, include more terms in the search string and run the automated searches again;

5. Exclude studies based on the exclusion criteria;

6. Exclude studies based on the full text read;

7. Exclude irrelevant studies based on researchers agreement; and

8. Obtain primary studies.

First, as recommended in the guidelines (Kitchenham and Charters, 2007), one researcher checked the title and abstract fields, eliminating papers that were not related to the subject (PLA). Such step resulted in a list of papers that were analyzed by the remaining two researchers, who validated the list by eliminating papers that are not related to the research questions. The duplicated papers from different sources were eliminated, and two researchers undertook the full-text reading of the studies that were agreed to remain in the process, after agreement of the three researchers involved. Then, after the full-text reading, the unclear cases were resolved by another round of discussions between the researchers. Bias regarding the reliability of inclusion/exclusion decisions was adjusted in frequent discussions between the researchers involved in the process.

3.5.1 Screening of Papers

The selection of studies involved a screening process for guiding the search for relevant work. From the different sources, we applied filters for selecting only potentially acceptable studies according to the defined inclusion criteria. For establishing the QGS, we manually searched 15 journals and 28 conferences. 84 studies were included from the title and year examination procedure. After examining their abstracts, 68 studies remained in the process and such set of studies was considered the QGS of the review. After

[6]http://www.informatik.uni-trier.de/ ley/db/

adjustments in the search string to consider all papers in the QGS, the automated search engines retrieved together 5697 studies. Only 320 were preserved after title and abstract analysis. Among the total of 388 studies (68 from manual and 320 from automated analysis), 272 were excluded due to duplication, and thus 116 were selected for full text reading. Two researchers undertook full text readings and after 17 studies eliminated, we agreed that 6 more needed to be excluded from the process. Finally, the resulting 93 studies represent the primary studies, which were critically appraised from the research questions points of view. The screening process is shown in Figure 3.3.

The complete list of primary studies in their categories is presented in Appendix A. We categorized the included studies according to the topic that is *explicitly* covered in them. For example, if a paper is listed in Table A.2, it reported discussions regarding the use of patterns in PLAs. The occurrences of the same study in different tables indicate that such study simultaneously addressed more than one research question.

3.5.2 Classification Scheme

For categorizing studies, we defined a set of facets that will guide researchers in drawing general conclusions regarding the primary studies and consequently the PLA area. The classification used was based on (Petersen *et al.*, 2008) and considered both general and topic specific reasoning.

As far as the general research facets, the categories allow reasonable understanding of the area by presenting the types of research and types of contribution of each primary study. With such categorization, the reader is able to easily identify the research-wise purpose of the previously published papers in the field of PLA. Among the general research facets, we included the following:

I. *Research type*: Validation Research, Evaluation Research, Solution Proposal, Philosophical Paper, Opinion Paper, Experience Paper (Wieringa *et al.*, 2005). A description of each category is presented in Table 3.2;

II. *Contribution type*: Process, Method, Model, Framework, Metric, Tool;

With respect to the context-specific classifications, we elaborated them after reading the primary studies. The data extraction phase generated several interesting pieces of information addressing the RQs, which were later distilled into facets. Such facets are presented below and further discussed while answering the RQs:

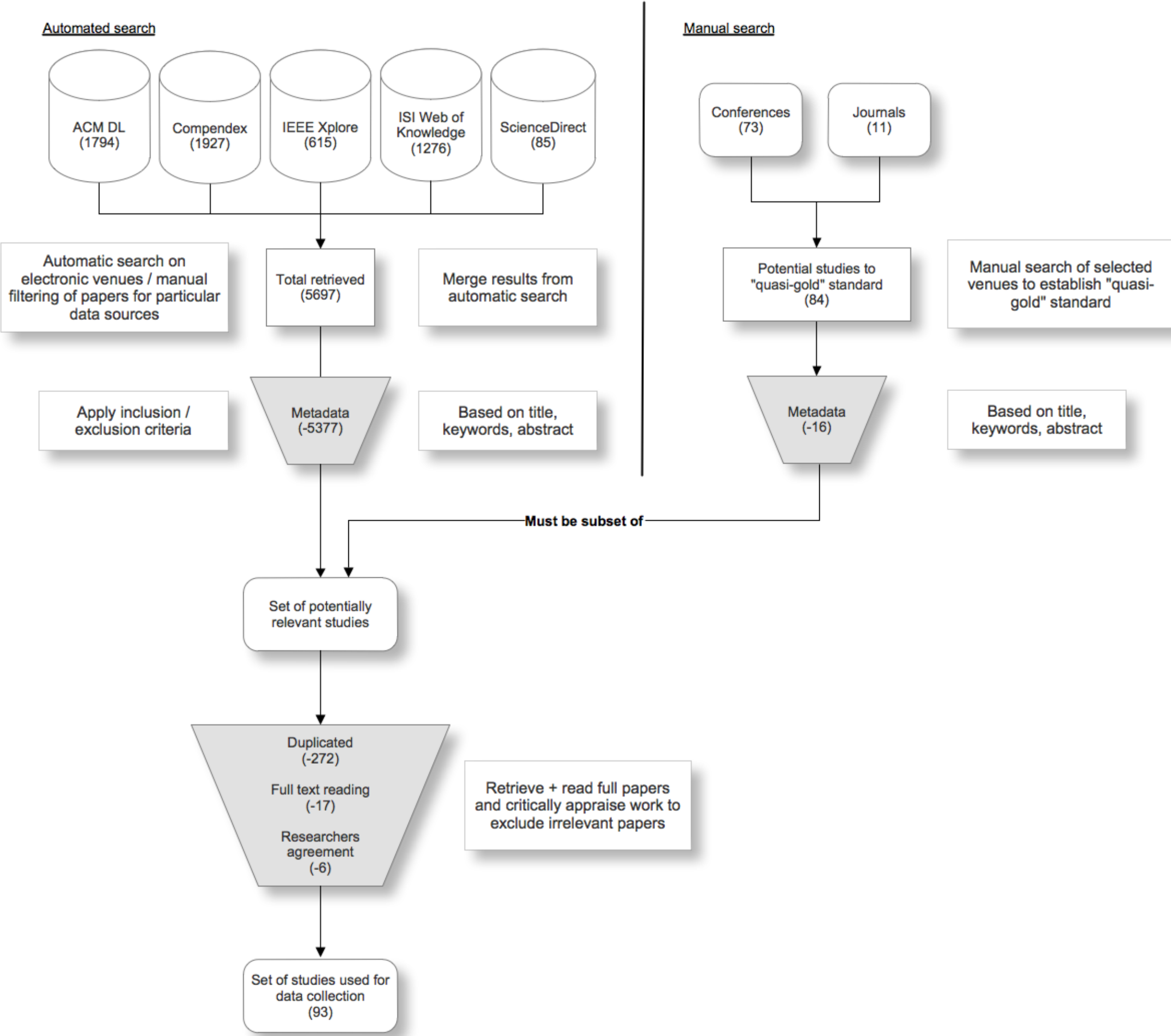


Figure 3.3 Screening of papers.

I. *Architectural patterns context*: Client requests, Service orientation, Aspect orientation, Repositories, Dataflow, Distributed systems, Derivation consistency, Adaptable systems, Multi-purpose;

II. *Artifacts related to variability*: [Integrated] Decomposition diagrams, Object-oriented specifications, UML diagrams, Process models, Component models; [Orthogonal] Decision models, Variability-Meta-Model, Conceptual models, Specific tools;

III. *Architectural view*: Logic, Development, Process, Code (Pohl *et al.*, 2005);

IV. *Evaluation method*: Architecture Tradeoff Analysis Method (ATAM); Software Architecture Analysis Method (SAAM); Scenario-Based Architecture Re-engineering (SBAR); Other;

3.5.3 Data Extraction

The data extraction strategy was designed to collect all the information needed to categorize the studies and also address the research questions. For each primary study we filled information into a data extraction form, as suggested by Dybå *et al.* (2007). The report contains general information, including the unique *identifier of the study*, *date of extraction*, *data extractor*, and *data checker*. Moreover, information related to the study description were recorded, which includes the *bibliographic information* (authors, year, title, source, venue), *objectives*, the main *problem*, *results*, and *categorization* based on the categories and facets.

Regarding the contribution of each study in terms of PLA, we extracted text related to our research questions, refined the obtained data in order to adequately answer them. The resulting data from this process was kept in text documents. We decided to keep the traceability throughout the process because we argue that it is valuable to have choices of different granularities concerning each topic of research, and also maintaining fairness as much as possible. Not rarely, studies were able to answer more than one question.

All information regarding the control of the selected primary studies, numbers and top-level categorization data was managed using spreadsheets. In the next section, we describe the outcomes of the review process, which include general conclusions, graphs, and discussions.

Table 3.2 Research Type Facet

Classes	Description
Validation Research	Techniques investigated are novel and have not yet been implemented in practice. Techniques used are, for example, experiments i.e., work done in the lab.
Evaluation Research	Techniques are implemented in practice and an evaluation of the technique is conducted. Implementation of the technique is shown in practice (solution implementation) and the consequences of the implementation in terms of benefits and drawbacks (implementation evaluation) are demonstrated.
Solution Proposal	A solution for a problem is proposed, the solution can be either novel or a significant extension of an existing technique. The potential benefits and the applicability of the solution is shown by a small example or a good line of argumentation.
Philosophical Papers	These papers sketch a new way of looking at existing things by structuring the field in form of a taxonomy or conceptual framework.
Opinion Papers	These papers express the personal opinion of somebody whether a certain technique is good or bad, or how things should be done. They do not rely on related work and research methodologies.
Experience Papers	Experience papers explain what and how something has been done in practice. It has to be the personal experience of the author.

3.6 Outcomes

In the previous sections, we described the phases of preparation, conduction, and data extraction of the review. From the process, we were able to obtain relevant knowledge in the field of PLAs. In following sections, we present the findings of the review, including the categorizations and the answers to the research questions. Further, we carry a discussion regarding the current knowledge on the topics previously defined: patterns, variability management, documentation and evaluation.

3.6.1 An Overview of the PLA field

In this section, we present an overview of the primary studies, followed by the answers to the research questions. We show a classification of the top 11 publication sources according to their contribution to this study in Table 3.3. Some conferences were merged together because we consider they represent the same venue or joint events. For example, the Product-Family Engineering (PFE) workshop was extinguished in 2004, as the Software Product Line Conference emerged and has been maintained active every year. As expected, a considerable part of the included studies was published either in ECSA/WICSA or SPLC/PFE proceedings. The concentration of relevant publications in those sources indicates the close relation among SPL engineering and architectural issues. Although we considered a limited number of conferences and journals for the manual

Table 3.3 Top 11 contributing publication sources

Source	Type	Count
European Conference on Software Architecture (ECSA) Working IEEE/IFIP Conference on Software Architecture (WICSA)	Conference	18
Software Product Line Conference (SPLC) Software Product-Family Engineering (PFE)	Conference	16
Information and Software Technology (IST)	Journal	6
International Conference on Software Engineering (ICSE)	Conference	5
Journal of Systems and Software (JSS)	Journal	3
International Conference on Software Reuse (ICSR)	Conference	3
Joint ACM SIGSOFT Conference - Quality of Software Architectures (QoSA) International Symposium on Architecting Critical Systems (ISARCS)	Conference	2
International Conference on Software Engineering Advances (ICSEA)	Conference	2
International Conference on Computational Science and its Applications (ICCSA)	Conference	2
International Conference on Software Engineering and Knowledge Engineering (SEKE)	Conference	2
Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)	Symposium	2

search, relevant studies that were retrieved with the automated search and published by unsearched sources were also included.

Temporal View: In terms of publication years, we identified a trend that allows us to briefly conclude that architectural aspects are becoming a frequently visited topic to SPL-related studies. Perhaps the increasing interest in the area is due to the great magnitude in complexity of the latest known families of systems. In addition, more organizations are adopting software reuse approaches as a viable and valuable practice to optimize their business processes. In Figure 3.4, the distribution of studies according to their publication years is presented.

Research Contibution Views: One of the main contributions of mapping studies is the *map*, frequently a bubble plot representation (Keith and Wen, 2010) of different perspectives. The graph is designed based on the extracted information, and takes advantage of multiple dimensions allowing reflexive reasoning on data. For the purpose of this study, we chose to consider the number of identified studies according to the research type, contribution type, and area of interest within PLA. The bubble plot is shown in Figure 3.5. Since studies often address multiple RQs, the sum of the numbers in the bubbles is higher than the number of primary studies included. The ‘solution proposal’ category carries the highest number of studies, and within this research type, the majority

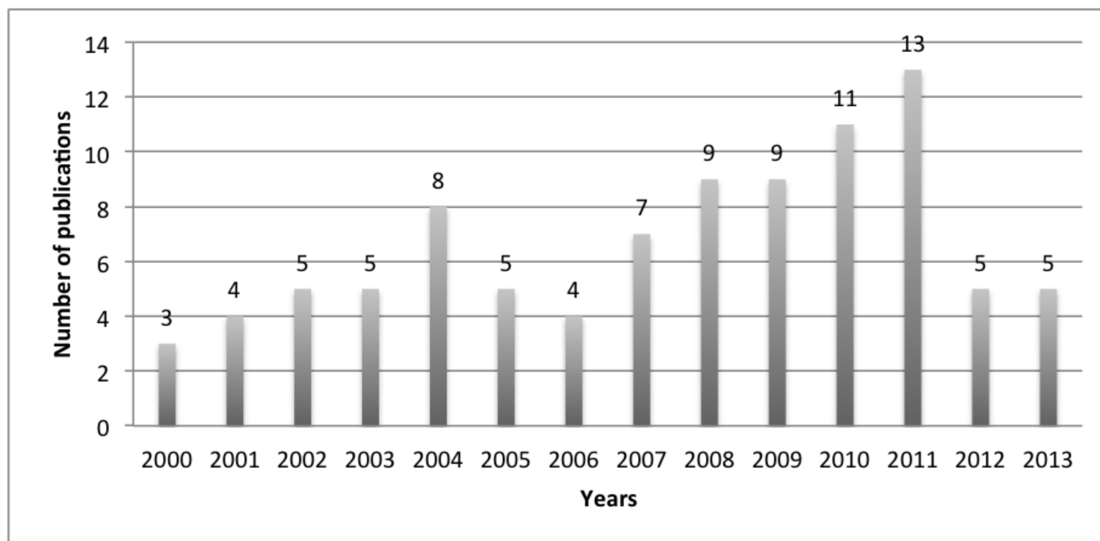


Figure 3.4 Distribution of studies in publication years.

of the studies somehow answered questions regarding the problem of handling variability at the architecture level.

Most studies propose solutions to a problem. In cases where a particular study presented characteristics of both 'solution proposal' and 'validation research', for instance, we considered it to be in the 'solution proposal' category, since the main objective is to actually solve a problem. For example, when a study proposed a novel method for solving a problem and afterwards applied some form of empirical validation.

We found it worthwhile to also categorize the studies in terms of contribution to the research community. Through Figure 3.5 we present the number of studies that aim at different goals according to the type of contribution, following the classification scheme mentioned in section 3.5.2. In summary, we identified that 67.7% of the primary studies proposed a method for resolving specific issues (e.g., evaluating derived products according to a set of quality attributes), followed by the proposal of a framework (8.6%), process (7.5%), metric (6.4%) and tool (3.2%).

3.6.2 Findings

In this section, answers to the research questions are addressed. We defined a number of topics to better categorize knowledge regarding each research issue. As previously mentioned, many times the studies were able to contribute to multiple topics of research, e.g., a proposal of a mechanism for documenting variability properties in an architecture process provides answers to both 'variability' and 'documentation' issues. The

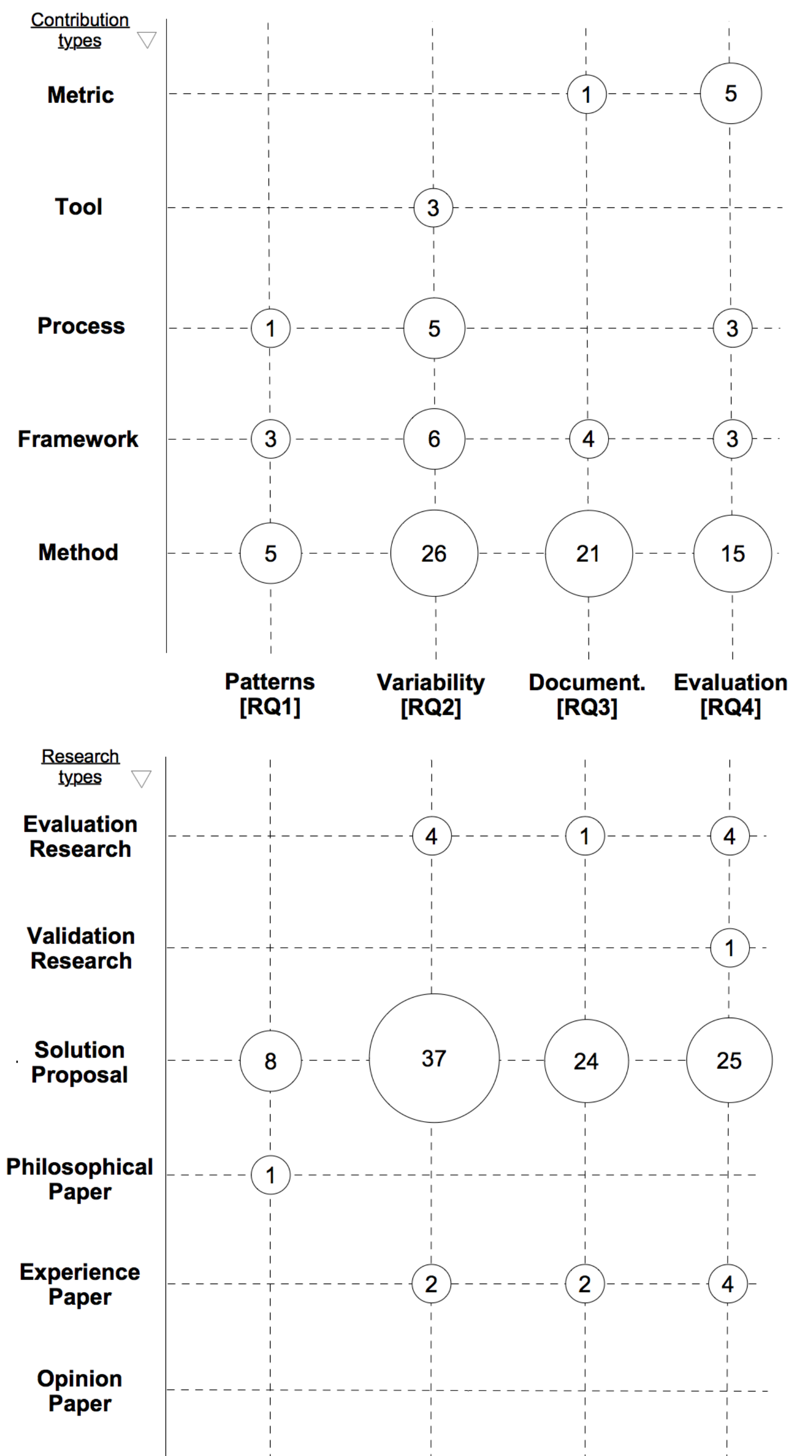


Figure 3.5 Overview of the area through a bubble plot.

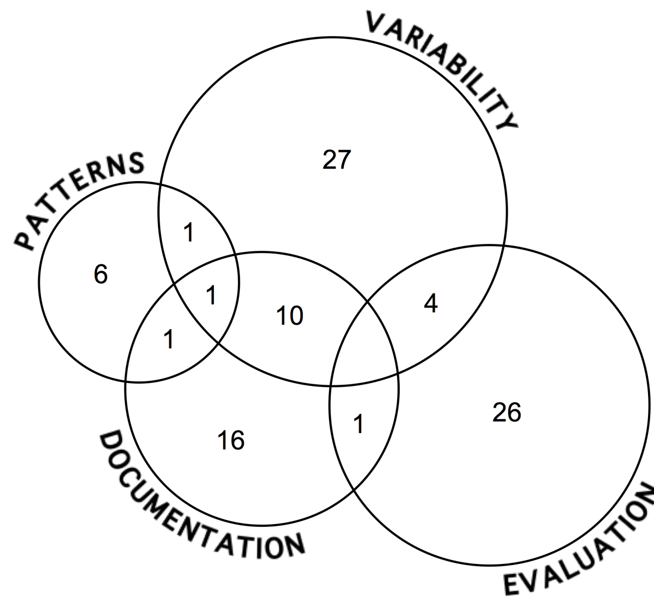


Figure 3.6 Number of studies addressing research topics.

distribution of studies that contribute with answers to each question is shown in Figure 3.6.

Most of the included studies address *variability* issues. It indicates that such aspect is frequently visited within PLA, as recent studies discuss mechanisms to explicitly deal with variability. Furthermore, *evaluation* issues are constantly subject of interest, mainly concerning scenario-based architecture assessment. Different forms of *documentation* are often objects of study, including Architectural Description Languages (ADLs) and extensions of existing methods originally for single systems. It is also noticeable that approaches for designing SPLs have emerged in the recent years, by either proposing novel ways to structure the product line or adapting already established activities. Not as much work has been undertaken regarding discussions on *patterns* issues, although several solutions made use of layered systems to design PLAs, for example.

The links shown in Figure 3.2 (topics within PLA) indicated the connection between variability handling and every other topic of interest covered in this review. In addiction to such assumption, after the review we were also able to identify links between (i) 'patterns' and 'documentation'; and (ii) 'documentation' and 'evaluation' (as shown in Figure 3.6). For example, one paper (Johansson and Höst, 2002) explicitly discussed the role of documentation in evaluating PLA degradation. On the other hand, we identified one paper (Babar, 2004) that discussed the importance of documenting architecturally significant information found in architectural patterns in the context of PLAs. Next, we

present the answers to each research question according to the findings from the primary studies.

RQ1 - Are architectural patterns (or styles) used in SPL?

Patterns are commonly used to help answering questions regarding how and where computer resources are distributed and how their communication should be implemented. As stated by Hallsteinsen *et al.* (2003), the goal in using patterns should be to resolve architectural problems by implementing a pattern language containing both known patterns and 'local' solutions for recurrent problems.

In our MS, we found only 9 primary studies that addressed explicit discussions about the use of patterns in PLAs (see Table A.2 in the Appendix). These studies proposed solutions to recurrent situations when developing SPLs, such as the deployment of services, distributed systems, or deriving products in the mobile games domain. We were also able to notice a few general-purpose (though SPL-specific) solutions to build PLAs. Some of the included studies reported frameworks to aid in the selection of adequate patterns that are commonly used in single systems. The metrics used in the decision are commonly based on the domain and on the desired quality attributes for that particular set of products to be developed.

Despite the number of studies addressing specific discussions about the use of patterns in PLA, we identified 33 primary studies that used some type of pattern in their architectures. Among those, we found 14 (42.4%) studies that used a layered architecture in their solutions. Such decision is driven by the achievable benefits of reusability and higher availability in systems designed that way. Next, we present the patterns found in the literature to structure PLAs.

The patterns. The patterns discussed in the primary studies are organized in categories in Table 4. The categories represent the type of organization used across different patterns. The references in parentheses show the authors who originally proposed the related patterns.

For managing client-request systems, which require appropriate process concurrency mechanisms to be implemented and supported by the architecture, the **Reactor**, **Proactor** and **Leader/follower** patterns are considered (Meister *et al.*, 2004). The priority of communication among components and coordinating services can also be adequately resolved by the Communication process pattern. Such practice is incorporated in the Heterogeneous style based ARchiTecture (HEART) model, which consists of three

Table 3.4 Patterns in PLAs

Category	Pattern	QAs	Studies
Client requests	Reactor (Synchronous) Proactor (Asynchronous) Leader/Follower (Concurrent)	Concurrency	Meister 2004 (Schmidt 2000)
Multi-purpose	MVC	Pluggable Exchangeability Usability	Meister 2004 Kim 2008
	PAC		Meister 2004
	Layered systems	Security Maintainability	Murwantara 2011
Aspect-orientation	UI-oriented Descentralized control Centralized control Contents-adaptable	Reusability Performance Traceability	Cho and Yang 2008
Service-orientation	Communication process	Concurrency	Lee 2010
Repositories	Blackboard	Fault tolerance Robustness Changeability Maintainability Reusability Usability	Kim 2008 (Bass 2003; Buschmann 1996; Shaw 1996)
Dataflow	Pipes and Filters	Availability Replacement Reusability Testability Usability	Kim 2008 (Bass 2003; Buschmann 1996; Shaw 1996)
Distributed systems	Broker	Interoperability Portability Reusability Changeability Extensiblity Client	Kim 2008 (Bass 2003; Buschmann 1996; Shaw 1996)
	Transaction | Query | Notifications (Centralized, Distributed, Asynchronous)	Concurrency	Morisawa 2001
Derivation consistency	Component-Relationship	Flexibility	Murwantara 2011
Adaptable systems	Microkernel	Flexibility	Meister 2004 (Buschmann 1996)

decomposition levels, and each level addresses specific design goals within a given domain by adopting architectural styles (Lee *et al.*, 2010).

There are cases in which PLAs are required to separate the representation of the information from the user interaction modules, following the same principles commonly followed in single systems. In such cases, which can be applied to several domains, the **Model - View - Controller** (MVC), the **Presentation - Abstraction - Control** (PAC) and **Layered solutions** can be used. These multi-purpose approaches were used when practitioners required functional flexibility, such as in a product line of statistical analysis software (Meister *et al.*, 2004), in which highly customizable products are derived based on different visibility groups.

Variable features can be implemented using aspect oriented programming. In these cases, framed aspects can be used to manage variability. Besides the implications of the particular domain, the use of aspect orientation allows the deployment of a number of patterns for the PLA. In this MS, we found approaches focusing on the control and structure of architectures in the mobile games domain(Cho and Yang, 2008).

A decision regarding which pattern should be used must take into consideration the domain and the purpose of the pattern to be adopted, as shown in the DRAMA framework (Kim *et al.*, 2008a). In the case of a service-oriented architecture, for example, the **Communication process** pattern is suggested to assure efficient concurrency in dealing with multiple services. The framework further discusses the organization of repositories using the **Blackboard** pattern (prioritizing robustness and fault tolerance), and dataflow-related solutions using the **Pipes and Filters** pattern (prioritizing availability and testability). The latter provides means to easily replace modules in the process, although testability is affected due to the data flow principle. Another discussion carried within the decision framework is whether there are distributed systems. In this case, interoperability is required to be a priority in order to manage concurrency, thus the **Broker** and **Transaction - Query - Notifications** patterns are recommended.

To address SPL derivation issues, we found that the **Component - Relationship** pattern can be used to assure the consistency of variable features to be composed into a product (Murwantara, 2011). The proposed pattern consists of three layers that collaborate to manage the derivation consistency: Component (to manage the derivation process), Relationship (to manage the consistency of variability), and Relationship-to-relationship (to manage quality attributes).

Finally, we identified the **microkernel** pattern, which can be used to enable functional flexibility. It was originally proposed to support the design of portable, adaptable

operating systems and to enable their extension by new services (Meister *et al.*, 2004).

We found that the solutions proposed by the approaches are isolated and very diverse. Approaches did not consider variability explicitly, as they used well-established patterns from single systems. The exception is the Component-Relationship pattern, which takes advantage of derivation properties. However, this approach lacks rigorous validation and a deep analysis on the side effects acquired when utilizing it.

Quality attributes as decision drivers. Morisawa and Torii (2001) state that architectural styles are usually selected based on designer's experience. It is known that patterns adoption decisions have high impact on quality, cost of development and administration. However, choosing among different patterns can be difficult, especially for non-experienced practitioners. One needs to consider that architectural patterns are closely related to quality attributes. The choice of using a particular pattern results in easily resolving a recurrent design issue, but also commits the software under development to the advantages and disadvantages related to that pattern.

One way to properly analyze the tradeoffs between deciding upon a pattern is through the DRAMA framework (Kim *et al.*, 2008a). It takes into consideration the positive and negative impacts on the following quality attributes: Availability, Modifiability, Performance, Security, Testability and Usability. In addition, they cross the evaluation of such quality attributes with a number of well-known patterns in a decision table.

Another interesting approach is reported in (Hallsteinsen *et al.*, 2003), which is based on quality scenarios. The process of deriving product architectures considers a decision model to select patterns that are suitable for the quality requirements. Since the selection of a particular pattern usually affects more than one quality attribute, designers must prioritize them in order to achieve the desired goals.

In fact, studies that discussed means to decide between patterns rely on the specific requirements of the SPL domains. The most considered quality attributes were related to Security, Maintainability, and Flexibility within the context of SPLs.

Moreover, it may be necessary to combine the use of different patterns (Goedicke *et al.*, 2004). In this case, scenarios can be used to bridge quality attributes and patterns in the design guidelines and help resolve trade-off situations (Hallsteinsen *et al.*, 2003). When patterns are established, there may be a conceptual confusion followed by the struggling in mapping the relationships among the patterns, quality entities, and scenarios. The relationships among those aspects can however be made explicit by using templates to document architectural information (Babar, 2004).

In conclusion, we identified very few approaches that discuss patterns and their relations with quality attributes explicitly in the context of PLAs. We argue that properly and explicitly bridging those aspects would be valuable when both selecting patterns and validating requirements.

RQ2 - How is variability handled in the architecture level of SPLs?

One of the main challenges in PLA development is to effectively accommodate the variability of the member products (Lin *et al.*, 2010). Once the feature model is designed, it requires considerable effort to maintain traceability and properly represent variability throughout the following SPL disciplines. The bridge between the feature model and the PLA is particularly important because it will guide the development process to an actual solution regarding an architecture that will support the instantiation of several others. In fact, it is known that when PLAs fail, such thing does not occur because they are not properly designed, but because they are not properly described as regards to variability (Moon *et al.*, 2006).

For this reason, representing commonality and variability in an adequate way is crucial to the quality of the overall SPL process. Recent discussions have been undertaken regarding the representation mechanisms for PLAs, and include the need for new variability mechanisms. Such phenomenon occurs due to the lack of research analyzing to what extent the existing approaches can express commonality and variability (Ahn and Kang, 2011).

In this work, we refer to variability as the aspect that drives different products to be derived from the PLA, considering the composition of different features. We do not refer to flexibility of the PLA, which covers the adaptation and changes in an SPL architecture (Galster, 2010). Variability in the context of this paper refers to the different architecture/product versions that can be instantiated from the SPL architecture, used as a platform.

Integrated vs. Orthogonal. There are currently two types of variability modeling techniques in PLA: *integrated*, which extend traditional software artifacts with variability aspects, and *orthogonal*, which add new representations separately from existing artifacts. Issues involving the usage of both types are discussed in the literature (Galster, 2010; Tekinerdogan and Sözer, 2012).

Integrated type techniques overlap functionalities of existing modeling features (e.g. creating stereotypes in Unified Modeling Language (UML) diagrams) to represent vari-

ability. Examples of integrated approaches include the KobrA mechanism (Atkinson *et al.*, 2000) and an object oriented approach, which extends UML notations using statecharts (Gomaa, 2000).

On the other hand, an orthogonal technique manages variability in a separate model, which requires a mapping strategy between the actual architectural objects and the corresponding variability entities. One example of such technique is shown in (Capilla and Babar, 2008), which also takes into consideration product constraints and the variability binding times. The main orthogonal approach adopts an Orthogonal Variability Model (OVM) (Pohl *et al.*, 2005), through which the variation points are identified and mapped to components in a component model. Such approach presents the advantage of maintaining the usual architecture model, since variability is managed remotely. In this case, the mapping between the OVM and the actual model adds complexity and requires robust traceability mechanisms. Two studies addressed the use of OVM (Ahn and Kang, 2011; Murwantara, 2011), and another one presented an approach aiming at integrating the OVM and feature modeling into LISA, a toolkit for architecture management an analysis (Groher and Weinreich, 2012).

Traceability. Since variability is a major concern in SPL engineering, it would be desirable to maintain an efficient traceability mechanism between the development artifacts. In the context of this work, we consider development artifacts every output generated from a SPL development discipline. For instance, the feature model in the requirements discipline, the component model in the architecture discipline, and the source code in the implementation discipline. The correct mapping of variable properties between such artifacts indicates the consistency of the development process, and eases the future maintenance activities.

From requirements to architecture disciplines, for example, the description of how to bind variability points can be used through meta-modeling approaches (Moon *et al.*, 2006, 2007; Díaz *et al.*, 2011). The commonalities and variability decisions are stored, and the product-specific requirement is formally mapped to a component in the PLA model.

We found that in an integrated approach, the practice of extending a component model is supported by the variations contained in the feature model. In this sense, the properties and relationships in the feature model will determine how the components interact (Mann and Rock, 2009; Lin *et al.*, 2010; Murwantara, 2011).

On the other hand, in orthogonal approaches, we identified the importance of rigorous mapping mechanisms between the commonly used design representation models (e.g.,

the component model) and the variations in the variability model. The mapping between the entities is managed with control tables and graphical representations, according to the desired focus. The graphical representation carries the disadvantage of being too limited for large amounts of data in the feature and in the variability models.

We found that, in several cases, variability properties were not explicitly bridged between artifacts (e.g., views, models). We argue that explicitly documenting where and which parts of the system vary, between different artifacts, brings beneficial effects to the understandability of the overall project.

In order to address such issue, a specific model can be used, such as the one proposed in (Galvão *et al.*, 2010). The model takes into account cognitive aspects (e.g. assumptions, properties and evidences) to capture the rationale behind the variability design, using an architectural description language. However, the approach requires refinement and is limited in regard to automatic interpretation. Formally capturing rationale behind variability decisions is useful for implementing traceability mechanisms. For example, first-class descriptions provide traceability of variation points across requirements, design and code Goedicke *et al.* (2004). Further, model-driven techniques can be used to trace variability to and from features and thus justify variations in different artifacts.

In summary, several issues are still open regarding traceability in software architecture. Aspects to be investigated include the balance between the real need of traceability and documentation overhead, as well as the application of (mature) traceability models in heterogeneous environments (Galster *et al.*, 2013).

Supporting languages. The use of ADLs is explored for specifically supporting variability properties in PLAs. One of the approaches that we found in the literature is ADLARS (Bashroush *et al.*, 2006), which attempts to bridge product features and the PLA through the following views: system-level, component-level and task-level. The language describes templates, which capture the feature dependencies and allows a straightforward derivation of product-specific architectures. This ADL allows the mapping between feature model entities and architectural entities. Since it also represents the behavior (i.e., the transition between component/interface states) of the specified entities, we classify it as an orthogonal approach for representing variability.

Further, we found a number of extensions to existing description languages (e.g., MontiArc in (Haber *et al.*, 2011a)). These integrated approaches take into account the already existing representation mechanisms (e.g., describing components and interfaces) and add parameters to enable the representation of variability as well.

Where variability is represented. Solutions in the literature describing ways to address variability in PLAs vary widely as far as processes and architectural artifacts (such as descriptions, views, models and viewpoints) that are taken into account. Furthermore, in general, they were proposed to resolve local issues and lack rigor in validation through empirical methods. Many different procedures and techniques are described, which calls to question the existence of conventions and silver bullets that would for instance be scalable and flexible enough to be utilized in a range of different domains.

Next, we present the classification of the artifacts involved in the proposed variability representation solutions. As far as integrated approaches, the entities through which variability points can be expressed include:

- **Decomposition view diagrams:** containing a specialized module to manage the rules and configuration (derivation) of required specific architectures/products (Bachmann and Bass, 2001; Thiel and Hein, 2002); or subsystems in the FORM approach, considering the link with the feature model (Kang *et al.*, 1998);
- **Object-oriented specifications:** expliciting predefined interfaces to articulate variable features (Pinzger *et al.*, 2003);
- **UML diagrams:** taking advantage of established mechanisms such as inheritance, extensions, parametrization (SEI, 2001); and aggregation and specialization (Matinlassi *et al.*, 2002); also considering activity diagrams (Abu-Matar and Gomaa, 2011); and use case models, which add a variability mechanism through the PLUS approach (Gomaa, 2004);
- **Process models:** which are associated with entities of the feature model, in the FORM approach;
- **Component models:** which are often enhanced with additional variability information (Taulavuori *et al.*, 2004);

Moreover, we identified a number of mechanisms that address variability in separate artifacts. Orthogonal solutions consider the creation of the following artifacts:

- **Decision models:** reducing complexity in understandability (Dhungana *et al.*, 2007); and also through the FAST (Weiss and Lai, 1999) and KobrA (Atkinson *et al.*, 2000) approaches. In the latter case, for example, each variability point is related to at least one decision in the decision model. The decisions provide possible resolutions to be implemented in the Komponents;

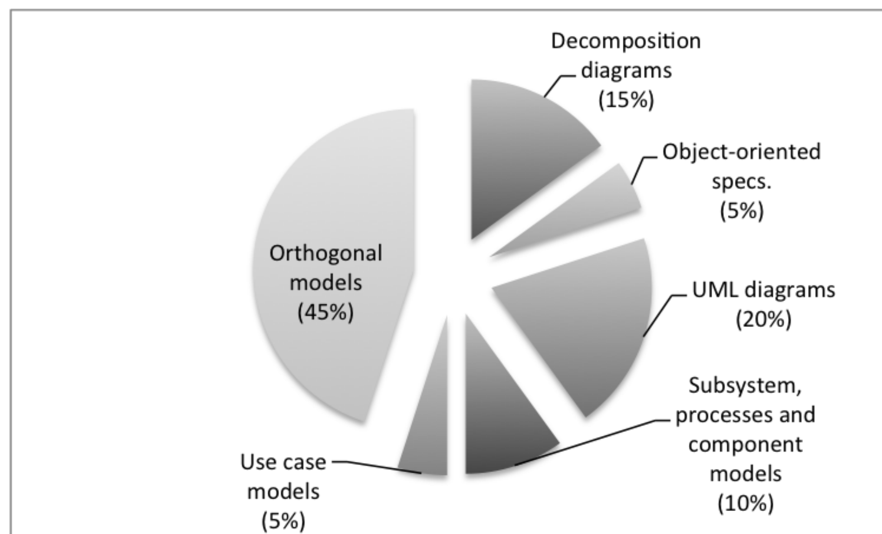


Figure 3.7 Artifacts affected by variability.

- **Variability-Meta-Model(VMM):** being responsible for modularizing variability as parameterized transformation rules. The model transformation rules already especified for the commonalities are separated from the variability rules (Kavimandan *et al.*, 2011);

- **Conceptual models:** describing architecture viewpoints based on the ISO/IEEE/IEC standard for architectural description (Tekinerdogan and Sözer, 2012) or neutral notations. The standard ISO/IEEE/IEC 42010 defines a conceptual model of architectural descriptions defining general syntax and semantics for them (Hilliard, 2010);

- **Specific tools:** associating the component model or the feature model with elementary functions, which contains variability information (Abele *et al.*, 2012). In the EPM tool, for example, the variability is introduced by declaring structural elements in a function hierarchy optional, or by attaching the feature model to an elementary function;

A graph showing the artifacts related to variability is presented in Figure 3.7. It shows the diversity of options discussed by the primary studies with regard to representation of variable properties. 44.4% of the studies that addressed such issue discussed orthogonal models, which represents a balance between orthogonal and integrated approaches. We found that the UML notation (which include component and activity diagrams, use case models) are commonly used to represent variability in PLAs.

Despite the SPL approaches, we argue that the community would benefit from the establishment of standards and guidelines to efficiently develop PLAs that clearly and explicitly deal with variability. A comparison between different approaches and methods for designing PLAs can be found in (Matinlassi, 2004a; Filho *et al.*, 2008). Further, more evaluations could be undertaken as regards to the included tradeoffs in creating or extending different design artifacts. The mapping between the created models and the remaining artifacts plays a fundamental role in quality attributes, and thus requires further discussion on orthogonal approaches.

RQ3 - How are the SPL architectures documented?

The importance of documentation is mainly revealed when time optimization issues arise. Artifacts can be used to communicate architectural knowledge, and thus help in the establishment of a traceable and consistent software process. Babar *et al.* (2009) surveyed a number of architects on their opinion regarding the role of documents in the PLA's processes, and their answers addressed the times when new architects join the team and are expected to read the architecture documents thoroughly. Such practice can be tiresome for the newcomer professional, but it saves the team a considerable amount of time, because they do not need to give presentations about the design.

It is clear that including proper documentation into architectural processes brings effective benefits to the overall project, including stakeholders such as subcontractors. In SPL, however, the existing documentation approaches focus on describing components and connectors but fail to reflect the decisions made along the architecting phase (Trujillo *et al.*, 2007). Further, documentation overhead around PLAs must be evaluated.

Frameworks and ADLs We found several approaches to document variability. These approaches are highlighted in a table below and better explained in the text.

One way to document architectures is through Architecture Description Languages (ADLs) (as in FAST approach (Weiss and Lai, 1999)). In order to support compositional specifications, algebraic languages and process algebras are used in PADL (Bernardo *et al.*, 2002), which aims at describing concurrent and distributed systems. This solution allows formal methods to be applied, thus increasing the PLA's maintainability. Formal analysis is also discussed in (Satyananda *et al.*, 2007b,a), using PVS theory specification documents to map features and the PLA. In the context of aspect-oriented solutions, variability descriptions can be achieved through the specification of components, connectors and ports. Describing both PLA and product architectures is possible through the

PL-AspectualACME language (Barbosa *et al.*, 2011).

Moreover, Extensible Markup Language (XML) schemas can be used to capture the basic elements of a SPL representation: versions, options and variants into the xADL language (Dashofy and Hoek, 2002a). These attributes are defined as independent extensions to later define a regular architecture.

An interesting approach is reported in (Babar, 2004), which takes into account the specification of scenarios (also in PuLSE (DeBaud *et al.*, 1998) and ATAM (Kazman *et al.*, 2000) approaches). The documents describe how architectural solutions can be implemented to support product requirements that are specified in the domain engineering artifacts (business and requirements documents). We argue that this practice can increase the overall process understandability, as well as improve the communication of architectural knowledge among stakeholders in the SPL project.

Further, PLAs can be modeled by using UML existing generalization concepts ((Gomaa, 2000; Muthig and Atkinson, 2002) as in the approaches: COPA (America *et al.*, 2000) and QADA (Matinlassi *et al.*, 2002)), in addition to problem frames (Dao and Kang, 2010). Mapping heuristics can be applied in accordance to each feature to be realized with a problem frame, which are in turn transformed into components using the UML-based approach for problem frame oriented software development (Choppy and Reggio, 2005). Mechanisms for explicitly handling variability are better discussed in section 3.6.2.

The extension of standards (e.g. IEEE P1471 in (Thiel and Hein, 2002)) and model-driven techniques are also object of study in PLA documentation. The use of model driven engineering for specifying features, variability, can provide a business view of the process (Asadi *et al.*, 2009; Botterweck *et al.*, 2007). This approach considers business goals and requirement models to incorporate commonality and variability properties and support product derivation.

We argue that one effective solution is having separate documentation artifacts such as in the KobrA approach (Atkinson *et al.*, 2000), which defines structural, behavioral, functional and decision models through which the 'komponents' are described and derivation rules documented.

Views. It is known that the use of multiple architecture views can help increase understandability of software design. Describing software according to different viewpoints allow specific attributes to be represented. The result is a set of models that carry different characteristics and thus aid different stakeholders in the development process.

Table 3.5 How PLAs are documented in different studies

Studies	**Method Process Framework**	**How the PLA is documented**	**Pattern related?**
(Weiss and Lai, 1999)	FAST	ADL	-
(Bernardo *et al.*, 2002)	-	PADL	Yes
(DeBaud *et al.*, 1998)	PuLSE	Scenarios specification	-
(Kazman *et al.*, 2000)	ATAM	Scenarios specification	-
(Babar, 2004)	Framework using two information templates	Relationships of scenarios, quality attributes and patterns are documented in templates	Yes
(Dashofy and Hoek, 2002a)	-	xADL (XML-based)	-
(Barbosa *et al.*, 2011)	-	PL-Aspectual ACME	-
(Satyananda *et al.*, 2007a)	-	PVS Theory specification	-
(America *et al.*, 2000)	COPA	UML	-
(Matinlassi *et al.*, 2002)	QADA	UML	-
(Gomaa, 2000)	-	UML	-
(Muthig and Atkinson, 2002)	-	UML	-
(Choppy and Reggio, 2005)	-	UML-based problem frame	-
(Atkinson *et al.*, 2000)	KobrA	Structural, behavioral, functional and decision models	-

The views together determine the structure of the system.

In this sense, we found that one way to document PLAs is through designing artifacts aimed to satisfy customers. For example, the *customer view* (as in the COPA approach (America *et al.*, 2000, 2005) can be used to describe market and business value drivers related to the architecture decisions.

Further, we found that the organization of components and their relationships can be documented using a set of three general views (Matinlassi *et al.*, 2002; Atkinson *et al.*, 2000; Oliveira and Rosa, 2009): - the *structural view*, to describe the organization of the modules, for example, through layers and descriptions associated to the entities in the design; - the *behavioral view*, to describe the behavior of the architectural elements considering the relationships between them (e.g. using scenarios); and - the *deployment view*, to describe the structures that can be deployed and form units (i.e. product specific designs).

In the approach proposed in (Gomaa, 2000), every product is considered a view of the PLA in the form of use cases. The views are then integrated into a domain model using an integration approach (Gomaa, 1995). In this approach, the key issue is to map the different views in the domain model back to the products that must be derived from it. We argue that it may be difficult to perform a reverse engineering procedure to filter design decisions that are not related to the product requirements.

We found an interesting approach based on Plastic Partial Components (Pérez *et al.*, 2009), which take into account the SPL variation properties to establish three views: - the *core architectural view*, to describe the components that are common in the PLA; - the *variability architectural view*, to specify the configurations and component variations; and - the *product architectural view*, to describe the architecture of individual products.

It is clear that solutions regarding different documentation views are sparse and many times focused on the needs of specific projects. Other views are also accounted for, to cover for example how the applications are used (America *et al.*, 2005). In order to classify the different PLA viewpoints used or discussed in the primary studies, we used the model proposed in (Pohl *et al.*, 2005), which consists of: - a *logical view* to describe the systems in terms of the problem domain; - a *development view* to describe the decomposition of the system into entities; - a *process view* to describe the behavior of the system with regard to the running system's ordered activities; and - a *code view* to show the decomposition of the executable code into files that are assigned to processing units.

In fact, from the primary studies that mentioned viewpoints or showed artifacts for our

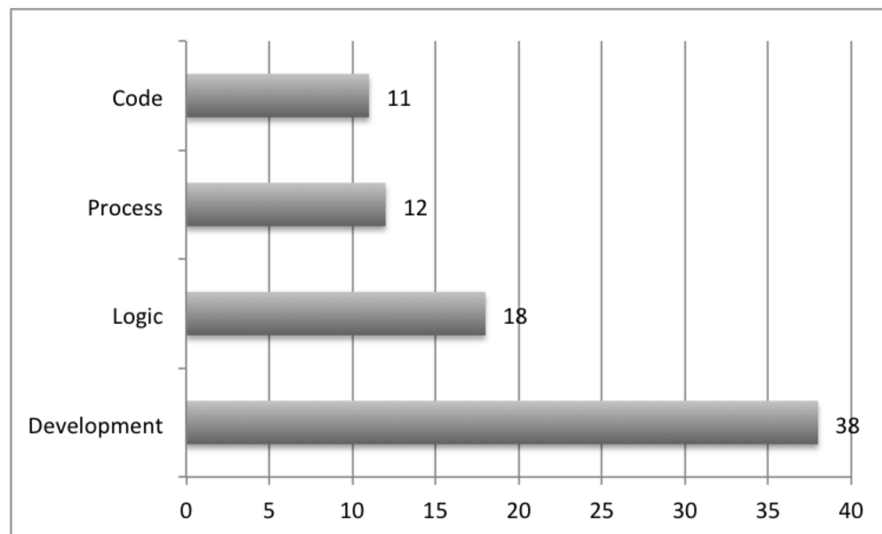


Figure 3.8 Architectural views addressed in the studies.

evaluation, the majority (76%) addressed documentation considering the development-related views, as shown in Figure 3.8. Studies also presented logic representations (36%), process (24%) and code (22%) viewpoints. Views were many times used in conjunction. The development viewpoint involves the description of the structural elements, in addition to the rationale behind the relationships between them. That is, the primary studies often presented component or class diagrams to describe the structure of PLAs. These numbers show that recent studies are more concerned with documenting PLAs from the developers' perspective, considering implementation and management issues.

RQ4 - How are the SPL architectures evaluated?

Many times called PLA assessment (van der Hoek *et al.*, 2003; Olumofin and Misic, 2007; Tizzei *et al.*, 2011), the evaluation of PLAs is often connected with two key aspects in the overall process: quality attributes (Matinlassi, 2004b; Etxeberria and Sagardui, 2008; Zhang *et al.*, 2008; Oliveira Junior *et al.*, 2010; Cavalcanti *et al.*, 2011) and evolution (Maccari, 2002; Rosso, 2006). Proper evaluation can lead to the identification of previously unknown architectural defects, and to the consequent planning of improvement initiatives (Maccari, 2002). In the particular case of SPLs, verification is important because during the product development, it is necessary to design the product based on the core PLA asset and check whether the design satisfies the selected features.

However, a number of problems may arise during evaluation, e.g., the interactions of quality attribute could lead to architectural conflicts (Olumofin and Misic, 2007). In this

sense, such procedure must be carefully undertaken and consider a tradeoff analysis.

We argue that evaluating PLA is very important since problems, potentially replicated among products, can be detected before the actual products are developed. These problems are easier and cheaper to correct in the earlier stages. Moreover, in the case of PLAs, the architecture evaluation becomes crucial to ensure that the PLA is flexible enough to support different products and in order to allow evolution.

Methods. A number of aspects need to be considered when choosing a method to evaluate a PLA, including the quality attributes to be assessed and the resources available to perform the procedure. Also, it is desirable to structure the assessment for performing both qualitative and quantitative evaluation (Abowd *et al.*, 1997). Among the quantitative methods, one can make use of design simulations, prototypes, experiments and mathematical models. On the other hand, qualitative evaluations are likely to include questionnaires, checklists and scenarios. In Figure 3.9, we present the distribution of studies addressing different methods for evaluation. 50% of the studies discussing evaluation methods addressed the use of the scenario-based method ATAM, alone or combined with its extensions.

In fact, we found that scenarios were widely discussed (Matinlassi, 2004b; Cavalcanti *et al.*, 2011) and used as the main evaluation method. It consists of identifying different cases in which the PLA is expected to provide support. That is, customer, business, structural and technical requirements are analyzed and provide basis to the creation of these cases. Scenarios are then compared to PLA artifacts (such as decomposition or object-oriented models on the technical side, and market/resources models on the business side) to identify points of improvement.

We found several studies (Maccari, 2002; Lutz and Gannod, 2003; Rosso, 2006; Olumofin and Misic, 2007; de Oliveira Junior *et al.*, 2011; Nakagawa, 2012) that discussed the use of the scenarios-based Architecture Tradeoff Analysis Method (ATAM) (Kazman *et al.*, 2000). This method takes into account the quality attribute requirements, and how the architecture artifacts respond to their stimuli (assessment). It is important to notice that prior to analysis, the architectural requirements must be expressed in terms that are concrete, measureable and observable.

Further, the ATAM has been extended to specifically cover both PLA and product-specific architectures (Olumofin and Misic, 2007). The holistic approach considers risks and quality attributes tradeoff analysis using architectural drivers and variability properties. The advantage of the HoPLAA approach is that it covers common and variable

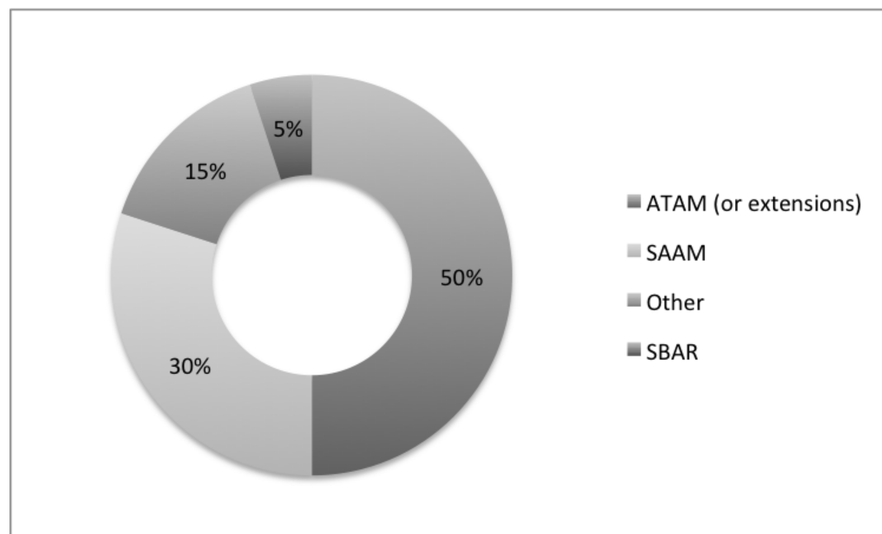


Figure 3.9 Evaluation methods addressed.

architectural aspects and allows reuse in the analysis.

We also found methods that take into consideration manual review of PLAs (Kishi *et al.*, 2005), which the authors argue is inefficient due to error proneness. Thus, another way to assess PLAs is through tools, such as design simulators and design model executions.

Kishi *et al.* (2005) affirmed that *reviewing* is one of the most common techniques to verify the design. However, it is not effective for exhaustive checking because the technique is performed manually. The authors also suggested design verification by *using tools* such as design simulators and design model execution.

Measurements against requirements. By measurement we mean the act of verifying whether the PLAs fulfill determined requirements. We argue that proper measurement can greatly increase the quality and correctness of PLAs, especially against quality attributes. The majority of the techniques and methods proposed had not developed tools to support the process (Montagud and Abrahão, 2009), which means that such measurement is usually performed with high human intervention and is thus error prone. We have, however, found one study (Lutz and Gannod, 2003) that addresses tool support for PLA analysis, taking as input ADL and additional formal specifications.

Different aspects of the PLA as a whole can be measured, such as complexity and extensibility (Oliveira Junior *et al.*, 2010), thus providing a reasonable understanding on the economic value of a SPL and identifying maintainability improvement points. Further, other aspects can be measured, such as similarity, variability, reusability (Zhang *et al.*,

2008).

We argue that the decision about which aspects must be measured should be carefully made envisioning business drivers and customer requirements. If efficient means for measurements in those areas are implemented, the quality of PLA is drastically increased.

We found that the architectural models and formal specifications are commonly evaluated against the desired quality attributes. Such measurements are driven by business processes.

Inputs and Outputs. For practical evaluation, we point the importance of adopting a process through which the input and output artifacts are explicitly considered.

When it comes to inputs, the architecture description is the main artifact to be assessed. However, we argue that architectural knowledge (Maccari, 2002) should also be assessed by considering the business drivers, SPL process properties, and different stakeholders requirements. Such extra-design information can be evaluated through also using text documents and presentation slides, as discussed in (Rosso, 2006). The generated artifacts include documents that present the actual architecture description, evolvability constraints, and schemas such as scenarios.

In conclusion, we identified that the architect's experience is important when assessing PLA. At this point, we verified that the approaches carry the need of human intervention. The architects involved in the process are required to specify parameters that directly interfere in the result of a given evaluation measure. We believe that such phenomenon occurs due to the subjectivity involved in the definition of quality attributes to be achieved. In fact, the communication between architects, developers and other stakeholders is an indirect consequence of performing assessments in PLA (Maccari, 2002). Based on the studies found, we identified a number of strategies that can facilitate the evaluation of PLAs, and most of them were related to the use of ATAM or its extensions to cover variability issues. By adopting scenarios or reviewing methods, architects are required to manually verify the consistency in PLAs. Thus, the current scenario suggests that subjectivity and expertise are commonly addressed in evaluating architectural models in SPLs.

3.7 Discussion

The increasing number of studies published that satisfy our inclusion criteria indicate a research trend over the last 14 years (see Figure 3.4). The SPL community is looking at

architectural issues in a much substantial way than it did ten years ago. We associate this evidence with the fact that information systems are becoming more and more complex and integrated, and implemented using different platforms and development approaches. In addition, many issues remain open SPL engineering research. The current scenario could benefit from efficient means to reuse knowledge and artifacts, e.g., taking into consideration the structural particularities and coupling requirements of each product.

Depite the fact that most studies (87%) proposed solutions to current problems, we noticed theoretical/methodological discussions, but few clarifications on actual experiences. Regardless of the demand on proposing new approaches to resolve issues, there is a need to properly validate the existing ones. Further, new tools and metrics support should be proposed to help managing different aspects of PLAs.

As evidenced in Figure 3.5, a low percentage of the studies proposed tools (3.2%) or metrics (6.4%). We understand that even though software architecture issues are widely discussed in the single systems domain, there is still a need for the research community to provide more effective means to specifically measure and give tool support for product lines. In addition, only 7.5% of the papers suggested the use of a process and 8.6% of a framework. These numbers suggest that the current studies are concerned with resolving smaller parts of the problem, instead of wrapping a more complete solution for the PLA domain or validating existing approaches.

Furthermore, 9.6% of the papers included explicitly addressed issues concerning Patterns in PLAs. We understand that these numbers indicate an opportunity for future research, since studies often proposed methods to choose appropriate patterns or measure quality attributes. For instance, the community would really benefit if tools were proposed to specify PLAs, and support the derivation of product-specific architectures using patterns. Next, we describe the findings according to each of the identified topics of research in PLA.

3.7.1 Main Findings

Next, we show the main findings of this study, separated according to the different points of interest previously defined for this review.

Patterns

Through analyzing the studies, we noticed a number of models that are used for obtaining solutions to recurrent architecture problems in SPL. The approaches used patterns that are

already established in the single systems domain, e.g. the model-view-controller (MVC) pattern. Among all included studies, 33 (37.5% of the total) used some type of pattern in their solutions. Further, 42.4% of these studies reported a PLA that used layers. We assert that the wide use of layered architecture occurs because with this solution, it is easier to make changes, and the SPL management is better performed through modularity.

In fact, a number of solutions were presented towards guiding SPL architects for classifying or selecting appropriate patterns for use, mainly considering quality attributes that are affected with the choice. However, there is a lack of empirical assessment in measuring the effects to SPL properties in applying those patterns. For example, it is clear that in SPL the derivation process is greatly affected by the properties of the selected patterns. However, only one study proposed a pattern that specifically deals with such SPL aspect. It makes explicit the components and their relationships to preserve consistency in the derivation process.

In summary, we argue that further investigation is required e.g., in model driven approaches that support formal specifications and thus allow automated mechanisms to be used.

Variability

Variability is a major concern in SPL. We verified that this aspect plays an important role also during architecture definition. Variability should be treated all throughout SPL disciplines, and thus the process should include effective traceability mechanisms between the artifacts. In this MS, we identified few approaches that maintain traceability between artifacts, however the existing ones still need refinement. The solutions should be empirically validated and better justified with respect to the trade offs involved in the process. In addition, it would be valuable to discuss the balance between the actual need to trace and the possible negative effect on usability and maintainability of documentation.

For representing variability, several studies discussed the use of integrated and orthogonal approaches, and we found that the majority (55.6%) of the studies addressing this issue presented integrated solutions. An integrated approach overlaps functionalities or properties of existing modeling techniques to express variations. We argue that the wide use of integrated approaches possibly represents the inflexibility of practitioners with regard to having additional technologies or processes to learn from. Further, by applying an integrated solution, it is not necessary to maintain as many artifacts as in an orthogonal solution. On the other hand, by using orthogonal approaches, despite the need to adequately map variability from a remote model to design artifacts, it is possible to

achieve a higher level of organization, thus understandability upon the related artifacts.

Moreover, we found that in many cases UML diagrams, domain-specific or architectural description languages are used for describing variation points. We argue that future publications should also take into account the binding time in with the variation occurs, thus discussing whether original artifacts representing variations can continue to be used, e.g. during runtime in a dynamic derivation approach.

Documentation

We noticed that the same methods used in traditional development for documenting architecture are used in SPL. Several studies addressed discussions on the implications of using UML diagrams and text documents, which suggest that variability is many times simply incorporated into the documentation artifacts, followed by adjustments in the process.

It could also be noticed that ADLs, along with suitable extensions, have been used to describe PLAs and thus support the range of products to be derived. The rules for derivation can be specified through formal methods, which allow the mapping between features and architectural entities.

Further, 76% of the studies that reported any means for documentation used representations in the development views. In this sense, concepts such as behavior, deployment and generalization were constantly mentioned. We argue that different views should also be discussed, since it is important for the stakeholders to have a reasonable understanding of the PLA from different viewpoints (e.g. process and logic).

It is clear that documenting not only the architecture itself, by also the knowledge and decisions, is important because such information may be valuable to future modifications. Moreover, model driven solutions can be explored to help in the definition and processes related to PLAs.

Evaluation

The importance in evaluating PLAs is noticeable from the discussions in the studies. We found that the ATAM method is widely discussed among the solutions for evaluating PLAs. This approach takes into account a measurement of quality attributes and business drivers to assess architectures. It does not originally cover variability issues, so extensions are used to cover such aspect. 47% of the studies that addressed methods for evaluation included ATAM or its extensions, which suggests that such technique is widely accepted in SPL practices as well.

From the studies that discussed evaluation methods, most of them considered the use of scenarios. Such practice allows different perspectives to be taken into account. We argue that this practice represents an effective way to evaluate PLAs, when compared to approaches that do not consider customer, business, structural and technical requirements within the evaluation.

3.8 Threats to Validity

Despite the fact that we asked broad questions and used a systematic method to reduce bias, it is possible that the search strategy was not designed perfectly so it covers the most relevant topics of research within PLA. On the review process, one researcher structured the contents of the protocol and presented to the research group for feedback, and thus, the review terms are a result of many discussion sessions.

We cannot guarantee that all relevant studies were included, although we put significant effort to avoid this threat by combining automated and manual searches. In addition, we established the "quasi-gold" standard to calibrate the search string and increase the trustworthiness of the review. Moreover, we conducted the search on the leading digital libraries (automated), journals and conferences (manual) according to experts.

Regarding the data extraction process, the granularity of the selected answers to questions and consequently conclusions drawing were based on our judgment. As an attempt to reduce bias, more than one researcher performed the classification process, and discussions were undertaken to settle agreements.

Also, studies that presented both a solution to a problem and some sort of validation method or experience reports, for example, were classified as solution proposal because this refers to their main goal. This decision might have influenced the classification scheme.

3.9 Chapter Summary

In this chapter, we discussed the motivation for performing a systematic mapping study. We presented details on the method chosen to review literature, and how it differs from classic systematic literature reviews. Further, we also presented the findings of the systematic mapping study performed in the field of PLAs. Based on the guidelines proposed by Kitchenham and Charters (2007) and Petersen *et al.* (2008), we selected a set of studies that satisfied a number of criteria. Then, we analyzed, categorized, and

extracted information from them in order to answer our research questions. We were able to provide evidence of the fields that still require further research for a consistent body of knowledge, despite the growing number of studies that are being published every year concerning PLAs.

We argue that one of the most relevant aspects to be discussed in this field is PLA evaluation, due to the inherent complexity involved in the SPL processes. However, we were not able to find approaches that attempt to evaluate PLAs explicitly with respect to lifecycle attributes, without considering business aspects. Thus, in the next chapter we present an exploratory study in the field of architectural bad smells in SPLs.

With the exploratory study, we aim to characterize the phenomenon of architectural bad smells in the context of SPLs, because such smells were initially proposed to single systems. We argue that such an investigation is valuable because the identification of architectural smells do not depend on previous knowledge with respect to the history of the system or organizational aspects.

4

Architectural Bad Smells in Software Product Lines: An Exploratory Study

Software architectures are often required to be modified or completely redesigned over time to improve software quality. Nevertheless, such activity is many times absent from a PLA's lifecycle because changing design decisions might represent a risk that affects several products. Instead, changes are usually made to the product architecture, which is a simpler choice to undergo.

However, we argue that points that can improve the maintainability of PLAs should be identified and treated whenever possible. Even though such task requires careful analysis and might result in changes in all products, the overall quality can also be increased. When the PLA is improved, benefits can be achieved in both the platform and in product specific structures.

This chapter is organized as follows: Section 4.1 presents the study setup, including the research questions and the conducted workflow. Section 4.2 presents information about our object of study: Notepad SPL. Section 4.3 describes the process followed to perform the recovery of its architecture. Section 4.4 describes the process of identification of smells from the extracted architecture. Section 4.5 presents the threats to the validity of this study. Finally, Section 4.6 presents a summary of this chapter.

4.1 Study Setup

One way to assess PLAs is through the identification of bad smells. Despite the different methods for evaluation and evolution of SPLs, to the best of our knowledge, no studies have discussed the issue of smells in PLAs. We argue that considering lifecycle properties is important because guidelines can aid in the improvement of the internal structures with

respect to quality attributes.

According to Runeson and Höst (2009), an *exploratory case study* should be performed when the purpose is to "find out what is happening, seeking new insights and generating ideas and hypotheses for new research". In this sense, the present study aims at characterizing the problem through the main question: "***Do architectural bad smells occur in software product lines?***". We are particularly interested in investigating whether the same identification method used in single systems is also effective in the SPL context. Further, we check whether the same smells that occur in single systems also occur in SPLs. In case of occurrences, we discuss their behavior and the implications of having such architectural design attributes in a variability-based environment.

In order to address the aforementioned issues, we selected a sample SPL that was available online and refined the implementation within our research group. Since there was no architecture artifacts related to this project, we undertook a recovery process prior to identifying smells. While searching for the four representative smells, we noticed the occurrence of a fifth smell, which is related to the context of variability expressed by the PLA. Thus, in addition to the search for the existing smells, we propose a new one.

Five major steps are suggested when performing a case study: (i) define objectives, (ii) prepare for data collection, (iii) collect evidence, (iv) analyze collected data, and (v) report findings (Runeson and Höst, 2009). After defining the objectives, cited earlier, we strictly followed these guidelines by having a workflow consisting of:

(1) analyzing a sample SPL in the domain of text editors / desktop;

(2) extracting its conceptual architecture from the code;

(3) searching for smells in the recovered architecture; and

(4) analyzing results.

Next, we describe the process in detail, starting with a description of the sample SPL.

4.2 Notepad SPL

The object of this study is Notepad, a Java implementation resulting from a feature-oriented design course in the University of Texas at Austin. This sample product line was also used in an empirical study within the FeatureVisu project, which is a structure analysis and measurement tool for SPLs (Apel and Beyer, 2011). The class assignment

at the University of Texas was to individually develop an application in the text editor / desktop domain using a common base and applying feature orientation development concepts.

The Notepad release contains a set of 7 different products, and can be obtained in the FeatureVisu project website[1]. Each of the products implements a random set of features related to text editing, such as *Copy*, *Paste*, and *Find*. The products contain from 1397 to 1716 lines of code, and also vary in the number of features: 4 to 10.

4.2.1 Feature Model

Since the source code of each product was conceived separately, the first step of our work consisted in analyzing the code and designing a Feature Model using the FeatureIDE tool (Thüm *et al.*, 2012) and providing each feature the proper level of abstraction. A few graduation student members of RiSE research group[2] were involved in the task of modeling the features and turning the separated products into a single product line.

The features in the feature model were grouped according to the user's point of view, representing dropdown menus in the GUI (i.e. *File*, *Format*, *Edit*, *Help* and *Menu*), as presented in Figure 4.1. All features on the tree represented as leaves refer to selectable functionalities when deriving a product. The inside nodes (except the root) represent a higher level of abstraction responsible for grouping and organizing the features from the user perspective. Solid circles connecting to their parents represent mandatory features, and hollow circles represent optional features.

The Feature Model serves as a guideline to develop the different products supported by the SPL. It defines the rules for the composition of features and allows stakeholders to acknowledge the scope of the product line through an overview. All products to be derived from the SPL must be supported by the rules contained in the feature model.

In this particular SPL, there are no constraints related to the inclusion of features. That is, there are not cases in which the inclusion of a feature requires the inclusion of another feature. The same occurs to the exclusion of features. In order to handle both product specific requirements and feature coexistence constraints, every SPL requires a variability management mechanism that must control the variability and assure that the expected products can be derived from the structure.

[1] http://www.fosd.de/FeatureVisu/

[2] http://www.rise.com.br

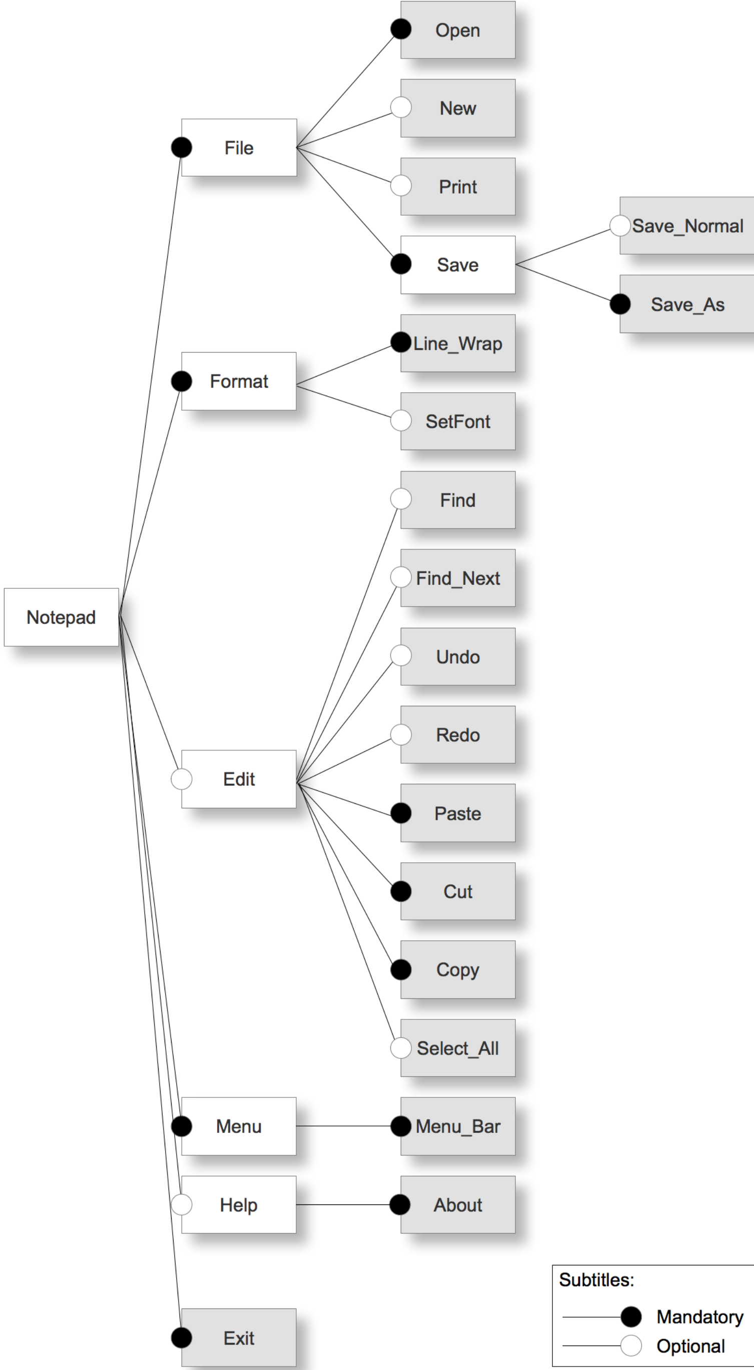


Figure 4.1 Feature Model for the Notepad SPL.

4.2.2 Variability Management

When implementing our version of the Notepad SPL, we used the Colored Integrated Development Environment (CIDE) tool (Feigenspan *et al.*, 2010) to orthogonally manage the SPL variability. The tool consists of an Eclipse IDE[3] plug-in that features manual code annotations and results in conditional compilation using preprocessors. The concerns (SPL features) are separated visually through background coloring. Moreover, different views can be used to identify scattered functionalities that may not be modularly thought of during design stage.

The CIDE tool allows programmers to annotate blocks of code concerning each of the previously defined features. Once the feature-related code is annotated and the tool is executed, the tool provides a feature selection window through which the product configuration is possible. Every feature contains a corresponding checkbox, and the mandatory ones are set as true by default. When a constraint-related feature is selected, the software automatically provides checks to resolve conflicts. A new product is generated once all the desired features are selected and the code corresponding to that given set of features is compiled.

Notepad SPL was developed without proper documentation to support its decisions. Besides the feature model, the only artifact that could be used for analysis and possible improvements was the source code with sparse comments. For this reason, maintaining such environment mainly as regards to structural improvements may represent a challenging task. Having a well-defined and updated specification is important not only to make the understanding of the overall system significantly easier, but also to enable the evolution of the artifacts.

Prior to extracting the SPL architecture, we were required to clearly understand what the scope of the SPL was. Thus, we produced the Product Map artifact, which aids in the description of the system scope through of the definition of the products to be derived with regards to their features.

4.2.3 Product Map

The products supported by the SPL should be the main focus for the organization and development activities. They can be depicted in the form of a table containing all functionalities that exist in each product, as showed in the Product Map (Table 4.1).

It is important to mention that there are many composition possibilities for the feature

[3] http://www.eclipse.org

model in question. In a company context, for example, there may be hundreds of products to be derived. In this work, 10 representative products were selected for the sake of showing the product map artifact and the variability implications to the architectural design (shown in the next sections).

Including the feature map artifact is useful for stimulating new product ideas and for explaining the difference between different products in the SPL. A customer, for example, can easily acknowledge the current products developed and request the inclusion or exclusion of features, if that is the case.

For our product map, all 18 features were aligned and crossed with 10 products within the scope of the Notepad SPL. The rows with a shaded background represent the mandatory features. As we were concerned about different composition of features and their effects to the architectural of the SPL, we set the following attributes to the products:

- P1 contains only the basic (mandatory) functionalities;
- P10 carries all functionalities supported in the SPL, including all optional features; and
- The remaining products randomly include features.

Having products with different feature compositions may represent different requirements with respect to architectural design. That is, in a component-based architecture, different features are likely to be related to different components. For this reason, the representative products were selected, and their architectures are expected to differ.

4.3 Architecture Recovery

The process of architecture recovery refers to the extraction or completion of architectural information about a software system. The main output of such process is an architectural model that represents the system components and their relations. As far as the case presented in this work, the output is a component model that supports all products in the scope of the SPL.

It is known that the architectural artifacts are often set aside when developing software, especially within small projects. Either because there is not much time available to deliver the product, or when developers do not realize the importance of structural specifications. There are a few reasons why an architecture recovery should be performed in these scenarios, such as: (i) to improve communication between stakeholders regarding the

Table 4.1 Notepad SPL Product Map.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
About	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Open	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
New	.	.	✓	.	.	.	.	✓	.	✓
Print	.	.	.	.	.	✓	.	.	✓	✓
Save	.	.	.	✓	✓	.	.	✓	.	✓
Save as	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Line wrap	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Set font	.	✓	.	✓	.	.	✓	.	.	✓
Find	.	.	✓	.	.	✓	.	✓	.	✓
Find next	.	.	✓	.	.	✓	.	✓	.	✓
Undo	.	.	.	✓	✓	.	.	.	✓	✓
Redo	.	.	.	✓	✓	.	.	.	✓	✓
Paste	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cut	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Copy	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Select all	.	.	✓	.	✓	✓	✓	.	.	✓
Exit	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Menu bar	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

system; (ii) to ease the process of system understanding for new employees; and (iii) to enable system improvements at architectural design level.

When there exists a thorough comprehension, the system can be properly managed and points of improvement may be identified at the architectural level. In the context of SPLs, it becomes easier to provide valuable information for application developers and maintainers to include new products or modify the existing ones.

One way to perform the recovery of PLAs, is considering the available artifacts, such as source code, design artifacts and domain knowledge to be manually and automatically analyzed (Eixelsberger, 1998). The architectural attributes can be specified in an architectural description language such as xADL (Dashofy and Hoek, 2002b), which allows automatic transformations and derivation of product specific architectures.

Nevertheless, the process might be error-prone. The recovered architecture can be inaccurate and a result of misinterpretations due to a number of factors that are involved in the process, including (i) different stakeholders' expertise levels, (ii) high complexity of code artifacts, (iii) impossibility of involving the project's original architect and/or software engineers for validation, and (iv) human subjectivity. We attempt to

minimize those risks by recovering a system that is already familiar, and involving different researchers for validating the outcomes.

As previously mentioned, the major difference of PLAs when compared to single systems' architectures is their purpose. While a system architecture concerns the design decisions of one system in particular, a PLA is required to depict all systems to be derived from the SPL. In other words, the PLA must sustain the design decisions of a range of different products through a software infrastructure. In order to recover the architectural design decisions of Notepad SPL, all available resources were used: the source code, the domain knowledge, and the available documentation of the original project. The recovery process is described in the next subsection.

4.3.1 Recovery Process

In order to obtain the architectural model of Notepad SPL, we followed the workflow presented in Figure 4.2. The first block represents the artifacts that were available for us to run an analysis in. The source code was available through our research group, since we were responsible for adjusting the features granularities and adding a variability/-configuration management mechanism at implementation level. Along the development process, we obtained the required expertise in the domain, which had led us to compose a feature model in such granularity and interaction constraints. The original documentation was also used as means to understand the implementation rationale. By documentation we mean the published article (Apel and Beyer, 2011) and the limited comments in the source code. The article presented very little description of Notepad, which has driven a lot of our effort towards analyzing the actual code.

4.3.2 Automated Analysis

The second block represents the artifacts obtained from both automated and manual analysis of the available artifacts. The automated analysis was performed using Structure101[4], which is a tool that receives the source code as input and provides a number of models as outputs after a lexical interpretation. Among the different types of representations in a higher abstraction level, we found it relevant to consider the decomposition view and the dependency view, which are shown in Figure 4.3 and in Figure 4.4.

The tool outputs a decomposition model showing the hierarchy of the entities. The implementation of Notepad SPL is supported by two main entities that orchestrate the

[4]http:www.structure101.com

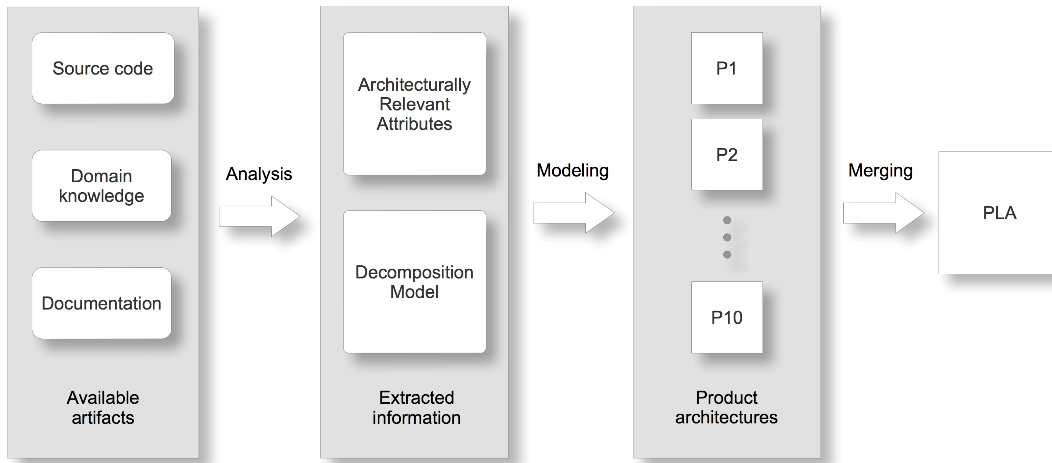


Figure 4.2 Architecture Recovery Process adapted from (Taylor *et al.*, 2009).

main procedures and GUI actions. Among them, the features are implemented and a trigger is set to generate a Java window with the product specified with code annotations. In this case, the system was implemented by separating two entities to specifically handle files and font customizations. Those structures are called by the Actions entity, which carries all functionalities and provides the selected functionalities to the product assembler Notepad.

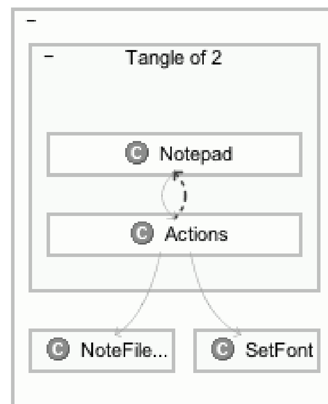


Figure 4.3 Decomposition View of Notepad SPL.

Figure 4.4 shows a dependency graph that explicates the relation between the core entities of Notepad. The constructor mechanism and the functionalities are organized within this core. The implementation technology requires specific libraries to implement functionalities, such as Undo/Redo and File handlers. The structure of the implementation was then separated to cover such needs and still maintain modularity. From the figure,

it is clear that the core entities are closer linked between them when compared to the functionalities provided by those libraries. The weighted arrows represent procedure calls, and indicate the level of dependency of each entity in respect to the related entities.

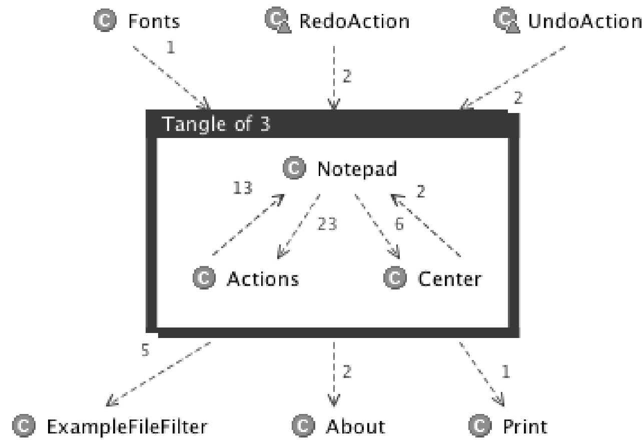


Figure 4.4 Dependency View of Notepad SPL.

4.3.3 Manual Analysis

Regarding the manual analysis, we were guided by the documentation and also considered domain knowledge to evaluate the source code. In order to understand the implemented mechanisms, we provided several different configurations through CIDE's interface, thus generating 10 different products that were previously presented.

Since we were interested in aspects that were relevant to the architecture, we searched the code for aspects that allowed the specification of Components, Connectors and Interfaces. For these terms, we maintain their definitions as described in Section 2.3.

The search was undertaken in the source code, and considered all characteristics within the concepts and the available artifacts from analysis. Since the code implemented all possible features in the SPL, the resulting specification refers to the SPL as a whole. We identified the components, interfaces, and specified the methods related to each interface. The specification is attached in Appendix B.

The features implemented in the Actions component are manually annotated at development time. A number of features require core handlers that are provided by pre-set Java libraries, such as the ones related to the Files and Fonts components. Derived products are instances of Notepad, which implements listeners for events provided through the GUI interfaces. Annotated code inputs the CIDE component, which in turn provides a valid configuration in a XML file according to previous feature modeling.

The configuration is built at run time, based on the selection of features through CIDE's graphics interface. Only the product configuration code specified in the XML is considered for derivation. The event-based interface provides control from the Java GUI libraries as the Notepad contains direct procedure calls to the features implementation within the Actions component. The code is conditionally compiled according to the annotations and the features selected by the user.

From the specification it was possible to identify components that were related to specific functionalities offered as features. For example, the Undo/Redo component is directly related to Undo and Redo features. Based on the product map, such analysis aided in the development of product specific architectural knowledge. The generated mapping between products and architecturally relevant elements (components) is shown in Table 4.2.

Table 4.2 Notepad SPL Variability Points.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Actions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Notepad	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Undo/Redo	.	.	.	✓	✓	.	.	.	✓	✓
Fonts	.	✓	.	✓	.	.	✓	.	.	✓
Files	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
GUI	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Through the mapping of components and products, we conclude that the only variable architectural entities are Undo/Redo and Fonts. The shaded rows represent components that are mandatory, as they realize features that are present in every product. In this sense, the architecture model of each product will contain at least the four components (Actions, Notepad, Files and GUI), and the Undo/Redo component will be added in the case of P4-5 and P9-10. Also, the Fonts component will be added in the case of P2, P4, P7 and P10.

4.3.4 Merging Product Architectures

Such incremental style of composing the architecture is used when focusing on the product specifications. The resulting PLA is the composition of all 6 components, as presented in Figure 4.5. It respects the rules that are determined by the components and interfaces as well as corresponds to the features and the variability management

mechanism implemented in the source code.

Component interactions are defined by the connectors, which in turn determine the type of communication in terms of required and provided interfaces.

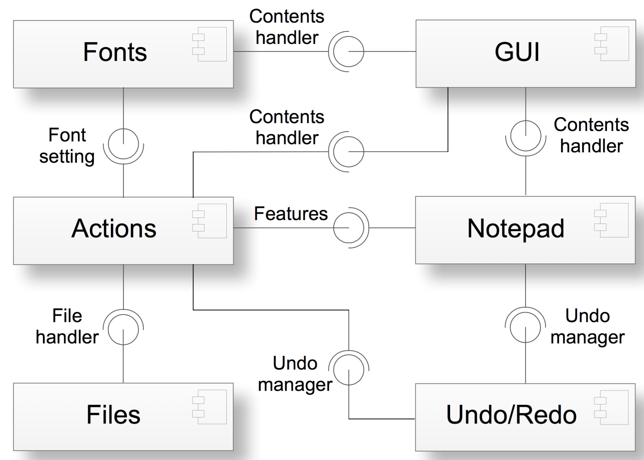


Figure 4.5 Notepad SPL Component Model.

For deriving a product, the PLA is reduced through a process called selection. In the case of Notepad SPL, the selection is managed by CIDE and the composition of features is verified by the XML specifications.

Recovering architectures represents a difficult task especially in the context of SPLs due to the variability involved in each step of the process. Instead of straightforwardly analyzing the code and specifying the system in terms of it's underlying structure, individual features and their composition rules need to be considered.

4.4 Identifying Architectural Bad Smells

From the recovered Notepad SPL architectural model, we manually searched for smells and discussed their implications in the SPL context. We also propose a new type of smell that is present in the current Notepad SPL specification. We maintain the definitions of components, connectors and interfaces as previously discussed. Next, we describe the identified smells and their existence is highlighted with respect to the impacts on quality attributes.

4.4.1 Connector Envy

Components with Connector Envy cover too much functionality as regards to connections. Instead of having the interaction facilities delegated to a connector, the components encompass, to a great extent, one or more of the following types of interaction services: *communication*, *coordination*, *conversion* and *facilitation*.

As previously mentioned, the Actions component contains the implementation of the functionalities, as the Notepad component triggers what is displayed in the text editor window after CIDE's validation. We identified the connector envy smell in the *Notepad* component, due to its extensive coordination and communication links to other components. *Notepad* contains an operation that transfers control set by the provided events to the *Actions* component through several conditional statements. The direct method invocation represents a negative effect to the component's reusability in that the dependability of these components is increased.

Despite the fact that ideally each feature should be considered a component, the granularity of these functionalities' specifications is set by the simplicity of the operations. By grouping fine-grained features into one architecture entity, different functionalities may demand diverse interfaces to realize their requirements. In the case of *Notepad* component, it requires both *Undo/Redo* functionalities and feature implementations in *Actions*, which can be translated to intense communication requirements between these components. The *Notepad* component imports several libraries that are used to implement the graphical interface of generated products, and also implements a listener of events, which are sent through GUI interactions.

Furthermore, the *Actions* component presents high communication and facilitation services by interacting with every other component in the system. In order to improve the component's maintainability, it is possible to designate the connection attributes to a specialized entity. However, in this case, the performance of the operations could be affected, since all features are concentrated within the functional part of the component.

Combining product construction capabilities and connector responsibilities represents a reduction in testability because application functionalities and interaction functionalities cannot be separately tested. In an SPL, it should be possible to perform tests both in product-specific architectures (e.g., through scenarios) and in the PLA for assessing the variability mechanism and derivation procedures with respect to quality attributes.

Understandability is also compromised due to the unclear separation of roles and concerns. On the other hand, providing an interaction mechanism would imply in the deployment of new processes, as well as increment the overall complexity of the system.

4.4.2 Scattered Parasitic Functionality

This smell is characterized by the existence of a high-level concern that is realized across multiple components. That is, at least one component addresses multiple concerns, which makes the smell a bottleneck for modifiability. If all components implementing a scattered concern were to be composed, the resulting component would realize orthogonal functionalities.

We were not able to find scattered functionality evidence in Notepad SPL. Even though the GUI concern implements the triggered GUI attributes in *Actions* and aids in constructing the visual components orchestrated by *Notepad*, it refers to the Java native library to handle interfaces. Since it does not represent a high-level concern, changes in such related components would not represent a widespread impact.

The main class responsible for specifying the product, Notepad, inherits the properties of JFrame class. Its constructor sets the default window settings that will be composed with the valid configuration received from CIDE at development time. The *Actions* component is notified about the user graphics interface events and realizes the required functionalities via a direct procedure call from *Notepad*. In this sample case, these two components would be required to be refactored and such modifications are not complicated. However, the cost requirements to change aspects in a shared concern of bigger projects might prevent such changes to be made.

In SPLs, scattered functionalities can represent aspects that are difficult to realize, given the widespread effect among products. In a shared concern environment, the understandability is compromised due to orthogonal contexts being realized in a single component. Feature granularities should be adjusted at specification time in order to take advantage of each component's reusability levels. That is, having coarse-grained features can elevate the chances of specifying a single component with multiple concerns. On the other hand, fine-grained features can induce architects to realize errors during specification, for instance, having similar functionalities under different components.

4.4.3 Ambiguous Interfaces

When a component offers only one single and generic entry-point, such interface is referred as ambiguous. Ambiguous interfaces reduce static analyzability and can occur independently of the implementation-level constructs that realize them. Despite the fact that the component may offer and process multiple services, an ambiguous interface will offer only one public service or method.

This smell results in reduced understandability because an ambiguous interface does not reveal which services a component is offering. In other words, a user of this component needs to inspect the implementation of the internals before using its services. In the case of Notepad SPL, this smell is characterized in the *Actions* component and its relations. All features are implemented in this component and accessed through one generic action listener.

The event-based parameterization presents a dynamic dispatch despite the conditional compilation imposed by CIDE. Such scenario is usually motivated by the implementation of generic GUI listeners. In the case of Notepad, the triggers are executed through the user relation with the buttons and text pane area. This type of solution can be implemented in small SPLs, but might cause complications when understanding and analyzing the feature entity. On the other hand, with such centralized solution, new features can be more easily added and specified respecting the limits of one component and obeying a single connecting point.

We argue that this smell is common in SPL engineering due to its intrinsic characteristics. The common implementation and composition of features is given by one single repository of features that obey a rule at the time of composition and product derivation. This solution would probably carry a number of down points in the case of a large SPL supporting hundreds of features, for example. Such a generic entry-point could be overloaded by functionalities that differ too much, thus require very different handlers for effective management.

4.4.4 Extraneous Adjacent Connector

This smell is characterized by the use of two connectors of different types to link a pair of components. For example, the use of procedure calls makes the transfer of control explicit, thus understandability is increased. On the other hand, event connectors can be easily replaced or updated because senders and receivers of events are usually unaware of each other. In this case, reusability and adaptability are increased.

However, having two components with both connector types may affect understandability, because it becomes difficult to determine whether and under which circumstances additional communication occurs between the affected components. The side effects of using both connector types should be assessed, since the beneficial effects of each individual connector might cancel each other out. Although we focused on the combination of procedure calls and event connectors, this smell also considers other types of connectors.

In our sample SPL, the *Actions* component provides functionality services to construct

a Notepad instance (product). A procedure call would represent the occurrence of this smell due to the use of event listeners to trigger the execution of features. However, the only interaction between Notepad and *Action* components is through events provided by the GUI. Thus, we were not able to identify the extraneous adjacent connector smell in Notepad SPL.

Although it may not be specific to PLAs, the scenario of extraneous adjacent connector could be found in an SPL implementation that breaks the unique connecting restriction set by a given PLA. For instance, if an event bus is designed to handle communications between an instantiation entity and a feature repository, there should be no direct procedure call between these two components. Such restriction can be assured by setting protective rules to access the repository, either through parameterization or strict procedure call policies.

4.4.5 Feature Concentration

This smell is specific to the SPL context, and is characterized by the centralization of the SPL features in one architectural entity. It is related to the ambiguous interfaces and scattered functionality smells in that different functionalities are implemented in a single construct, which might offer only one generic entry point.

Design decisions are usually driven by metrics, such as coupling and cohesion between different entities. For example, in a store management system, the employee registration and payment modules should be closely related in the architectural design because both of them represent human resources concerns. At the same time, the inventory should be handled by different instances since it is not closely related to these modules.

Such design solution might not be obvious in an SPL environment when the feature granularity is too fine grained, because it would require an effort to accommodate different concerns in different design entities. A generic solution would be to group all features into one entity and implement a configuration management mechanism that will derive products according to previously defined rules. The decision to ignore different concerns at design time represents an easier development, but also negatively affects the system understandability and modularity attributes.

As previously mentioned, the *Actions* component concentrates the features in one component through a single generic entry-point, which characterizes both feature concentration and ambiguous interfaces smells. In the case of Notepad SPL, such decisions are supported by the fact that it is not a complex implementation, so understandability is not a major concern in the project. However, we argue that making the same design

decisions in an SPL with several hundreds of features would represent severe impacts on this PLA's understandability and changeability throughout its lifecycle.

4.5 Threats to Validity

The threats to validity of this work are described next.

We understand that many software projects do not include architecture documentation, which makes it difficult and costly for one to investigate the occurrence of architectural smells. We minimized this threat by describing an architecture recovery process which allows the smell identification process to be carried out.

For the architecture recovery process, we designed the PLA based on the analysis of the source code and adjacent artifacts. Recovering architectures from code represents a difficult task through which important aspects may not be properly captured accurately, such as coupling and cohesion. We minimized the threat of misconceptions in defining the architecture by carrying out discussions with developers involved in the implementation of our Notepad SPL version. However, we understand that many times the developers are not available.

Despite the fact that several studies had been undertaken in the area of code smells, the concept of architectural smells is fairly recent. In order to maintain the transparency of the process and concepts, we throughly described what are smells and how they may influence in an SPL project.

The identification of smells was originally proposed to be carried out in single systems. Thus, identifying bad smells in PLAs may represent a threat. The feasibility of using such technique in the context of PLAs is also considered in this work.

We limited the discussion on a set of five smells, although additional ones may occur. In order to minimize this threat, we intend to further investigate this issue and possibly discover other smells. In this study, we only took into consideration one SPL. However, in the next chapters, we minimize this threat by considering another one.

4.6 Chapter Summary

This chapter presented the exploratory study, which was conducted to characterize the phenomenon of architectural bad smells in SPLs. We selected a sample SPL in the text editor / desktop domain, and extracted its architecture through a recovery process.

For the recovery process, we considered both manual and automated analysis:

- For the automated analysis, we ran a supporting software - having the source code as input - to aid in the understanding of the software structure, and how the messages are passed between subsystems;
- For the manual analysis, we considered the source code and the available documentation, including variability management, and architecturally relevant information to built the architecture.

Then, we analyzed the architecture and the code in order to identify smells. We described where the smells were found and what type of effect their occurrence would represent to the PLA's lifecycle. In addition, we proposed a new smell - Feature Concentration - which has been identified from the analysis.

Next chapter presents a replicated study considering an SPL in a different domain. The idea is to follow the same procedures adopted to perform the present exploratory study in order to obtain and compare results.

5

A Replicated Study on Architectural Bad Smells in Software Product Lines

In the last chapter, we obtained evidence that architectural bad smells also occur in SPLs. Moreover, the smells can be identified using the same method initially proposed to the context of single systems. In this chapter, we report on a replicated study with an SPL in a different domain.

This chapter is organized as follows: Section 5.1 presents the study setup, including the research question and the conducted workflow. Section 5.2 presents information about our object of study: RescueMe SPL. Section 5.3 describes the process followed to perform the recovery of its architecture. Section 5.4 describes the process of identification of smells from the extracted architecture. Section 5.5 describes the comparative analysis considering both studies performed. Section 5.6 presents the main findings of this study. Section 5.7 presents the threats to the validity of this study. Finally, Section 5.8 presents a summary of this chapter.

5.1 Study Setup

In the last chapter, we presented an exploratory study that aimed at characterizing the architectural bad smells phenomenon in the context of SPLs. We were able to draw initial conclusions from the analysis we carried out in a PLA in the text editor / desktop domain. As next step, we performed a case study aiming to replicate the procedures conducted in the exploratory study. By replicating the last study in a different domain, the idea is to obtain more evidence to be compared and analyzed towards a better understanding of smells in PLAs.

According to Carver (2010), there are no existing guidelines for reporting replications.

However, the author claims that a replicated study should contain (i) information about the original study, (ii) information about the replication, (iii) a comparison of results to the original and (iv) conclusions drawn across the studies. The complete original study can be found in the previous chapter of this dissertation, as well as a brief discussion on its results. This chapter reports on the motivation for conducting the replication, as well as describes the steps performed to replicate the original study. After describing the procedures in the search for smells, we report on a comparative analysis that considers the characteristics and results of both studies.

In summary, the present study aims at replicating the exploratory study, presented previously, through the main research question: "***Do architectural bad smells occur in software product lines?***". We are particularly interested in investigating whether the same smells found in the text editor / desktop domain can also be found in the emergency / mobile domain. In addition, we present a comparative analysis considering the two case studies, and further discuss the results. Thus, the findings from our replication study increase the generalizability of the initial results by providing (i) the same smells identification strategy with a PLA in a different domain; and (ii) comparing results from both studies.

In order to address the aforementioned goals, we selected a sample SPL, that has been developed by students in our research group. Since there were not sufficient architecture artifacts related to the project, we were also required to extract its component model prior to the search for smells.

The study workflow is similar to the one conducted in the exploratory study, with the addition of a comparative analysis. It consisted in:

(1) analyzing a sample SPL in the domain of emergency / mobile;

(2) extracting its conceptual architecture from the code;

(3) searching for smells in the recovered architecture; and

(4) analyzing and comparing results from both studies.

Next, we describe the process in detail, starting with a description of the sample SPL.

5.2 RescueMe SPL

The object of this study is RescueMe SPL, an Objective-C implementation of a set of applications in the mobile domain. The scope of RescueMe SPL is to develop products

Figure 5.1 Screenshot of the RescueMe app main screen.

for the Apple iOS platform[1], which runs on Apple smartphone devices and tablets. The development team was comprised of one post-doctoral researcher, five Ph.D. students, two M.Sc. students and two B.Sc. students, all members of RiSE research group.

RescueMe products aim to aid users in emergency and dangerous situations. In summary, the application contains a red button which triggers pre-set messages to be sent to a list of contacts using SMS (short message service), Email or Social networks (see Figure 5.1). When the message is sent through the latter, the recipients are able to track the sender's location in a map.

The RescueMe SPL release contains a set of 5 different products, 22 features in total, and can be obtained from a repository[2], as an open-source project. The entire project contains 75,928 lines of source code, including the 3rd party code blocks, Facebook SDK and Message UI Framework. The application is arranged in 310 files and 303

[1] http://www.apple.com/ios/
[2] http://svn.code.sf.net/p/rescueme-spl/code-0/trunk

classes. With respect to the blocks that are strictly built for the application - without supporting services - there are 2,504 lines of source code, divided into 28 files and 28 classes. Products derived from RescueMe SPL contain from 5 to 16 features in their composition.

It is important to note that in this work we consider the first release of RescueMe SPL, since the second release is currently under development. The original planning includes more features and different configurations for the products. However, we decided to only work with the features that are fully implemented and functional.

5.2.1 Feature Model

The features and sub-features implementation was carried out using Objective-C language with XCode IDE[3] version 4.6.2. The features in the feature model were grouped according to the user's point of view, which were distributed among developers to be implemented.

Features in the feature model obey the layers of abstraction presented earlier: the inside nodes (except the root) represent a higher level of abstraction, responsible for grouping and organizing the features from the user perspective. All features on the tree represented as leaves refer to selectable functionalities when deriving a product. Solid circles connecting to their parents represent mandatory features, and hollow circles represent optional features. The tree is presented in Figure 5.2.

In this SPL, there are 2 dependencies with respect to feature inclusion: when selecting "*Twitter Destination*", the user must also include "*Twitter Import*"; also, when selecting "*Facebook Destination*", the "*Facebook Import*" feature must be included. This occurs because the importation of contacts is necessary to the composition of Twitter and Facebook messages, although they were implemented separately. In RescueMe SPL, there are not cases of feature coexistence restrictions (i.e. forced exclusion of features due to the selection of a given feature).

5.2.2 Variability Management

Variability management in the RescueMe SPL project is characterized by conditional compilation. Such technique is realized through XCode IDE's macro definitions, which represent support for pre-processor directives. In the IDE, each feature is represented by a macro with the same name.

[3] http://developer.apple.com/xcode

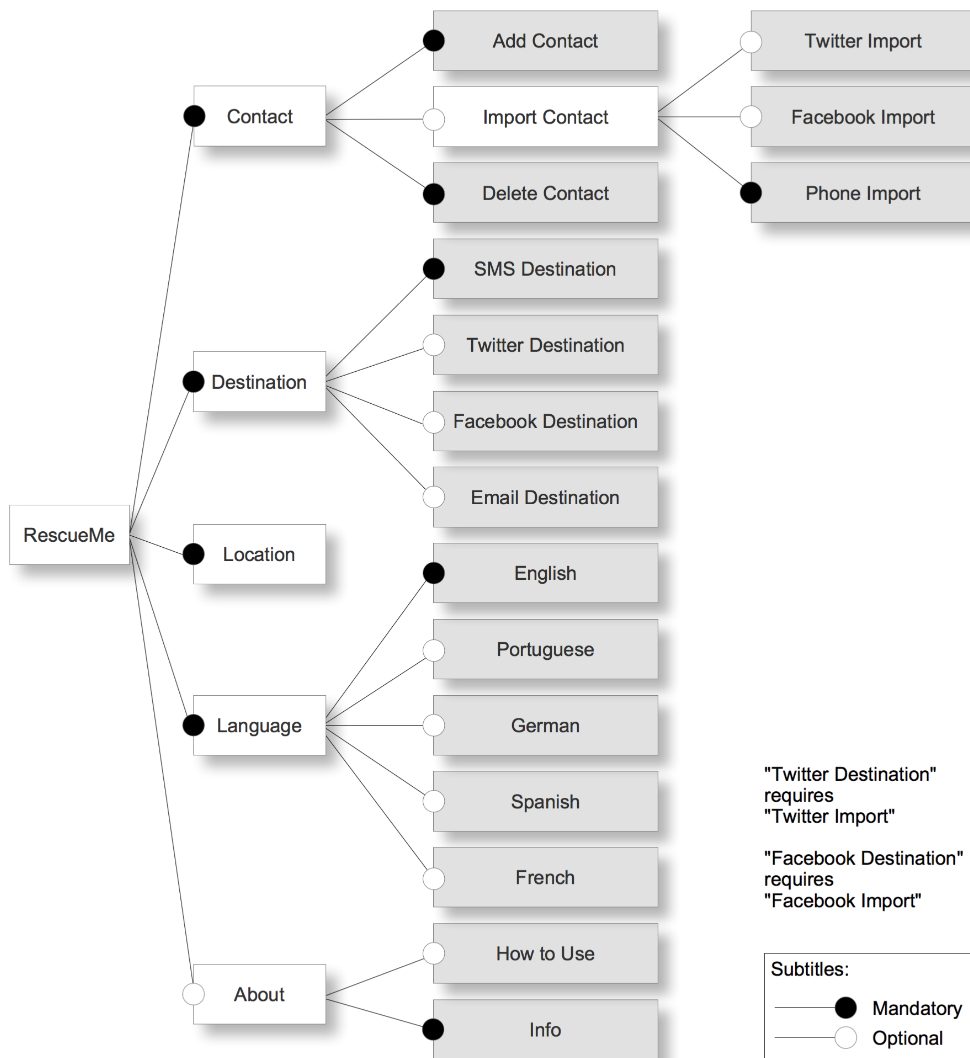


Figure 5.2 Feature Model for the RescueMe SPL.

Variability at the code level is well defined in the source code with annotations and comments. As far as documentation, the project contains some artifacts that describe what varies and how these variations behave. In addition, there are text documents concerning the management of time, risk, quality, configuration and communication.

5.2.3 Product Map

One of the available documentation artifacts was the product map, which is shown in Table 5.1. It consists of 16 features. These features are the selectable ones depicted in the feature model, i.e., abstract features are not considered in the product map. Shadowed rows represent mandatory features.

From the business rules defined in the initial stages of this SPL, there are 5 versions (or products, in the context of SPLs) that must be derived: Lite, Standard, Social, Pro and Ultimate. The first product is the simplest one, and carries only add/delete contact, sending SMS, English language and basic information functionalities. On the other hand, the most complete version of RescueMe contains all available functionality, including social networks support and multiple languages.

Table 5.1 RescueMe SPL Product Map

	Lite	Standard	Social	Pro	Ultimate
Add Contact	✓	✓	✓	✓	✓
Delete Contact	✓	✓	✓	✓	✓
Twitter Import	.	.	✓	✓	✓
Facebook Import	.	.	✓	✓	✓
Phone Import	.	✓	✓	✓	✓
SMS Destination	✓	✓	✓	✓	✓
Twitter Destination	.	.	✓	✓	✓
Facebook Destination	.	.	✓	✓	✓
Email Destination	.	✓	✓	✓	✓
English (language)	✓	✓	✓	✓	✓
Portuguese (language)	.	.	.	.	✓
German (language)	.	.	.	.	✓
Spanish (language)	.	.	.	.	✓
French (language)	.	.	.	.	✓
How to Use	.	✓	✓	✓	✓
Info	✓	✓	✓	✓	✓

5.3 Architecture Recovery

Despite the number of documentation artifacts available, at the architectural design level, only two diagrams were produced: deployment view and modules view, which are depicted in Figure 5.3 and Figure 5.4, respectively.

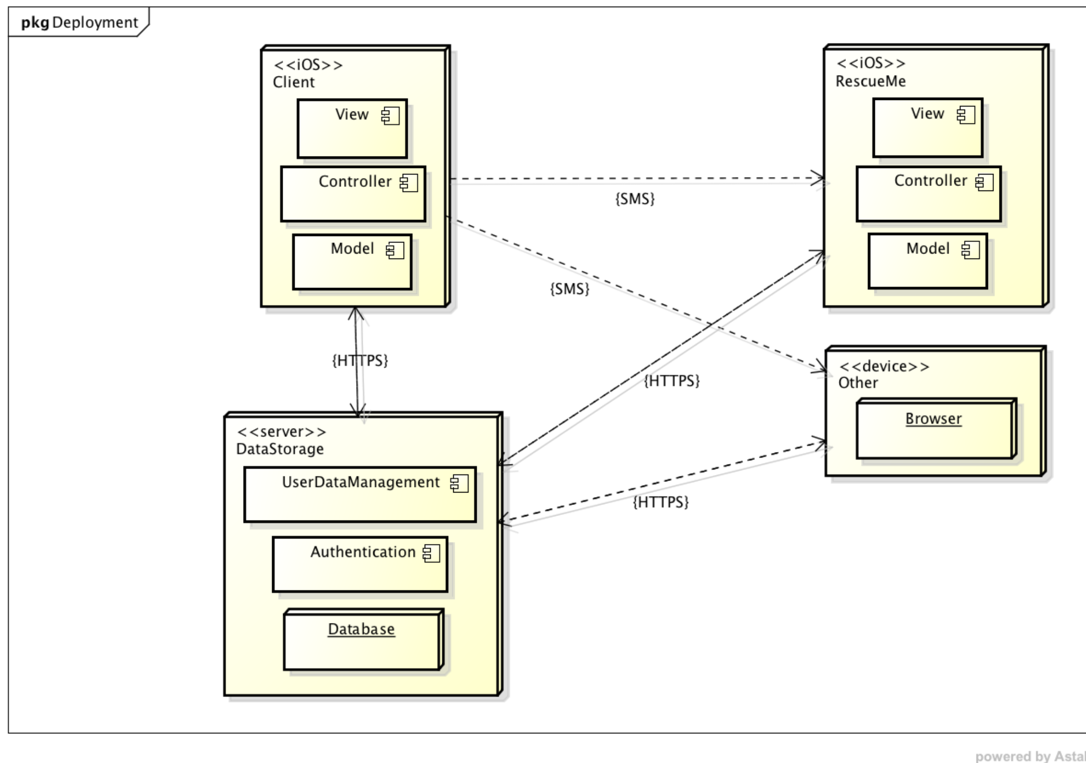


Figure 5.3 Deployment View for the RescueMe SPL.

These models represent how the project is organized as far as deployment units, using the MVC pattern. The protocols for exchanging messages are explicit, providing an idea of how the *Client*, *Storage Server* and *Devices* communicate with the central application. The MVC pattern is also used in the modules abstraction, showing the top-level composition and dependency connections between them. As noted in the figures, the consistency is depicted in the *Model* layer, and basically includes contact information. *Controller* layer is responsible for managing the consistency, as the *View* layer allows to update information through the GUI.

Although these diagrams were of high importance for the understanding of the project, they are not sufficient for the search for smells. As mentioned earlier, the search for smells assumes the existence of a component model, as well as a component specification.

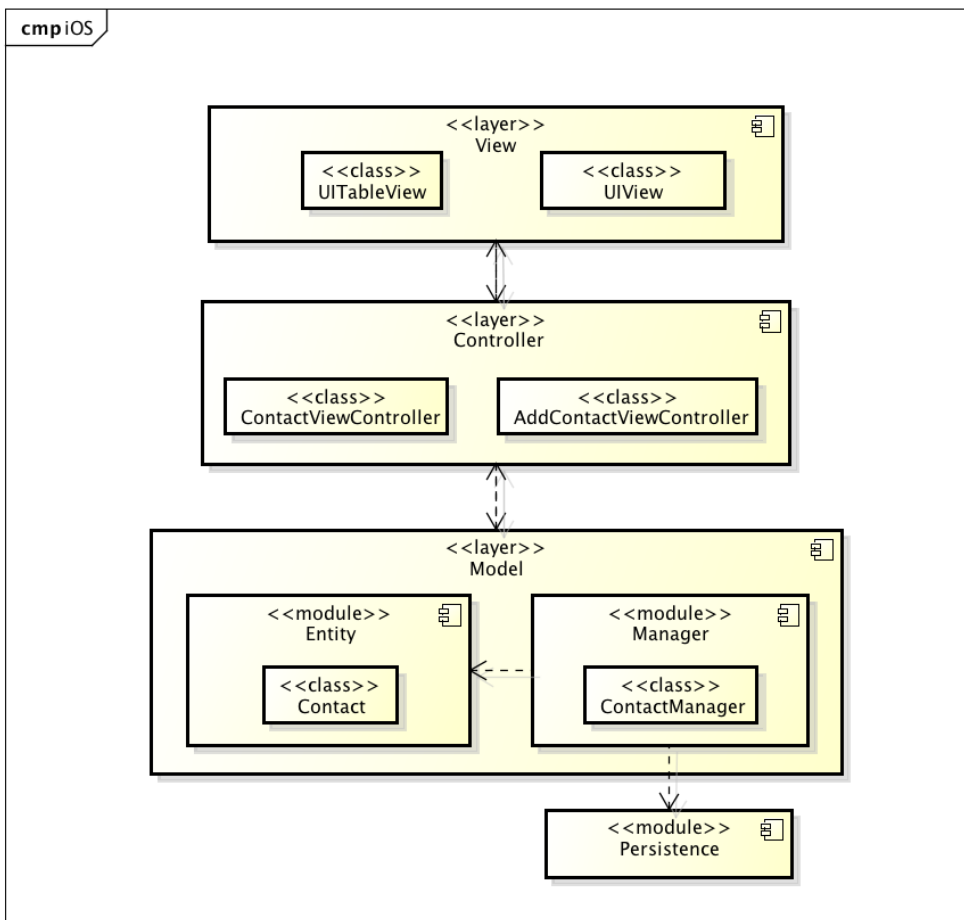


Figure 5.4 Modules View for the RescueMe SPL.

Thus, prior to the search for smells, we were required to also recover the architecture of RescueMe SPL. We explain such process in the next subsection.

5.3.1 Recovery Process

In order to recover the architecture of RescueMe SPL, we followed the same process that was described in the last chapter, depicted in Figure 4.2. As mentioned before, the source code and some documentation artifacts were available from within our research group. The domain knowledge was obtained from meetings with the developers and from studying the documents.

After analyzing the source code and the available documentation, we extracted attributes that are relevant to the architecture (from the automated analysis) and the decomposition model (from the manual analysis).

5.3.2 Automated Analysis

We were not able to use Structure101 tool to perform the automated analysis due to the lack of support to the Objective-C language. Thus, the analysis was performed using Understand[4], whose developers kindly provided us with product licensing for our research project. The tool takes the source code as input, and provides diagrams, different types analysis and metrics as outputs. It supports several languages and describes the dependencies contained between the project modules.

For the understanding of the project, we found it relevant to obtain a number of graphs that consider internal and external dependencies, procedure calls and hierarchical structures, as shown in the figures.

Figure 5.5 shows a graph which includes both header and implementation classes. The weighted arrows represent message passing between classes. The numbers represent how many times procedure calls take place. From the figure, it is evident that *Contact-Manager.h* is constantly called, as this class is responsible for selecting/updating contacts, regardless of their type (phone, facebook or twitter contacts). Relationships between the implementation class and its corresponding header are also evident.

Figure 5.6 shows a diagram depicting the calls between entity clusters, while maintaining the original project's directory structure. The lines represent calls, and the layers represent the hierarchical level in which they are organized. Despite the classes created

[4] http:www.scitools.com

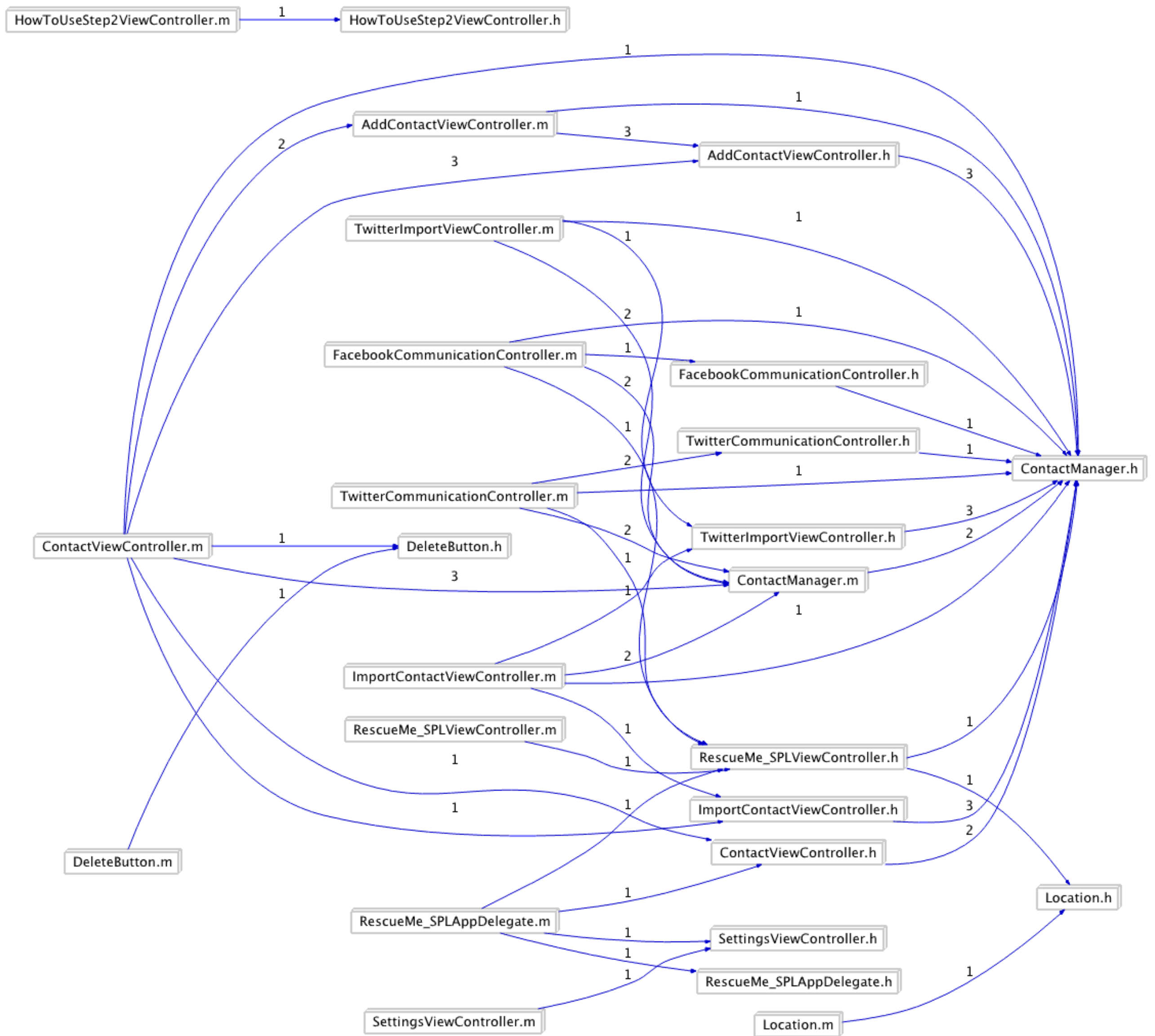


Figure 5.5 Internal Dependencies Classes of RescueMe SPL.

to implement functionalities of RescueMe, the diagram shows their relationships with 3rd party clusters and external dependencies.

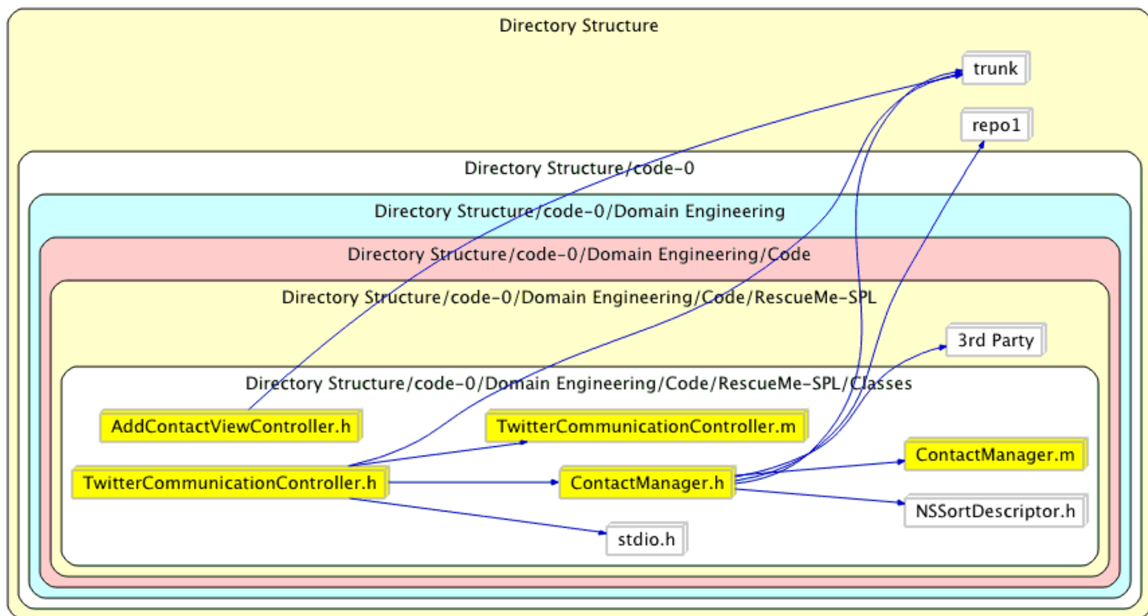


Figure 5.6 Cluster Call Graph of RescueMe SPL.

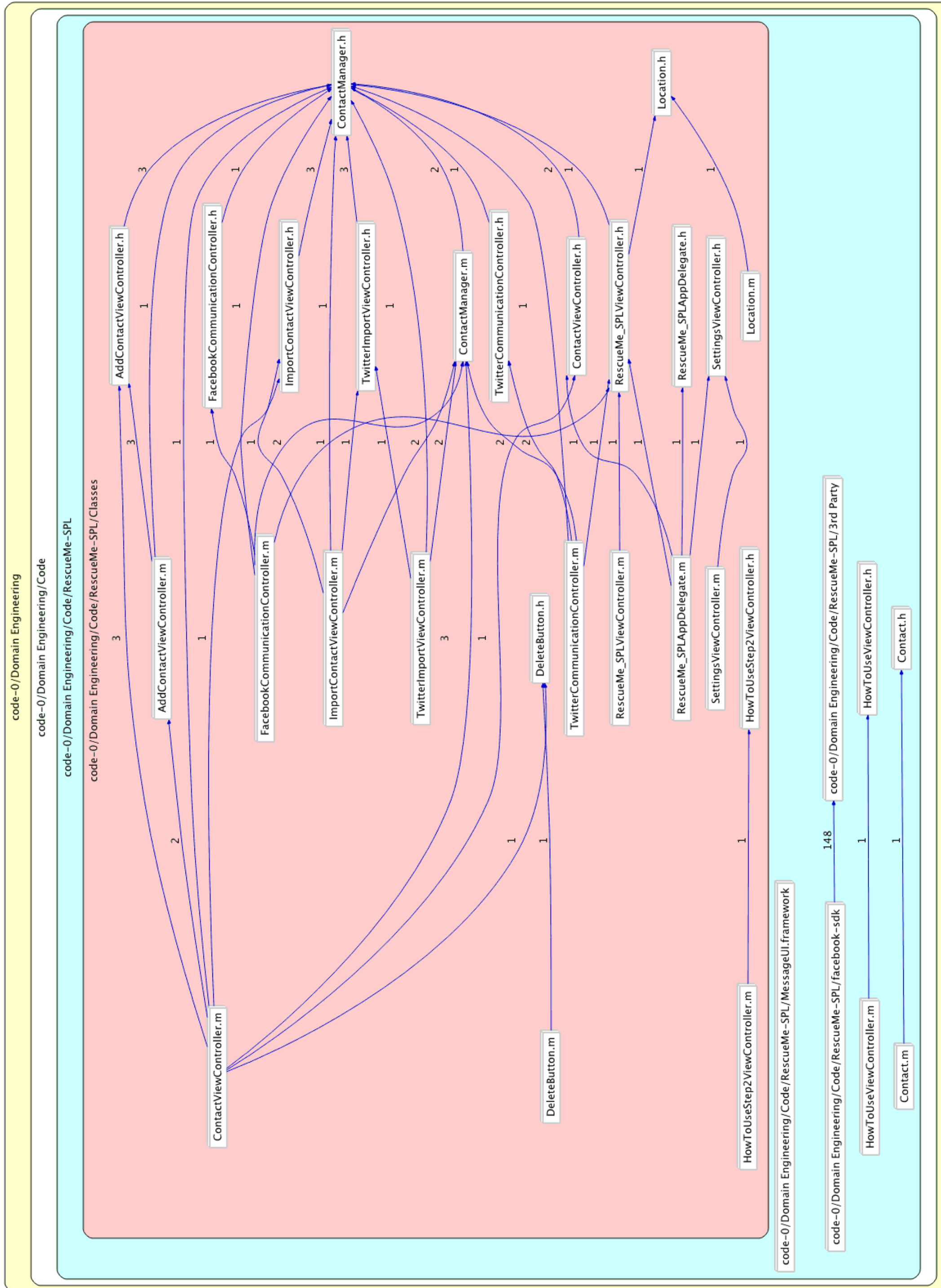


Figure 5.7 Hierarchy Internal Dependencies of RescueMe SPL.

Figure 5.7 shows the same internal dependencies, but this time considering more details of 3rd party application blocks, still obeying the hierarchical layers. The diagram provides an overview of the dependencies of the entities according to the layer in which they are allocated. Relations between classes of different layers are not depicted here.

5.3.3 Manual Analysis

By analyzing the diagrams generated by the source code analysis tool, we obtained knowledge regarding the top-level configuration of the project. We then analyzed the source code, the available documentation, and the generated graphs in order to start the specification of RescueMe SPL architecture.

For the specification, we considered the source code as a whole, i.e., all features that were implemented, regardless of the *ifdefs* for product configuration. The specification is attached in Appendix C.

The configuration of products is built at development time, based on the feature model constraints and the desired product to be derived. There is no GUI interface for selecting features or products to be developed, thus the variations are realized through the code annotations using pre-processors.

In the case of RescueMe SPL, we were not required to select random features and generate products for building their architecture and then merging them to form the PLA. The products had been already defined and their scope was clearly stated by the original documentation and code. Thus, we realized the specification of components and crossed them with the 5 individual products, to find out the variability points. The generated mapping between products and architecturally relevant elements (components) is shown in Table 5.2.

Table 5.2 shows the components recovered from the project's source. Shadowed rows represent components that are mandatory, i.e., components that exist in every product derived. In this sense, every product (or version) developed from this SPL will contain such components. The only two components that are optional in RescueMe SPL are *Facebook API* and *Twitter API*, which are present in versions Social, Pro and Ultimate. These components are required for obtaining contacts through a connection with social networks, by using the services they provide. Optional features that are not related to the connection with social networks are implemented within the components that must be included in every product derivation.

Table 5.2 RescueMe SPL Variability Points

	Lite	Standard	Social	Pro	Ultimate
Contact	✓	✓	✓	✓	✓
Location	✓	✓	✓	✓	✓
GUI	✓	✓	✓	✓	✓
Contact Manager	✓	✓	✓	✓	✓
Contact View Controller	✓	✓	✓	✓	✓
Destination	✓	✓	✓	✓	✓
Message Composer	✓	✓	✓	✓	✓
Facebook API	.	.	✓	✓	✓
Twitter API	.	.	✓	✓	✓
iOS API	✓	✓	✓	✓	✓

5.3.4 Merging Product Architectures

As previously conducted, in order to form the PLA, we merged the individual products' architectures, according to the extracted specification. The resulting PLA is the composition of 10 components, as shown in Figure 5.8. The model respects the rules that are determined by the functionality and connection attributes implemented in the source code. Component interactions here are also defined by the connectors, which in turn determine the type of communication in terms of required and provided interfaces.

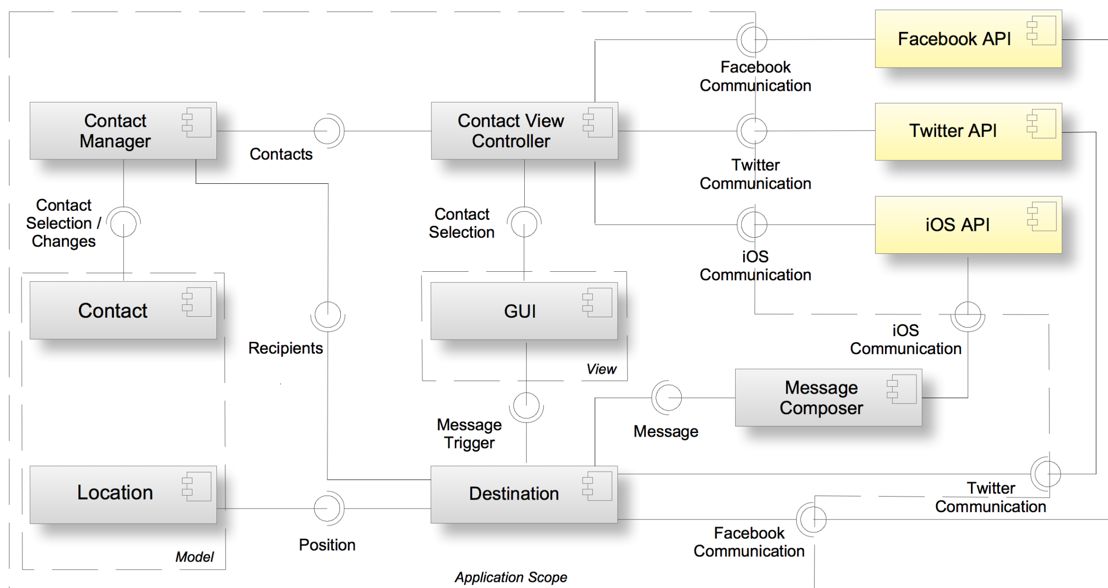


Figure 5.8 RescueMe SPL Component Model.

The component model is defined by a set of components that are divided into 4 major groups: model, view, controller, and external entities. *Contact* and *Location* components are responsible for the persistence of the application. While the first is responsible for storing contact information that will be accessed in case of emergency, the *Location* component keeps information with respect to the geographic location of the sender (application user).

In all cases, the recipients are previously selected by the user through a screen provided by the *GUI* component. Users are able to select the contacts who are going to receive the messages in case of emergency. In addition, the trigger of sending the messages is activated by the touch of a red button on the screen.

Changes in the Model components are managed by *Contact Manager*, through the Contacts interface and by *Destination*, through the Position interface. The *Destination* component is responsible for the main operations within the application. It requests a number of services that will be used for sending the rescue message: the user location, the recipients of the message, the means through which the message will be sent, the actual composed message, and the trigger for finally sending it. The message is passed by the *Message Composer* component, which builds the message according to the language selected by the user and the available version of the application. Recipients are collected depending on the type of connectivity existing in the version of the product: Phone contacts, Twitter contacts, Facebook contacts, or any combination of the three, being the phone contacts mandatory. The language in which the message is composed depends on a pre-configuration of the user preferences, when available in the application version. For the most basic versions of the application, the English language is standard, and does not require selection from the user.

A number of services are required to complete the functionality of sending the rescue message to the user's social network contacts. *Facebook* and *Twitter APIs* are part of the project when the Facebook and Twitter-related features are selected. The *iOS API* component, on the other hand, is always included, since it is responsible for providing the framework for a working iOS application. All three components are organized in such a way that a high level of abstraction is maintained for the programer to use their services. In other words, these components are not extensively accessible and changeable, as the MVC model and the iOS development constraints exist.

The component model and its components' specification have been validated with the developers. Once the model is built, we are able to search for the smells. In the next section, we report on such identification of the architectural bad smells.

5.4 Identifying Architectural Bad Smells

From the recovered RescueMe SPL architectural model, we searched for smells and hereby discuss their implications in the lifecycle properties of the project. For the characterization of the architecture and identification of smells, we maintain the definitions of components, connectors and interfaces as previously discussed. Next, we describe the identified smells and their existence is highlighted with respect to the impacts on quality attributes.

5.4.1 Connector Envy

The *Destination* component concentrates most of the functionality provided by the application. As depicted in Figure 5.8, it has connections with other 6 components, which makes the *Destination* component fundamental for the main functionality of the application: sending rescue messages. We identified the connector envy smell in the *Destination* component, due to its extensive communication and coordination responsibility. The component contains several connections to both pass execution control, and also send/receive data (messages, computational results) to/from other components.

For example, the *Destination* component requires several attributes to be obtained in order to send the rescue message (which is itself a message passed from the *Message Composer*): the recipients of the message (provided by the social networks or the contact manager of the stored information), the location of the user, and the trigger for finally sending the message.

As far as the coordination services, *Destination* carries the responsibility of submitting the message after the thread of execution has passed to and from components managing (phone and social networks) contacts, the location obtained from the local user's device, and the rescue message that is composed. Such activity includes the language in which the message is composed, and a warning in case the message was not successfully sent. However, the main passing of execution thread is passed to and from the *GUI* component. This entity is responsible for capturing user inputs and showing outputs on the screen. All of such activity, thus control of execution, is sent over the interface to the *Destination* component.

The *Destination* component was designed in a way that compromises the overall system's reusability, understandability and testability. It would be very difficult to reuse the component without carrying other components (and their services) due to the

number of connections they own. In other words, the components are not concise to be reused separately. For the same reason, the organization with respect to functionality and connections is not easy to understand. The component carries the functionality of sending the rescue message, and also manages several message passing tasks and coordination characteristics to and from other components. Another concern with the implementation of *Destination* is the lack of proper means for testing. By having such commingled concerns, unit testing is not feasible, forcing developers to test the component in conjunction with other components. Such scenario requires developers to spend considerable time in this task, and could compromise the quality of the component.

In the case of RescueMe SPL, one possible justification of such design decision is the reduced number of features to the overall project. Instead of designating a separate connector only to deal with the numerous connections required to send the rescue message, it is simpler and handier to implement a component as it is. In the context in which the *Destination* component was implemented, such design it may be acceptable also because of the low expectations of maintainability activities for this SPL. The higher priority seems to be related to the performance of the derived product, instead of planning to make the project grow and become sufficiently complex to require maintainability directives.

5.4.2 Scattered Parasitic Functionality

When the scattered parasitic functionality smell occurs, there is at least one concern that is shared between different components. We were not able to find such scenario in the RescueMe SPL. Perhaps the SPLs are defined in such a way that the components are responsible for a separate functionality, being less probable to meet several concerns at the same time. This is due to the definition of features, and the style of developing feature-oriented blocks of code.

The graphics interfaces in the application are managed by the *GUI* component, which provides the navigation screens for selecting preferences, triggering the rescue message and also selecting/updating contacts. In the case where contacts are provided by social networks, the interfaces are also provided by their APIs. We decided not to consider such scenario as a shared concern because the APIs are external to the actual RescueMe application. Further, the MVC pattern plays a role in separating the concerns and forcing developers to write code following a set of separation constraints. At the same time, the protected type of interaction between the actual application and the supporting services (iOS and Social Networks) also contribute for this smell not to occur.

By not presenting the scattered parasitic functionality smell, RescueMe SPL maintains

an acceptable level of modifiability, understandability, testability and reusability for its components.

5.4.3 Ambiguous Interfaces

Usually, systems using the event-based publish-subscribe pattern (Birman and Joseph, 1987) are susceptible to ambiguous interfaces due to the uncertainty of what type of data is exactly being passed. We were not able to find such scenario in the implementation of RescueMe SPL. Despite the fact that the *GUI* component handles a number of events occurring from the interaction with the user, all types of activities are predictable and consistent with the expected responses.

The features in the project are not concentrated into a single entity, so an ambiguous interface is not the first idea to compose functionalities in order to derive products. Further, one other characteristic that might have influenced in the clear specification of data is the organization of which the SPL is developed. The separation of concerns into model, view and controller aids in the understanding of the responsibilities of each component and what types of messages are being passed.

Another characteristic that helped in the clear role of interfaces is the fact that the APIs are intrinsically strict in their uses. Very well defined methods and attributes are available from the social networks and iOS frameworks, which reduces the possibility of confusing types of data and services that are being required and provided. In addition, such mandatory and strict interfaces ease the analyzability and understandability of the overall project. The services provided by the APIs are meant to be consistent and clearly reusable in different context, thus such intrinsic characteristic helps in avoiding such smell.

5.4.4 Extraneous Adjacent Connector

In the case of RescueMe SPL, the most extensive activity regarding events and listeners is given by the *GUI* component and its interactions. The relating components receive messages asynchronously as the *GUI* requests information from the user. We have not found any case in which such events are parallel to a set of procedure calls, making the relationship between components confusing. The implemented listeners are concerned with activities such as: the button clicking, row and contact selecting, and receive rescue message warnings. Despite the fact that the listeners are not handled by an exclusive designated connector (such as an event bus), procedure calls related to those component

relationships do not exist.

It was clear that the development of this SPL has given priority to the simplicity of having listeners and direct commands to the events that asynchronously occur. The communication between any pair of components is not ambiguous to the extent that the data or control flow are put in doubt. The *GUI* component implements listeners that are used to determine what type of changes will take place in different (managerial and functionality) components.

The overall reusability and adaptability of the involved components are increased due to the fact that those entities carry generic pre-defined types of events to be sent and received.

5.4.5 Feature Concentration

RescueMe is an SPL aiming to develop products that, summarily, send text messages to a number of recipients in dangerous situations. The architectural design under which the SPL was implemented includes a component that majorly controls the functionality of uniting the required attributes to have the messages sent (*Destination* component). In other words, the *Destination* component is essential for the functionality of every application that is derived from this SPL. However, the high-level features defined in the feature model are not entirely implemented within this component. *Destination* component is not a repository of features, although it contains most of the variable functionalities defined in the feature model.

The role of sending messages is fundamentally attributed to the Destination component. However, features such as *Twitter Import* and *German* are essentially implemented in external components (*Contact View Controller* and *Message Composer*, respectively). In conclusion, we cannot affirm that the feature concentration smell appears in this project.

It is true that if the *Destination* component were to be changed, there would be a major effect to the overall system, since this component implements key functionality to any product developed. Thus, understandability is negatively affected by this decision, as well as the modularity/reusability of the component. However, the negative impact is not as severe as it would be if all features were concentrated within this single entity.

Table 5.3 Comparative table considering the two objects of study.

	Notepad SPL	RescueMe SPL
Language	Java	Objective-C
SLOC (internal)	1,927	2,504
Domain	Text-Editor (Desktop)	Emergency (Mobile)
Variability Technique	Conditional Compilation (CIDE GUI)	Conditional Compilation (IFDEFs)
Pattern	.	Model-View-Controller
Number of Products	10	5
Number of Features	24 (18 selectable)	22 (16 selectable)
Number of Mandatory Features	13	10
Number of Optional Features	11	12
Number of Alternative Features	0	0
Granularity of Features	Very Fine-Grained	Fine-Grained
Architectural Smells Found	Connector Envy Ambiguous Interfaces Feature Concentration	Connector Envy
Affected Maintainability Attributes	Reusability Understandability Changeability Testability	Reusability Understandability Testability

5.5 Comparative Analysis

In the last section, we conducted the search for architectural smells in the RescueMe PLA. We were only able to identify the occurrence of one smell in the specification - the connector envy smell. In this section, we compare relevant attributes and the results obtained from analyzing both architectures and searching for smells.

5.5.1 Features

The Notepad SPL project contains 24 features in total (18 if the abstract ones are removed). 13 of them are mandatory, and 11 are optional. There are not alternative features. Since Notepad is in a domain that is not complex, when developing in the feature-oriented way, the features tend to be very fine-grained, i.e., individual features implement very little functionality because the variations are also fine-grained.

In the previous chapter, we raised the question of the possible relationship between having a fine-grained feature specification and the occurrence of a number of smells. In the case of feature concentration, for example, such smell tends to occur in projects with fine-grained features because it is probably the simplest way to implement variability in those conditions. Also, by having fine-grained features, the chances of choosing an ambiguous interface solution for the design are increased due to the feature repository

that is created. As far as the architecture level, the most intuitive solution would be to simplify the interface through which these features are accessed.

The design issues of the Notepad PLA are related to the implementation of the *Actions* component. It contains all features - mandatory and optional - thus we see no distinction in the occurrence of smells with the types of features. The ambiguous interface is also given by how the *GUI* component is implemented. However, the non-functional requirement of having a user interface is not included in the feature model.

Identifying bad smells basically means that the quality attributes related to the maintainability of the project may be compromised. In the case of Notepad, the chosen design represent a negative impact on reusability, understandability, changeability and testability.

On the other hand, the RescueMe SPL project contains 22 features in total (16 when removing the abstract ones). There are 10 mandatory features and 12 optional. Alternative features are also not present in this project.

The features in RescueMe are also fine-grained, but not as much as the features in Notepad. The functionalities implemented throughout RescueMe components are more complex in extension and responsibility. In this project, the main mandatory features, are orchestrated by the *Destination* component. By analyzing this component, we identified the occurrence of the connector envy smell, indicating too much responsibility with respect to connections and actual functionality. In this sense, we identified a relationship between the occurrence of the connector envy smell with the implementation of mandatory features. The *SMS Destination* feature is mandatory to all products, and is realized by the *Destination* component. This functionality requires several communication activities with other components in order to have its service performed.

The overall project's reusability, understandability and testability are negatively impacted by the occurrence of the connector envy smell. One interesting finding is that although Notepad presented more smells, only one additional quality attribute was affected (changeability).

5.5.2 Domain

The first study on architectural bad smells reports on the Notepad, which is an SPL in the text editor / desktop domain. From the analysis of features that were specified in this project, we conclude that, in this domain, features tend to be fine-grained. In other words, functionalities that are specific to text editing are likely to be of small complexity.

When the project contains very fine-grained features specified, the feature concentration smell tends to appear. Such phenomenon occurs because it is more convenient to

group small features into one single architectural entity, thus simplifying the implementation.

On the other hand, the RescueMe SPL is in the emergency / mobile domain. The implementation of emergency applications fundamentally requires several connections to be made across different components (or subsystems), in order to provide their functionality. For example, it is expected that a communication service is considered when an emergency application is designed, thus forcing the application to be developed in either a strongly dependent or a distributed manner. Such obligation increases the possibility of occurring the connector envy smell, since a lot of connection capabilities need to be implemented in addition to the application functionalities.

Further, emergency applications usually deal with asynchronous events. Having implemented a design that allows events (e.g., rescue messages) and accidentally making a procedure call between the same pair of components characterizes the extraneous adjacent connector smell. We were not able to identify such smell in the RescueMe SPL project.

5.5.3 Lines of code and complexity

One way to measure a system's complexity is through the number of lines of code and connections realized between components. Both SPLs studied in this work are fairly small and represent relatively low complexity in their functionalities. We argue that when a given system is complex, its design tends to allow the scattered parasitic functionality to appear. This occurs because several concerns are implemented throughout the components, and in such condition it is not difficult to end up developing one concern across multiple components. When this occurs, architecture entities are responsible for implementing multiple concerns, which decreases the overall component's reusability and understandability.

In summary, considering Notepad and RescueMe SPLs, we were not able to find this smell perhaps due to the low complexity of these projects.

5.5.4 Variability technique

Both SPLs use the conditional compilation technique to realize variability at the code level. While Notepad uses a GUI for manually selecting features to be included in the product derivation (at running time), RescueMe requires developers to annotate the desired features in the code. The binding time of RescueMe is development time perhaps

due to the pre-defined products that were meant to be developed - obeying the product requirements initially specified.

We argue that the conditional compilation technique may induce programmers to concentrate concerns into a single architecture entity. The parts of the code referring to specific features are annotated and easily recognized, thus making it easy to use the architectural entity as a feature repository.

In conclusion, having conditional compilation as variability technique at the code level decreases the possibility of having the scattered parasitic functionality.

5.5.5 Patterns

One attribute of RescueMe that differs from Notepad is the pattern in which the developers were forced to develop the application. Developing for iOS assumes that every application is written in the MVC pattern. As previously mentioned, such scenario aids in the separation of concerns, thus decreasing the possibility of occurring the scattered parasitic functionality.

Further, such obligation to follow strict rules may induce the occurrence of ambiguous interfaces, as they are not directly managed by the developers. Despite the need to include the frameworks that provide essential services to the developing application. the implementation mechanism is not always transparent to developers when developing iOS applications. Systems containing clear contents within components and how they communicate through interfaces are easier to maintain.

In summary, we conclude that the architectural pattern might influence in the occurrence of smells. In special, developing for the iOS platform may even prevent such design decisions to be made.

5.6 Main Findings

This study aimed to help in the understanding of architectural bad smells in software product lines. The objective was to acquire evidence on whether such phenomenon occurred in SPLs, by considering two PLAs in different domains. Thus, we performed architecture recovery, identified bad smells, and compared results of both studies. The main findings obtained from these activities can be summarized as follows:

- The granularity of the features plays an important role in the occurrence of smells. Having a small SPL (or having a SPL in a domain such as the Text Editor) may

influence in the definition of fine grained features, thus increasing the chances of the architect to decide upon a solution that concentrates features into a single architecture entity. By making such decision, the feature concentration smell occurs and the PLA is negatively impacted on understandability, changeability;

- Having too many mandatory features may influence in the occurrence of the connector envy smell. Since these features are present in every product to be derived, developers tend to inflate the referred architectural entities with both functionality and connection capabilities. In these cases, reusability, understandability and testability are negatively affected;

- Developing a complex system is many times necessary, but such scenario may induce scattered parasitic functionality to occur. Although none of the sample SPLs were very complex, we argue that distributing concerns/functionalities in complex systems is easier to occur;

- iOS applications are required to be developed under specific rules, and some aspects of implementation are hidden from the developer. This scenario alone can make possible the occurrence of ambiguous interfaces, since not all internals of the components are clear. Fortunately, we were not able to find ambiguous interfaces in RescueMe;

- Deciding upon the use of a pattern that clearly separates responsibilities may decrease the chances of occurring the scattered parasitic functionality smell. When the concerns are transparently separated, the architecture achieves higher cohesion levels, thus avoiding the negative impacts related to the scattered parasitic functionality smell;

- Using conditional compilation may induce the occurrence of the feature concentration smell. Such technique most of the times allows code annotation, thus making it easier to separate concerns visually, but not necessarily architecturally.

5.7 Threats to Validity

There are a number of threats to the validity of this work, as described next.

The present study subjected to the same threats as described in the last chapter. The architecture recovery process may be guided by subjective interpretations. The specifications obtained when analyzing the same code and available artifacts can vary when

conducted by different individuals. In order to minimize this threat, the recovery process was carefully conducted by two researchers. We discussed the different interpretations and agreed on the specification reported in this work. Further, we carefully validated the obtained model and specification with the developers.

In addition, we were only able to use fairly small SPLs and the search for smells is more valuably conducted in more complex systems. In order to minimize this threat, we selected two SPLs in different domains. By having two different objects of study and different results, we increase the generalizability of our conclusions.

On the other hand, having SPLs in different domains may be a threat on its own. The SPLs considered in this work are in different domains and were implemented using different programming languages, technologies and platforms. Such scenario may have influenced in the search for smells and comparison of results. However, observing how the proposed smells are identified and behave in different contexts was part of the goals in the comparative analysis.

5.8 Chapter Summary

In this chapter, we reported on a replication study that aimed at obtaining evidence from an analysis of an SPL in a different domain than the one used in the previous chapter. This time, we selected a sample SPL in the emergency / mobile domain. In this study, we were also required to recover the PLA, because the available architecture artifacts were not sufficient for the intended search for smells. Thus, we followed the same recovery process explained in the last chapter, and obtained means for properly searching for smells.

Then, we performed the same procedures for identifying smells. With the intention of drawing further conclusions from the conduction of both studies, we performed a comparative analysis, discussing differences and similarities in both studies. The Connector Envy smell was identified in both studies, and we related the occurrence of smells with a number of aspects of the compared SPLs: (i) types of features specified, (ii) domain of the SPLs, (iii) number of source lines of code and complexity, (iv) chosen variability management technique, and (v) architectural patterns used.

Next chapter presents the conclusions of this dissertation. We briefly describe the work conducted, the published paper, and ideas on future directions of research in this field.

6 Conclusions

This dissertation reports on the conduction of a systematic mapping study, from which we were able to determine what issues have been addressed in the field of Software Product Line Architectures. It also provides maps and discussions to help researchers in their planning for future research.

The amount of methods that handle different and specific aspects in PLAs make the studies comparison a difficult task, since they do not deal with the same goals or contexts. Nevertheless, in this work we identified and discussed properties of published work in order to draw conclusions on how the researchers were conducting research work in PLA in the lastest years. Relevant research aspects were raised, and these can be considered an important input to further research.

The results of the review suggest that the current studies are concerned with resolving smaller parts of the problem, instead of wrapping a more complete solution for the PLA domain. We have also identified very few papers concerning the validation of existing approaches.

Further, we presented a case study that aimed at understanding the phenomenon of architectural bad smells in the context of SPLs. The study consisted in (i) recovering the architecture of a sample PLA, (ii) analyzing both the code and the recovered design decisions, and (iii) discussing the occurrence of smells against issues related to variability and SPLs.

We have discovered that the same smells initially identified in single systems can also be found in PLAs. As far as the evaluation process, we were able to use the same method, originally proposed to single systems, for analysing architectures and identifying smells in the SPL context. On the other hand, we argue that the choice of using Java libraries may have influenced the process of identifying smells due to its intrinsic architectural implications. Moreover, we proposed a SPL-specific smell that indicates the concentration

of features in a single design entity.

After conducting the exploratory study, we considered an SPL in a different domain, in order to replicate the initial study and find out whether the same smells occur in SPLs in different domains. From the replicated study, we were only able to identify one smell - connector envy - which is characterized by the double responsibility of an architecture component (application functionality and internal connections). From the analysis, we compared the attributes and the results obtained from both studies, concerning the two sample SPLs. We identified a number of aspects that might have influenced in the occurence of smells in them.

6.1 Published Work

The work reported in this dissertation has resulted in one publication (de Andrade *et al.*, 2014) at the Third International Workshop on Variability in Software Architecture, which is an internal event held within WICSA - Working IEEE/IFIP Conference on Software Architecture, in the year of 2014.

Additional papers concerning the systematic mapping study and the empirical studies have been submitted and are currently under evaluation.

6.2 Future Work

From the results obtained in this dissertation, we have identified a number of aspects that can be investigated in the future, as decribed next.

- **Patterns in PLAs**. We identified a number of aspects that can be further investigated in the area of PLAs, such as the use of patterns. Additional experiences in using different types of patterns in SPLs can be valuable to aid architects in designing better PLAs, according to their needs;

- **Validating existing approaches**. From the literature review, we found several approaches aiming to resolve different issues in PLAs. However, it would be valuable to conduct additional research aiming at the evaluation of the existing ones. In this sense, the approaches would be empirically validated, thus achieving higher levels of consistency and trustability over time;

- **Industrial SPL projects**. It would be interesting to combine the evidence obtained in this work with evidence from empirical studies in industrial SPL projects. For

example, a survey can be conducted with SPL experts, regarding their experiences in maintainability issues of PLAs;

- **Types of features**. One can also further investigate the relation between the occurrence of smells the types of features that were specified;
- **Identifying smells**. Another valuable work can be conducted on the feasibility of using a tool for identifying smells;
- **Avoiding smells**. We discussed the benefits of identifying architectural smells in SPL projects. Thus, one interesting work to be conducted is on the need/effectiveness of adopting an inspection process during the PLA design stage;
- **Resolving smells**. Another interesting investigation would be on how to effectively remedy the occurrence of smells. It would be valuable to evaluate whether it is appropriate to resolve smells through specific manual operations, using tools to support such corrections, or other manners.

6.3 Concluding Remarks

The goal of this work was to investigate the phenomenon of architectural bad smells in software product lines. The search for architectural smells is motivated by the fact that extensive knowledge with respect to the business' processes is not required for identifying points of improvement in PLAs. The concept is similar to *code smells*, but held in a different level of abstraction.

From this work, we conclude that additional investigations regarding architectural bad smells in the context of SPLs would be valuable for both practitioners and researchers. Despite the evidence we obtained from the studies, it would be interesting to further investigate the phenomenon in industry contexts.

Our findings helped us in the understanding of architectural smells in SPLs, and our evidences show that such design decisions have a direct impact on the lifecycle properties of PLAs.

References

Abele, A., Lönn, H., Reiser, M.-O., Weber, M., and Glathe, H. (2012). EPM: a prototype tool for variability management in component hierarchies. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 246–249, New York, NY, USA. ACM.

Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L., and Zaremski, A. (1997). Recommended Best Industrial Practice for Software Architecture Evaluation. Technical report, Software Engineering Institute (SEI) Carnegie Mellon University.

Abu-Matar, M. and Gomaa, H. (2011). Feature Based Variability for Service Oriented Architectures. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 302–309.

Adachi, E., Batista, T., Kulesza, U., Medeiros, A., Chavez, C., and Garcia, A. (2009). Variability Management in Aspect-Oriented Architecture Description Languages: An Integrated Approach. In *Software Engineering, 2009. SBES '09. XXIII Brazilian Symposium on*, pages 1–11.

Ahn, H. and Kang, S. (2011). Analysis of Software Product Line Architecture Representation Mechanisms. In *9th International Conference on Software Engineering Research, Management and Applications, SERA 2011, Baltimore, MD, USA, August 10-12, 2011*, pages 219–226. IEEE Computer Society.

Alves, V., Niu, N., Alves, C., and Valença, G. (2010). Requirements engineering for software product lines: A systematic literature review. *Inf. Softw. Technol.*, **52**(8), 806–820.

America, P., Obbink, H., Muller, J., and van Ommering, R. (2000). COPA: A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products.

America, P., Hammer, D. K., Ionita, M. T., Obbink, J. H., and Rommes, E. (2005). Scenario-based decision making for architectural variability in product families. *Software Process: Improvement and Practice*, **10**, 171–187.

Angelov, S., Grefen, P. W. P. J., and Greefhorst, D. (2009). A classification of software reference architectures: Analyzing their success and effectiveness. In *Joint Working IEEE/IFIP Conference on Software Architecture 2009 and European Conference on*

Software Architecture 2009, WICSA/ECSA 2009, Cambridge, UK, 14-17 September 2009, pages 141–150. IEEE.

Angelov, S., Grefen, P., and Greefhorst, D. (2012). A framework for analysis and design of software reference architectures. *Inf. Softw. Technol.*, **54**(4), 417–431.

Apel, S. and Beyer, D. (2011). Feature cohesion in software product lines: an exploratory study. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 421–430, New York, NY, USA. ACM.

Arcoverde, R., Macia, I., Garcia, A., and Von Staa, A. (2012). Automatically detecting architecturally-relevant code anomalies. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 90–91.

Asadi, M., Mohabbati, B., Kaviani, N., Gasevic, D., Boskovic, M., and Hatala, M. (2009). Model-driven development of families of service-oriented architectures. In S. Apel, W. R. Cook, K. Czarnecki, C. Kästner, N. Loughran, and O. Nierstrasz, editors, *Proceedings of the First International Workshop on Feature-Oriented Software Development, FOSD 2009, Denver, Colorado, USA, October 6, 2009*, ACM International Conference Proceeding Series, pages 95–102. ACM.

Atkinson, C., Bayer, J., and Muthig, D. (2000). Component-based product line development: the KobrA Approach. In P. Donohoe, editor, *Software Product Lines; Experiences and Research Directions, Proceedings of the First International Conference, SPLC 1, Denver, Colorado, USA, August 28-31, 2000, Proceedings*, pages 289–310. Kluwer.

Babar, M. A. (2004). Scenarios, quality attributes, and patterns: Capturing and using their synergistic relationships for product line architectures. In *11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 574–578. IEEE Computer Society.

Babar, M. A., Ihme, T., and Pikkarainen, M. (2009). An industrial case of exploiting product line architectures in agile software development. In D. Muthig and J. D. McGregor, editors, *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, volume 446 of *ACM International Conference Proceeding Series*, pages 171–179. ACM.

Bachmann, F. and Bass, L. J. (2001). Managing variability in software architectures. In *SSR*, pages 126–132.

Barbosa, E. A., Batista, T. V., Garcia, A. F., and Silva, E. (2011). PL-AspectualACME: An Aspect-Oriented Architectural Description Language for Software Product Lines. In I. Crnkovic, V. Gruhn, and M. Book, editors, *Software Architecture - 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings*, volume 6903 of *Lecture Notes in Computer Science*, pages 139–146. Springer.

Bashroush, R., Brown, T. J., Spence, I. T. A., and Kilpatrick, P. (2006). ADLARS: An Architecture Description Language for Software Product Lines. In *29th Annual IEEE / NASA Software Engineering Workshop (SEW-29 2005), 6-7 April 2005, Greenbelt, Maryland, USA*, pages 163–173. IEEE Computer Society.

Bass, L., Clements, P., and Kazman, R. (2003a). *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc.

Bass, L. J., Bachmann, F., and Klein, M. (2003b). Making Variability Decisions during Architecture Design. In F. van der Linden, editor, *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, volume 3014 of *Lecture Notes in Computer Science*, pages 454–465. Springer.

Bernardo, M., Ciancarini, P., and Donatiello, L. (2002). Architecting families of software systems with process algebras. *ACM Trans. Softw. Eng. Methodol.*, **11**(4), 386–426.

Bertoncello, I. A., Dias, M. O., Brito, P. H. S., and Rubira, C. M. F. (2008). Explicit exception handling variability in component-based product line architectures. In *Proceedings of the 4th International Workshop on Exception Handling, WEH 2008, Atlanta, Georgia, USA, November 14, 2008*, pages 47–54. ACM.

Birman, K. and Joseph, T. (1987). Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, **21**(5), 123–138.

Bosch, J. (2000). *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Botterweck, G., O'Brien, L., and Thiel, S. (2007). Model-driven derivation of product architectures. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 469–472. ACM.

Breivold, H. P., Crnkovic, I., and Larsson, M. (2012). A systematic review of software architecture evolution research. *Inf. Softw. Technol.*, **54**(1), 16–40.

Brown, W. J., Malveau, R. C., McCormick, III, H. W., and Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA.

Budgen, D., Turner, M., Brereton, P., and Kitchenham, B. (2008). Using Mapping Studies in Software Engineering. In *Proceedings of PPIG 2008*, pages 195–204. Lancaster University.

Capilla, R. and Babar, M. A. (2008). On the Role of Architectural Design Decisions in Software Product Line Engineering. In R. Morrison, D. Balasubramaniam, and K. E. Falkner, editors, *Software Architecture, Second European Conference, ECSA 2008, Paphos, Cyprus, September 29 - October 1, 2008, Proceedings*, volume 5292 of *Lecture Notes in Computer Science*, pages 241–255. Springer.

Carvalho, S., Murta, L., and Loques, O. (2012). Variabilities as first-class elements in product line architectures of homecare systems. In *Software Engineering in Health Care (SEHC), 2012 4th International Workshop on*, pages 33–39.

Carver, J. C. (2010). Towards reporting guidelines for experimental replications: A proposal. In *1st international workshop on replication in empirical software engineering research (RESER 2010)*, RESER '10, New York, NY, USA. ACM SIGSOFT Software Engineering Notes.

Cavalcanti, R. O., de Almeida, E. S., and Meira, S. R. L. (2011). Extending the RiPLE-DE process with quality attribute variability realization. In I. Crnkovic, J. A. Stafford, D. C. Petriu, J. Happe, and P. Inverardi, editors, *7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS 2011. Boulder, CO, USA, June 20-24, 2011, Proceedings*, pages 159–164. ACM.

Cho, H. and Yang, J.-S. (2008). Architecture Patterns for Mobile Games Product Lines. In *Advanced Communication Technology, 2008. ICACT 2008. 10th International Conference on*, volume 1, pages 118–122.

Choi, Y., Shin, G., Yang, Y., and Park, C. (2005). An approach to extension of uml 2.0 for representing variabilities. In *Proceedings of the Fourth Annual ACIS Interna-*

tional Conference on Computer and Information Science, ICIS '05, pages 258–261, Washington, DC, USA. IEEE Computer Society.

Choppy, C. and Reggio, G. (2005). A uml-based approach for problem frame oriented software development. *Inf. Softw. Technol.*, **47**(14), 929–954.

Clements, P. and McGregor, J. (2012). Better, faster, cheaper: Pick any three. *Business Horizons*, **55**(2), 201 – 208.

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond (2nd Edition)*. Addison-Wesley Professional, 2 edition.

da Mota Silveira Neto, P. A., Carmo Machado, I. d., McGregor, J. D., de Almeida, E. S., and de Lemos Meira, S. R. (2011). A systematic mapping study of software product lines testing. *Inf. Softw. Technol.*, **53**(5), 407–423.

Dao, T. M. and Kang, K. C. (2010). Mapping features to reusable components: A problem frames-based approach. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, volume 6287 of *Lecture Notes in Computer Science*, pages 377–392. Springer.

Dashofy, E. M. and Hoek, A. v. d. (2002a). Representing product family architectures in an extensible architecture description language. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 330–341, London, UK, UK. Springer-Verlag.

Dashofy, E. M. and Hoek, A. v. d. (2002b). Representing product family architectures in an extensible architecture description language. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 330–341, London, UK, UK. Springer-Verlag.

Dashofy, E. M. and Hoek, A. v. d. (2002c). Representing product family architectures in an extensible architecture description language. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 330–341, London, UK, UK. Springer-Verlag.

de Andrade, H. S., Almeida, E., and Crnkovic, I. (2014). Architectural bad smells in software product lines: An exploratory study. In *Proceedings of the WICSA 2014 Companion Volume*, WICSA '14 Companion, pages 12:1–12:6, New York, NY, USA. ACM.

de Oliveira, L. B. R., Felizardo, K. R., Feitosa, D., and Nakagawa, E. Y. (2010). Reference models and reference architectures based on service-oriented architecture: A systematic review. In M. A. Babar and I. Gorton, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings*, volume 6285 of *Lecture Notes in Computer Science*, pages 360–367. Springer.

de Oliveira Junior, E. A., de Souza Gimenes, I. M., and Maldonado, J. C. (2011). A Meta-Process to Support Trade-Off Analysis in Software Product Line Architecture. In *SEKE*, pages 687–692. Knowledge Systems Institute Graduate School.

DeBaud, J.-M., Flege, O., and Knauber, P. (1998). Pulse-dssa—a method for the development of software reference architectures. In *Proceedings of the third international workshop on Software architecture*, ISAW '98, pages 25–28, New York, NY, USA. ACM.

Del Rosso, C. (2006). Continuous evolution through software architecture evaluation: a case study: Practice articles. *J. Softw. Maint. Evol.*, **18**(5), 351–383.

Dhungana, D., Rabiser, R., and Grünbacher, P. (2007). Decision-Oriented Modeling of Product Line Architectures. In *Sixth Working IEEE / IFIP Conference on Software Architecture (WICSA 2007), 6-9 January 2005, Mumbai, Maharashtra, India*, page 22. IEEE Computer Society.

Díaz, J., Pérez, J., Garbajosa, J., and Wolf, A. L. (2011). Change Impact Analysis in Product-Line Architectures. In I. Crnkovic, V. Gruhn, and M. Book, editors, *Software Architecture - 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings*, volume 6903 of *Lecture Notes in Computer Science*, pages 114–129. Springer.

Dincel, E., Medvidovic, N., and van der Hoek, A. (2001). Measuring Product Line Architectures. In F. van der Linden, editor, *Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers*, volume 2290 of *Lecture Notes in Computer Science*, pages 346–352. Springer.

Ducasse, S. and Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Softw. Eng.*, **35**(4), 573–591.

Dybå, T., Dingsoyr, T., and Hanssen, G. (2007). Applying Systematic Reviews to Diverse Study Types: An Experience Report. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 225–234.

Easterbrook, S., Singer, J., Storey, M.-A., and Damian, D. (2008). Selecting Empirical Methods for Software Engineering Research. In F. Shull and J. Singer, editors, *Guide to Advanced Empirical Software Engineering*, chapter 11, pages 285–311. Springer London, London.

Eixelsberger, W. (1998). Recovery of a reference architecture: a case study. In *Proceedings of the third international workshop on Software architecture*, ISAW '98, pages 33–36, New York, NY, USA. ACM.

Engström, E. and Runeson, P. (2011). Software product line testing - a systematic mapping study. *Inf. Softw. Technol.*, **53**(1), 2–13.

Etxeberria, L. and Mendieta, G. S. (2005). Product-Line Architecture: New Issues for Evaluation. In J. H. Obbink and K. Pohl, editors, *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*, pages 174–185. Springer.

Etxeberria, L. and Sagardui, G. (2008). Variability driven quality evaluation in software product lines. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 243–252.

Fant, J. S. (2011). Building domain specific software architectures from software architectural design patterns. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1152–1154. ACM.

Feigenspan, J., Kästner, C., Frisch, M., Dachselt, R., and Apel, S. (2010). Visual support for understanding product lines. In *The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010*, pages 34–35. IEEE Computer Society.

Filho, E. D. S., Cavalcanti, R. O., Neiva, D. F. S., Oliveira, T. H. B., Lisboa, L. B., Almeida, E. S., and Meira, S. R. L. (2008). Evaluating domain design approaches

using systematic review. In R. Morrison, D. Balasubramaniam, and K. E. Falkner, editors, *2nd European Conference on Software Architecture (ECSA'08)*, volume 5292 of *Lecture Notes in Computer Science*, pages 50–65. Springer.

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Galster, M. (2010). Describing variability in service-oriented software product lines. In I. Gorton, C. E. Cuesta, and M. A. Babar, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Companion Volume*, ACM International Conference Proceeding Series, pages 344–350. ACM.

Galster, M., Weyns, D., Avgeriou, P., and Becker, M. (2013). Variability in software architecture: views and beyond. *SIGSOFT Softw. Eng. Notes*, **37**(6), 1–9.

Galvão, I., van den Broek, P., and Aksit, M. (2010). A model for variability design rationale in SPL. In I. Gorton, C. E. Cuesta, and M. A. Babar, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Companion Volume*, ACM International Conference Proceeding Series, pages 332–335. ACM.

Gannod, G. C. and Lutz, R. R. (2000). An approach to architectural analysis of product lines. In *ICSE*, pages 548–557.

Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. (2009). Toward a catalogue of architectural bad smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems*, QoSA '09, pages 146–162, Berlin, Heidelberg. Springer-Verlag.

Ghezzi, C. and Molzam Sharifloo, A. (2013). Model-based verification of quantitative non-functional properties for software product lines. *Inf. Softw. Technol.*, **55**(3), 508–524.

Goedicke, M., Köllmann, C., and Zdun, U. (2004). Designing runtime variation points in product line architectures: three cases. *Sci. Comput. Program.*, **53**(3), 353–380.

Gomaa, H. (1995). Reusable software requirements and architectures for families of systems. *Journal of Systems and Software*, **28**(3), 189 – 202.

Gomaa, H. (2000). Object Oriented Analysis and Modeling for Families of Systems with UML. In W. B. Frakes, editor, *Software Reuse: Advances in Software Reusability, 6th International Conerence, ICSR-6, Vienna, Austria, June 27-29, 2000, Proceedings*, volume 1844 of *Lecture Notes in Computer Science*, pages 89–99. Springer.

Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Groher, I. and Weinreich, R. (2012). Integrating Variability Management and Software Architecture. In *Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture and 6th European Conference on Software Architecture (accepted for publication)*, WICSA/ECSA'12.

GröNer, G., BošKović, M., Silva Parreiras, F., and GašEvić, D. (2013). Modeling and validation of business process families. *Inf. Syst.*, **38**(5), 709–726.

Guana, V. and Correal, D. (2013). Improving software product line configuration: A quality attribute-driven approach. *Information and Software Technology*. <ce:title>Special Issue on Software Reuse and Product Lines</ce:title> <ce:subtitle>Special Issue on Software Reuse and Product Lines</ce:subtitle>.

Haber, A., Kutz, T., Rendel, H., Rumpe, B., and Schaefer, I. (2011a). Delta-oriented architectural variability using MontiCore. In W. Hasselbring and V. Gruhn, editors, *Software Architecture, 5th European Conference, ECSA 2011, Essen, Germany, September 13 - 16, 2011. Companion Volume*, ACM International Conference Proceeding Series, page 6. ACM.

Haber, A., Rendel, H., Rumpe, B., Schaefer, I., and van der Linden, F. (2011b). Hierarchical variability modeling for software architectures. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 150–159, Washington, DC, USA. IEEE Computer Society.

Hallsteinsen, S. O., Fægri, T. E., and Syrstad, M. (2003). Patterns in Product Family Architecture Design. In F. van der Linden, editor, *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, volume 3014 of *Lecture Notes in Computer Science*, pages 261–268. Springer.

Hendrickson, S. A. and van der Hoek, A. (2007). Modeling Product Line Architectures through Change Sets and Relationships. In *29th International Conference on Software*

Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, pages 189–198. IEEE Computer Society.

Hilliard, R. (2010). On representing variation. In I. Gorton, C. E. Cuesta, and M. A. Babar, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Companion Volume*, ACM International Conference Proceeding Series, pages 312–315. ACM.

Johansson, E. and Höst, M. (2002). Tracking degradation in software product lines through measurement of design rule violations. In *SEKE*, pages 249–254.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, **5**, 143–168.

Kavimandan, A., Gokhale, A. S., Karsai, G., and Gray, J. (2011). Managing the quality of software product line architectures through reusable model transformations. In I. Crnkovic, J. A. Stafford, D. C. Petriu, J. Happe, and P. Inverardi, editors, *7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS 2011. Boulder, CO, USA, June 20-24, 2011, Proceedings*, pages 13–22. ACM.

Kazman, R., Bass, L., Webb, M., and Abowd, G. (1994). Saam: a method for analyzing the properties of software architectures. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 81–90, Los Alamitos, CA, USA. IEEE Computer Society Press.

Kazman, R., Klein, M., and Clements, P. (2000). ATAM: Method for Architecture Evaluation. Technical report, CMU/SEI.

Keith, M. and Wen, G. (2010). Advances in Social Science Research Using R, Book Chapter Bubble Plots as a Model-Free Graphical Tool for Continuous Variables. *Journal of Statistical Software, Book Reviews*, **34**(2).

Kekre, S. and Srinivasan, K. (1990). Broader product line: A necessity to achieve success? *Manage. Sci.*, **36**(10), 1216–1231.

Kim, J., Park, S., and Sugumaran, V. (2008a). DRAMA: A framework for domain requirements analysis and modeling architectures in software product lines. *Journal of Systems and Software*, **81**(1), 37–55.

Kim, K., Kim, H., Kim, S., and Chang, G. (2008b). A Case Study on SW Product Line Architecture Evaluation: Experience in the Consumer Electronics Domain. In *Software Engineering Advances, 2008. ICSEA '08. The Third International Conference on*, pages 192–197.

Kim, T., Ko, I. Y., Kang, S. W., and Lee, D. H. (2008c). Extending ATAM to assess product line architecture. In *Computer and Information Technology, 2008. CIT 2008. 8th IEEE International Conference on*, pages 790–797.

Kim, Y.-G., Lee, S. K., and Jang, S.-B. (2011). Variability Management for Software Product-Line Architecture Development. *International Journal of Software Engineering and Knowledge Engineering*, **21**(07), 931–956.

Kishi, T., Noda, N., and Katayama, T. (2005). Design Verification for Product Line Development. In J. H. Obbink and K. Pohl, editors, *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*, pages 150–161. Springer.

Kitchenham, B. and Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering. *Software Engineering Group School of*, **2**, 1051.

Kolb, R. and Muthig, D. (2006). Making testing product lines more efficient by improving the testability of product line architectures. In R. M. Hierons and H. Muccini, editors, *Proceedings of the 2006 Workshop on Role of Software Architecture for Testing and Analysis, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), ROSATEA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 22–27. ACM.

Kruchten, P., Obbink, H., and Stafford, J. (2006). The past, present, and future for software architecture. *IEEE Softw.*, **23**(2), 22–30.

Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, **24**(2), 131–183.

Lee, H., Yang, J.-s., and Kang, K. C. (2012). Vulcan: Architecture-model-based workbench for product line engineering. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 260–264, New York, NY, USA. ACM.

Lee, J., Muthig, D., and Naab, M. (2010). A feature-oriented approach for developing reusable product line assets of service-based systems. *Journal of Systems and Software*, **83**(7), 1123–1136.

Lin, Y., Ye, H., and Li, G. (2010). An Approach for Modelling Software Product Line Architecture. In *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, pages 1 –4.

Lippert, M. and Roock, S. (2006). *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 1 edition.

Lutz, R. R. and Gannod, G. C. (2003). Analysis of a software product line architecture: an experience report. *Journal of Systems and Software*, **66**(3), 253–267.

Maccari, A. (2002). Experiences in assessing product family software architecture for evolution. In *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 585–592. ACM.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., and von Staa, A. (2012a). Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 167–178, New York, NY, USA. ACM.

Macia, I., Arcoverde, R., Garcia, A., Chavez, C., and von Staa, A. (2012b). On the relevance of code anomalies for identifying architecture degradation symptoms. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 277–286, Washington, DC, USA. IEEE Computer Society.

Mann, S. and Rock, G. (2009). Dealing with Variability in Architecture Descriptions to Support Automotive Product Lines. In D. Benavides, A. Metzger, and U. W. Eisenecker, editors, *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings*, volume 29 of *ICB Research Report*, pages 111–120. Universität Duisburg-Essen.

Matinlassi, M. (2004a). Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In *26th International Conference*

on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom, pages 127–136. IEEE Computer Society.

Matinlassi, M. (2004b). Evaluating the portability and maintainability of software product family architecture: Terminal software case study. In *4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12-15 June 2004, Oslo, Norway*, pages 295–300. IEEE Computer Society.

Matinlassi, M., Niemelä, E., and Dobrica, L. (2002). *Quality-driven Architecture Design and Quality Analysis Method: A Revolutionary Initiation Approach to a Product Line Architecture*. VTT publications: Valtion Teknillinen Tutkimuskeskus. Technical Research Centre of Finland.

Meister, J., Reussner, R., and Rohde, M. (2004). Applying Patterns to Develop a Product Line Architecture for Statistical Analysis Software. In *4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004), 12-15 June 2004, Oslo, Norway*, pages 291–294. IEEE Computer Society.

Meyer, M. and Lehnerd, A. (1997). *The Power of Product Platforms*. Free Press.

Montagud, S. and Abrahão, S. (2009). Gathering current knowledge about quality evaluation in software product lines. In D. Muthig and J. D. McGregor, editors, *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, volume 446 of *ACM International Conference Proceeding Series*, pages 91–100. ACM.

Montagud, S., Abrahão, S., and Insfran, E. (2012). A systematic review of quality attributes and measures for software product lines. *Software Quality Journal*, **20**, 425–486. 10.1007/s11219-011-9146-7.

Moon, M. and Yeom, K. (2006). An Approach to Developing Domain Architectures Based on Variability Analysis. In M. L. Gavrilova, O. Gervasi, V. Kumar, C. J. K. Tan, D. Taniar, A. Laganà, Y. Mun, and H. Choo, editors, *Computational Science and Its Applications - ICCSA 2006, International Conference, Glasgow, UK, May 8-11, 2006, Proceedings, Part II*, volume 3981 of *Lecture Notes in Computer Science*, pages 441–450. Springer.

Moon, M., Chae, H. S., and Yeom, K. (2006). A Metamodel Approach to Architecture Variability in a Product Line. In M. Morisio, editor, *Reuse of Off-the-Shelf Components,*

9th International Conference on Software Reuse, ICSR 2006, Turin, Italy, June 12-15, 2006, Proceedings, volume 4039 of *Lecture Notes in Computer Science*, pages 115–126. Springer.

Moon, M., Chae, H. S., Nam, T., and Yeom, K. (2007). A Metamodeling Approach to Tracing Variability between Requirements and Architecture in Software Product Lines. In *Seventh International Conference on Computer and Information Technology (CIT 2007), October 16-19, 2007, University of Aizu, Fukushima, Japan*, pages 927–933. IEEE Computer Society.

Moraes, M. B. S., Almeida, E. S., and Meira, S. R. L. (2009). A Systematic Review on Software Product Lines Scoping. In *VI Experimental Software Engineering Latin American Workshop (ESELAW)*, pages 63–72.

Morisawa, Y. and Torii, K. (2001). An architectural style of product lines for distributed processing systems, and practical selection method. In *ESEC / SIGSOFT FSE*, pages 11–20.

Murugesupillai, E., Mohabbati, B., and Gasevic, D. (2011). A preliminary mapping study of approaches bridging software product lines and service-oriented architectures. In I. Schaefer, I. John, and K. Schmid, editors, *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011. Workshop Proceedings (Volume 2)*, page 11. ACM.

Murwantara, I. (2011). Initiating layers architecture design for Software Product Line. In *Uncertainty Reasoning and Knowledge Engineering (URKE), 2011 International Conference on*, volume 1, pages 48–51.

Muthig, D. and Atkinson, C. (2002). Model-Driven Product Line Architectures. In G. J. Chastek, editor, *Software Product Lines, Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22, 2002, Proceedings*, volume 2379 of *Lecture Notes in Computer Science*, pages 110–129. Springer.

Nakagawa, E. Y. (2012). Reference architectures and variability: current status and future perspectives. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, pages 159–162, New York, NY, USA. ACM.

Nakagawa, E. Y., Antonino, P. O., and Becker, M. (2011). Reference architecture and product line architecture: a subtle but critical difference. In *Proceedings of the 5th*

European conference on Software architecture, ECSA'11, pages 207–211, Berlin, Heidelberg. Springer-Verlag.

Neto, P. A. d. M. S., Machado, I. d. C., Cavalcanti, Y. C., Almeida, E. S. d., Garcia, V. C., and Meira, S. R. d. L. (2010). A regression testing approach for software product lines architectures. In *Proceedings of the 2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse*, SBCARS '10, pages 41–50, Washington, DC, USA. IEEE Computer Society.

Niemelä, E., Matinlassi, M., and Taulavuori, A. (2004). Practical Evaluation of Software Product Family Architectures. In R. L. Nord, editor, *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004, Proceedings*, volume 3154 of *Lecture Notes in Computer Science*, pages 130–145. Springer.

Oliveira, D. and Rosa, N. (2009). Ubá: A Software Product Line Architecture for Grid-Oriented Middleware. In S. I. Ahamed, E. Bertino, C. K. Chang, V. Getov, L. Liu, H. Ming, and R. Subramanyan, editors, *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2009, Seattle, Washington, USA, 20-24 July 2009*, pages 160–165. IEEE Computer Society.

Oliveira Junior, E., Maldonado, J., and Gimenes, I. (2010). Empirical Validation of Complexity and Extensibility Metrics for Software Product Line Architectures. In *Software Components, Architectures and Reuse (SBCARS), 2010 Fourth Brazilian Symposium on*, pages 31–40.

Oliveira Junior, E. A., Gimenes, I. M. S., Maldonado, J. C., Masiero, P. C., and Barroca, L. (2013). Systematic evaluation of software product line architectures. *Journal of Universal Computer Science*, **19**(1), 25–52.

Olumofin, F. G. and Misic, V. B. (2005). Extending the ATAM Architecture Evaluation to Product Line Architectures. In *Fifth Working IEEE / IFIP Conference on Software Architecture (WICSA 2005), 6-10 November 2005, Pittsburgh, Pennsylvania, USA*, pages 45–56. IEEE Computer Society.

Olumofin, F. G. and Misic, V. B. (2007). A holistic architecture assessment method for software product lines. *Information & Software Technology*, **49**(4), 309–323.

Pérez, J., Díaz, J., Soria, C. C., and Garbajosa, J. (2009). Plastic Partial Components: A solution to support variability in architectural components. In *Joint Working IEEE/IFIP*

Conference on Software Architecture 2009 and European Conference on Software Architecture 2009, WICSA/ECSA 2009, Cambridge, UK, 14-17 September 2009, pages 221–230. IEEE.

Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). Systematic Mapping Studies in Software Engineering. *12th International Conference on Evaluation and Assessment in Software Engineering*.

Pinzger, M., Gall, H., Girard, J.-F., Knodel, J., Riva, C., Pasman, W., Broerse, C., and Wijnstra, J. G. (2003). Architecture recovery for product families. In F. van der Linden, editor, *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, volume 3014 of *Lecture Notes in Computer Science*, pages 332–351. Springer.

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Razavian, M. and Khosravi, R. (2008). Modeling variability in the component and connector view of architecture using uml. In *Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications*, AICCSA '08, pages 801–809, Washington, DC, USA. IEEE Computer Society.

Reinhartz-Berger, I. and Sturm, A. (2009). Utilizing domain models for application design and validation. *Information & Software Technology*, **51**(8), 1275–1289.

Rosik, J., Le Gear, A., Buckley, J., Babar, M. A., and Connolly, D. (2011). Assessing architectural drift in commercial software development: a case study. *Softw. Pract. Exper.*, **41**(1), 63–86.

Rossel, P. O., Perovich, D., and Bastarrica, M. C. (2009). Reuse of Architectural Knowledge in SPL Development. In S. H. Edwards and G. Kulczycki, editors, *Formal Foundations of Reuse and Domain Engineering, 11th International Conference on Software Reuse, ICSR 2009, Falls Church, VA, USA, September 27-30, 2009. Proceedings*, volume 5791 of *Lecture Notes in Computer Science*, pages 191–200. Springer.

Rosso, C. D. (2006). Continuous evolution through software architecture evaluation: a case study. *Journal of Software Maintenance*, **18**(5), 351–383.

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, **14**(2), 131–164.

Satyananda, T., Lee, D., Kang, S., and Hashmi, S. (2007a). Identifying Traceability between Feature Model and Software Architecture in Software Product Line using Formal Concept Analysis. In *Computational Science and its Applications, 2007. ICCSA 2007. International Conference on*, pages 380–388.

Satyananda, T. K., Lee, D., and Kang, S. (2007b). Formal Verification of Consistency between Feature Model and Software Architecture in Software Product Line. In *Proceedings of the Second International Conference on Software Engineering Advances (ICSEA 2007)*. IEEE Computer Society.

SEI, F. (2001). *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Silva, E., Medeiros, A. L., Cavalcante, E., and Batista, T. (2013). A lightweight language for software product lines architecture description. In *Proceedings of the 7th European Conference on Software Architecture*, ECSA'13, pages 114–121, Berlin, Heidelberg. Springer-Verlag.

Silva de Oliveira, D. and Rosa, N. (2010a). Evaluating Product Line Architecture for Grid Computing Middleware Systems: Ubá experience. In *Advanced Information Networking and Applications Workshops (WAINA), 2010 IEEE 24th International Conference on*, pages 257 –262.

Silva de Oliveira, D. J. and Rosa, N. S. (2010b). Evaluating product line architecture for grid computing middleware systems: Ubá experience. In *Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, WAINA '10, pages 257–262, Washington, DC, USA. IEEE Computer Society.

Simidchieva, B. I. and Osterweil, L. J. (2010). Categorizing and modeling variation in families of systems: a position paper. In I. Gorton, C. E. Cuesta, and M. A. Babar, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Companion Volume*, ACM International Conference Proceeding Series, pages 316–323. ACM.

Taulavuori, A., Niemelä, E., and Kallio, P. (2004). Component documentation—a key issue in software product lines. *Information and Software Technology*, **46**(8), 535 – 546.

Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley, 1 edition.

Tekinerdogan, B. and Sözer, H. (2012). Variability viewpoint for introducing variability in software architecture viewpoints. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, pages 163–166, New York, NY, USA. ACM.

Thiel, S. and Hein, A. (2002). Systematic Integration of Variability into Product Line Architecture Design. In G. J. Chastek, editor, *Software Product Lines, Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22, 2002, Proceedings*, volume 2379 of *Lecture Notes in Computer Science*, pages 130–153. Springer.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2012). Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*. to appear; accepted 7 Jun 2012.

Tizzei, L. P., Dias, M. O., Rubira, C. M. F., Garcia, A., and Lee, J. (2011). Components meet aspects: Assessing design stability of a software product line. *Information & Software Technology*, **53**(2), 121–136.

Trujillo, S., Azanza, M., Diaz, O., and Capilla, R. (2007). Exploring Extensibility of Architectural Design Decisions. In *Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent, 2007. SHARK/ADI '07: ICSE Workshops 2007. Second Workshop on*.

van der Hoek, A. (2004). Design-time product line architectures for any-time variability. *Sci. Comput. Program.*, **53**(3), 285–304.

van der Hoek, A., Dincel, E., and Medvidovic, N. (2003). Using service utilization metrics to assess the structure of product line architectures. In *9th IEEE International Software Metrics Symposium (METRICS 2003), 3-5 September 2003, Sydney, Australia*, pages 298–308. IEEE Computer Society.

van der Linden, F. J., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, Berlin.

van Gurp, J. and Bosch, J. (2002). Design erosion: problems and causes. *J. Syst. Softw.*, **61**(2), 105–119.

Weiss, D. M. and Lai, C. T. R. (1999). *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Wieringa, R., Maiden, N., Mead, N., and Rolland, C. (2005). Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requir. Eng.*, **11**(1), 102–107.

Zhang, H., Babar, M. A., and Tell, P. (2011). Identifying relevant studies in software engineering. *Information and Software Technology*, **53**(6), 625 – 637. <ce:title>Special Section: Best papers from the APSEC</ce:title> <ce:subtitle>Best papers from the APSEC</ce:subtitle>.

Zhang, T., Deng, L., Wu, J., Zhou, Q., and Ma, C. (2008). Some metrics for Accessing Quality of Product Line Architecture. In *International Conference on Computer Science and Software Engineering, CSSE 2008, Volume 2: Software Engineering, December 12-14, 2008, Wuhan, China*, pages 500–503. IEEE Computer Society.

Zheng, L., Wu, Z., Zhang, C., and Yang, F. (2010). Developing an Architecture Description Language for Data Processing Product Line. In S. Latifi, editor, *Seventh International Conference on Information Technology: New Generations, ITNG 2010, Las Vegas, Nevada, USA, 12-14 April 2010*, pages 944–949. IEEE Computer Society.

Appendix

Mapping Study

A.1 List of Journals Manually Searched

Journal
ACM Computing Surveys
ACM Transactions on Software Engineering and Methodology
Annals of Software Engineering
Communications of the ACM
Empirical Software Engineering Journal
IEEE Software
IEEE Transactions on Software Engineering
IET Software
Information and Software Technology
Journal of Object Technology
Journal of Systems and Software
Journal of Systems Architecture
Management Science
Software Practice and Experience Journal
Software Process: Improvement and Practice

A.2 List of Conferences Manually Searched

Acronym	Conference
APSEC	Asia Pacific Software Engineering Conference
ASE	International Conference on Automated Software Engineering
ASWEC	Australian Software Engineering Conference
CAiSE	International Conference on Advanced Information Systems Engineering
CBSE	International Symposium on Component-based Software Engineering
COMPSAC	International Computer Software and Applications Conference
ECBS	International Conference and Workshop on the Engineering of Computer Based Systems
ECOOP	European Conference for Object-Oriented Programming
ECSA	European Conference on Software Architecture
ESEC	European Software Engineering Conference
ESEM	Empirical Software Engineering and Measurement
FASE	Fundamental Approaches to Software Engineering
ICEIS	International Conference on Enterprise Information Systems
ICPC	International Conference on Program Comprehension
ICSE	International Conference on Software Engineering
ICSM	International Conference on Software Maintenance
ICSR	International Conference on Software Reuse
IRI	International Conference on Information Reuse and Integration
GPCE	International Conference on Generative Programming and Component Engineering
OOPSLA	ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications
PFE	Software Product-Family Engineering
PROFES	International Conference on Product Focused Software Development and Process Improvement
QoSA	International Conference on the Quality of Software Architectures
QSIC	International Conference on Quality Software
SEAA	Euromicro Conference on Software Engineering and Advanced Applications
SEKE	International Conference on Software Engineering and Knowledge Engineering
SPLC	Software Product Line Conference
VaMoS	Variability Modeling of Software-intensive Systems
WICSA	Working IEEE/IFIP Conference on Software Architecture

A.3 Data Extraction Report

Data Extraction Report: Systematic Mapping on PLAs
General Information Study Identifier: (Unique ID for the study) Date of data extraction: Data extractor: Data checker:
Study Description Objectives: Problem (main problem approached in the study): Results (contributions and outcomes found): Classification (based on the categories and facets):
Answers to Research Questions Q1. Are architectural patterns (styles) used in SPL? Q2. How is variability handled in the architecture level of SPLs? Q3. How are the SPL architectures documented? Q4. How are the SPL architectures evaluated?

Table A.1 Data extraction report

A.4 Primary Studies Mapped According to Aspects in PLA

(following pages in landscape)

Table A.2 Primary Studies addressing Patterns

Title	Reference
A feature-oriented approach for developing reusable product line assets of service-based systems	(Lee *et al.*, 2010)
An architectural style of product lines for distributed processing systems, and practical selection method	(Morisawa and Torii, 2001)
Applying Patterns to Develop a Product Line Architecture for Statistical Analysis Software	(Meister *et al.*, 2004)
Architecting families of software systems with process algebras	(Bernardo *et al.*, 2002)
Architecture Patterns for Mobile Games Product Lines	(Cho and Yang, 2008)
Building domain specific software architectures from software architectural design patterns	(Fant, 2011)
DRAMA: A framework for domain requirements analysis and modeling architectures in software product lines	(Kim *et al.*, 2008a)
Patterns in Product Family Architecture Design	(Hallsteinsen *et al.*, 2003)
Scenarios, Quality Attributes, and Patterns: Capturing and Using their Synergistic Relationships for Product Line Architectures	(Babar, 2004)

Table A.3 Primary Studies addressing Variability

Title	Reference
A Metamodel Approach to Architecture Variability in a Product Line	(Moon *et al.*, 2006)
A model for variability design rationale in SPL	(Galvão *et al.*, 2010)
ADLARS: An Architecture Description Language for Software Product Lines	(Bashroush *et al.*, 2006)
An Approach for Modelling Software Product Line Architecture	(Lin *et al.*, 2010)
An approach to developing domain architectures based on variability analysis	(Moon and Yeom, 2006)
An Approach to Extension of UML 2.0 for Representing Variabilities	(Choi *et al.*, 2005)
Analysis of Software Product Line Architecture Representation Mechanisms	(Ahn and Kang, 2011)
Architecting families of software systems with process algebras	(Bernardo *et al.*, 2002)
Architecture Recovery for Product Families	(Pinzger *et al.*, 2003)
Categorizing and modeling variation in families of systems: a position paper	(Simidchieva and Osterweil, 2010)
Change Impact Analysis in Product-Line Architectures	(Díaz *et al.*, 2011)
Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA	(Matinlassi, 2004a)
Component-based product line development: the KobrA Approach	(Atkinson *et al.*, 2000)
Dealing with Variability in Architecture Descriptions to Support Automotive Product Lines	(Mann and Rock, 2009)
Decision-Oriented Modeling of Product Line Architectures	(Dhungana *et al.*, 2007)
Delta-oriented architectural variability using MontiCore	(Haber *et al.*, 2011a)
Describing variability in service-oriented software product lines	(Galster, 2010)
Design-time product line architectures for any-time variability	(van der Hoek, 2004)
Designing runtime variation points in product line architectures: three cases	(Goedicke *et al.*, 2004)
DRAMA: A framework for domain requirements analysis and modeling architectures in software product lines	(Kim *et al.*, 2008a)
EPM: a prototype tool for variability management in component hierarchies	(Abele *et al.*, 2012)
Explicit exception handling variability in component-based product line architectures	(Bertoncello *et al.*, 2008)
Feature Based Variability for Service Oriented Architectures	(Abu-Matar and Gomaa, 2011)
Hierarchical Variability Modeling for Software Architectures	(Haber *et al.*, 2011b)
Improving software product line configuration: A quality attribute-driven approach	(Guana and Correal, 2013)
Initiating layers architecture design for Software Product Line	(Murwantara, 2011)
Integrating Variability Management and Software Architecture	(Groher and Weinreich, 2012)
Making Variability Decisions during Architecture Design	(Bass *et al.*, 2003b)
Managing the quality of software product line architectures through reusable model transformations	(Kavimandan *et al.*, 2011)
Managing variability in software architectures	(Bachmann and Bass, 2001)
Model-based verification of quantitative non-functional properties for software product lines	(Ghezzi and Molzam Sharifloo, 2013)
Modeling and validation of business process families	(GröNer *et al.*, 2013)
Modeling Product Line Architectures through Change Sets and Relationships	(Hendrickson and van der Hoek, 2007)
Modeling Variability in the Component and Connector View of Architecture Using UML	(Razavian and Khosravi, 2008)
On representing variation	(Hilliard, 2010)
On the Role of Architectural Design Decisions in Software Product Line Engineering	(Capilla and Babar, 2008)
Plastic Partial Components: A solution to support variability in architectural components	(Pérez *et al.*, 2009)
Reference architectures and variability: current status and future perspectives	(Nakagawa, 2012)
Scenario-based decision making for architectural variability in product families	(America *et al.*, 2005)
Systematic Integration of Variability into Product Line Architecture Design	(Thiel and Hein, 2002)
Variabilities as first-class elements in product line architectures of homecare systems	(Carvalho *et al.*, 2012)
Variability Management for Software Product-line Architecture Development	(Kim *et al.*, 2011)
Variability Management in Aspect-Oriented Architecture Description Languages: An Integrated Approach	(Adachi *et al.*, 2009)

Table A.4 Primary Studies addressing Documentation

Title	Reference
A lightweight Language for Software Product Lines Architecture Description	(Silva *et al.*, 2013)
ADLARS: An Architecture Description Language for Software Product Lines	(Bashroush *et al.*, 2006)
An Approach to Extension of UML 2.0 for Representing Variabilities	(Choi *et al.*, 2005)
An industrial case of exploiting product line architectures in agile software development	(Babar *et al.*, 2009)
Analysis of Software Product Line Architecture Representation Mechanisms	(Ahn and Kang, 2011)
Architecting families of software systems with process algebras	(Bernardo *et al.*, 2002)
Component documentation - a key issue in software product lines	(Taulavuori *et al.*, 2004)
Developing an architecture description language for data processing product line	(Zheng *et al.*, 2010)
Exploring Extensibility of Architectural Design Decisions	(Trujillo *et al.*, 2007)
Identifying traceability between feature model and software architecture in software product line using formal concept analysis	(Satyananda *et al.*, 2007a)
Initiating layers architecture design for Software Product Line	(Murwantara, 2011)
Mapping features to reusable components: A problem frames-based approach	(Dao and Kang, 2010)
Model-driven derivation of product architectures	(Botterweck *et al.*, 2007)
Model-driven development of families of Service-Oriented Architectures	(Asadi *et al.*, 2009)
Model-Driven Product Line Architectures	(Muthig and Atkinson, 2002)
Modeling Product Line Architectures through Change Sets and Relationships	(Hendrickson and van der Hoek, 2007)
Modeling Variability in the Component and Connector View of Architecture Using UML	(Razavian and Khosravi, 2008)
Object Oriented Analysis and Modeling for Families of Systems with UML	(Gomaa, 2000)
PL-AspectualACME: An Aspect-Oriented Architectural Description Language for Software Product Lines	(Barbosa *et al.*, 2011)
Plastic Partial Components: A solution to support variability in architectural components	(Pérez *et al.*, 2009)
Representing Product Family Architectures in an Extensible Architecture Description Language	(Dashofy and Hoek, 2002c)
Reuse of architectural knowledge in SPL development	(Rossel *et al.*, 2009)
Scenario-based decision making for architectural variability in product families	(America *et al.*, 2005)
Scenarios, Quality Attributes, and Patterns: Capturing and Using their Synergistic Relationships for Product Line Architectures	(Babar, 2004)
Systematic Integration of Variability into Product Line Architecture Design	(Thiel and Hein, 2002)
Tracking degradation in software product lines through measurement of design rule violations	(Johansson and Höst, 2002)
Uba: A Software Product Line Architecture for Grid-Oriented Middleware	(Oliveira and Rosa, 2009)
Utilizing domain models for application design and validation	(Reinhartz-Berger and Sturm, 2009)
Variability Management in Aspect-Oriented Architecture Description Languages: An Integrated Approach	(Adachi *et al.*, 2009)

Table A.5 Primary Studies addressing Evaluation

Title	Reference
A Case Study on SW Product Line Architecture Evaluation: Experience in the Consumer Electronics Domain	(Kim *et al.*, 2008b)
A classification of software reference architectures: Analyzing their success and effectiveness	(Angelov *et al.*, 2009)
A holistic architecture assessment method for software product lines	(Olumofin and Misic, 2007)
A meta-process to support trade-off analysis in software product line architecture	(de Oliveira Junior *et al.*, 2011)
A Regression Testing Approach for Software Product Lines Architectures	(Neto *et al.*, 2010)
An approach to architectural analysis of product lines	(Gannod and Lutz, 2000)
Analysis of a software product line architecture: an experience report	(Lutz and Gannod, 2003)
Change Impact Analysis in Product-Line Architectures	(Díaz *et al.*, 2011)
Components meet aspects: Assessing design stability of a software product line	(Tizzei *et al.*, 2011)
Continuous evolution through software architecture evaluation: a case study	(Rosso, 2006)
Design Verification for Product Line Development	(Kishi *et al.*, 2005)
Empirical Validation of Complexity and Extensibility Metrics for Software Product Line Architectures	(Oliveira Junior *et al.*, 2010)
Evaluating Product Line Architecture for Grid Computing Middleware Systems: Uba Experience	(Silva de Oliveira and Rosa, 2010a)
Evaluating the Portability and Maintainability of Software Product Family Architecture: Terminal Software Case Study	(Matinlassi, 2004b)
Experiences in assessing product family software architecture for evolution	(Maccari, 2002)
Extending ATAM to assess product line architecture	(Kim *et al.*, 2008c)
Extending the ATAM Architecture Evaluation to Product Line Architectures	(Olumofin and Misic, 2005)
Extending the RiPLE-DE process with quality attribute variability realization	(Cavalcanti *et al.*, 2011)
Formal Verification of Consistency between Feature Model and Software Architecture in Software Product Line	(Satyananda *et al.*, 2007b)
Making testing product lines more efficient by improving the testability of product line architectures	(Kolb and Muthig, 2006)
Managing the quality of software product line architectures through reusable model transformations	(Kavimandan *et al.*, 2011)
Measuring Product Line Architectures	(Dincel *et al.*, 2001)
Model-based verification of quantitative non-functional properties for software product lines	(Ghezzi and Molzam Sharifloo, 2013)
Practical Evaluation of Software Product Family Architectures	(Niemelä *et al.*, 2004)
Reference architectures and variability: current status and future perspectives	(Nakagawa, 2012)
Some Metrics for Accessing Quality of Product Line Architecture	(Zhang *et al.*, 2008)
Systematic Evaluation of Software Product Line Architectures	(Oliveira Junior *et al.*, 2013)
Tracking degradation in software product lines through measurement of design rule violations	(Johansson and Höst, 2002)
Using service utilization metrics to assess the structure of product line architectures	(van der Hoek *et al.*, 2003)
Variability Driven Quality Evaluation in Software Product Lines	(Etxeberria and Sagardui, 2008)
Vulcan: Architecture-model-based workbench for product line engineering	(Lee *et al.*, 2012)

B
Exploratory Study

B.1 Notepad SPL Architecture Specification

Component: ***Fonts***

Required Interface: Contents handler [GUI]
ActionListener();
JLabel(); *JDialog
JPanel();
JComboBox();
 	
Provided Interface: Font setting [Actions]
setFont();
setVisible();
pack();
getOkjb();
getCajb();

Component: ***Files***
Provided Interface: File handler [Actions] addExtension(); *Filter setDescription(); *Filter setFileChooser(); *Chooser PrintWriter(); *Write File StringTokenizer(); *Tokens file content

Component: *Notepad*

Required Interface: Features [Actions]
newFile (textPane, Notepad);
open (textPane);
save_as (textPane);
save (textPane);
exit (textPane);
copy (textPane);
cut (textPane);
paste (textPane);
select_all (textPane);
find (textPane);
find_next (textPane);
select_found (textPane);
fonT (textPane);
undo (UndoManager);
redo (UndoManager);
about ();

Required Interface: Contents handler *textPane [GUI]
ActionListener();
getTextComponent();
getScreenSize(); *Dimension
getContentPane(); *Container

Required Interface: Undo/Redo manager [Undo/Redo]
undo();

Component: *Undo/Redo*

Provided Interface: Undo manager [Notepad]
undo();
redo();

Provided Interface: Undo manager [Actions]
undo();
redo();

Component: *Actions*

Required Interface: File Handler [Files]
addExtension(); * Filter
setDescription(); * Filter
setFileChooser(); * Chooser
PrintWriter(); * Write File
StringTokenizer(); *Tokens file content

Required Interface: Contents handler *textPane, JFrame, JLabel, Dimension [GUI]
getText();
frame.(config)(label);
Toolkit.getScreenSize() *Dimension
addActionListener(); *Action event

Required Interface: Font setting [Fonts]
setFont();
setVisible();
pack();
getOkjb();
getCajb();

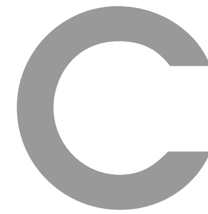
Required Interface: Undo manager [Undo/Redo]
undo();
redo();

Provided Interface: Features [Notepad]
newFile (textPane, Notepad);
open (textPane);
save_as (textPane);
save (textPane);
exit (textPane);
copy (textPane);
cut (textPane);
paste (textPane);
select_all (textPane);
find (textPane);
find_next (textPane);
select_found (textPane);
fonT (textPane);
undo (UndoManager);
redo (UndoManager);
about ();

Component: *GUI*

Provided Interface: Contents handler [Font]
getText();
frame.(config)();
ActionListener();
getTextComponent();
getScreenSize(); *Dimension
getContentPane(); *Container
JLabel(); *JDialog
JPanel();
JComboBox();

Provided Interface: Contents handler *textPane [Notepad]
ActionListener();
getTextComponent();
getScreenSize(); *Dimension
getContentPane(); *Container



Replicated Study

C.1 RescueMe SPL Architecture Specification

C.1.1 Model

Component: *Contact*

Provided Interface: Contact Selection / Changes [Contact Manager]
getAllContacts();
contactAlreadyExists();
addContactWithName();
removeContact();
saveContact();
deleteContact();
persistenceCoordinator();
getAllPhones();
getAllEmails();
getAllTwitterIDs();
getAllFacebookIDs();
getAllDataOf: String();

Component: *Location*

Provided Interface: Position [Destination]
latitude;
longitude;
locationUpdate(CLLocation);
locationError();

C.1.2 View

Component: *GUI*

Provided Interface: Contact Selection [Contact View Controller]
viewDidLoad();
viewDidUnload();
viewWillAppear();
shouldAutorotateToInterfaceOrientation();
viewAddContactScreen();
viewImportContactScreen();
numberOfSelectionsInTableView();
numberOfRowsInSelection();
cellForRowAtIndexPath();
buttonDeletePressed();
clickedButtonAtIndex();
didSelectRowAtIndexPath();

Provided Interface: Message Trigger [Destination]
redButtonPressed(UIButton);
viewDidLoad();
viewDidUnload();
didReceiveMemoryWarning();
messageComposeViewController();

C.1.3 Controller

Component: *Destination*

Required Interface: Recipients [Contact Manager]
emailDestination; // NSArray *toRecipients
SMSDestination; //getAllPhones

Required Interface: Message Trigger [GUI]
redButtonPressed(UIButton);
viewDidLoad();
viewDidUnload();
didReceiveMemoryWarning();
messageComposeViewController();

Required Interface: Position [Location]
latitude;
longitude;
locationUpdate(CLLocation);
locationError();

Required Interface: Twitter Communication [Twitter API]
TwitterCommunicationController;

Required Interface: Facebook Communication [Facebook API]
FacebookCommunicationController;

Required Interface: Message [Message Composer]
emailDestination; // NSString *emailTitle
emailDestination; // NSString *messageBody
selectedLanguage;
getRescueMessage();
mailComposeController();

Component: *Contact View Controller*

Required Interface: Contacts [Contact Manager]
contactsList;
managedObjectContext;
sortDescriptors;
namesInTheList;

Required Interface: Contact Selection [GUI]
viewDidLoad();
viewDidUnload();
viewWillAppear();
shouldAutorotateToInterfaceOrientation();
viewAddContactScreen();
viewImportContactScreen();
numberOfSelectionsInTableView();
numberOfRowsInSelection();
cellForRowAtIndexPath();
buttonDeletePressed();
clickedButtonAtIndex();
didSelectRowAtIndexPath();

Required Interface: Facebook Communication [Facebook API]
initWithContactManager();
facebookImport(id sender);
facebookViewControllerDoneWasPressed (id sender);
showAlertWithContactName();
peoplePickerNavigationController();
peoplePickerNavigationControllerDidCancel();
viewDidLoad();
viewDidUnload();

Required Interface: Twitter Communication [Twitter API]
initWithContactManager();
twitterImport(id sender);
showAlertWithContactName();
peoplePickerNavigationController();
peoplePickerNavigationControllerDidCancel();
viewDidLoad();
viewDidUnload();

Required Interface: iOS Communication [iOS API]
initWithContactManager();
phoneImport(id sender);
getContactInfo();

Component: ***Contact Manager***

Required Interface: Contact Selection / Changes [Contact]
getAllContacts();
contactAlreadyExists();
addContactWithName();
removeContact();
saveContact();
deleteContact();
persistenceCoordinator();
getAllPhones();
getAllEmails();
getAllTwitterIDs();
getAllFacebookIDs();
getAllDataOf: String();
 	
Provided Interface: Contacts [Contact View Controller]
contactsList;
setVisiblemanagedObjectContext;
sortDescriptors;
namesInTheList;
 	
Provided Interface: Recipients [Destination]
emailDestination; // NSArray *toRecipients
SMSDestination; // getAllPhones

Component: ***Message Composer***

Required Interface: iOS Communication [iOS API]
emailDestination();
SMSDestination();
 	
Provided Interface: Message [Destination]
emailDestination; // NSString *emailTitle
emailDestination; // NSString *messageBody
selectedLanguage;
getRescueMessage();
mailComposeController();

C.1.4 3rd Party

Component: ***Facebook API***

Provided Interface: Facebook Communication [Contact View Controller]
initWithContactManager();
facebookImport(id sender);
facebookViewControllerDoneWasPressed (id sender);
showAlertWithContactName();
peoplePickerNavigationController();
peoplePickerNavigationControllerDidCancel();
viewDidLoad();
viewDidUnload();

Provided Interface: Facebook Communication [Destination]
FacebookCommunicationController;

Component: ***Twitter API***

Provided Interface: Twitter Communication [Contact View Controller]
initWithContactManager();
twitterImport(id sender);
showAlertWithContactName();
peoplePickerNavigationController();
peoplePickerNavigationControllerDidCancel();
viewDidLoad();
viewDidUnload();

Provided Interface: Twitter Communication [Destination]
TwitterCommunicationController;

Component: *iOS API*

Provided Interface: iOS Communication [Contact View Controller]
initWithContactManager();
phoneImport(id sender);
getContactInfo();
 	
Provided Interface: iOS Communication [Message Composer]
emailDestination();
SMSDestination();