Universidade Federal da Bahia

Universidade Salvador

Universidade Estadual de Feira de Santana

TESE DE DOUTORADO

Fault Model-Based Variability Testing

Ivan do Carmo Machado

Programa Multiinstitucional de

Pós-Graduação em Ciência da Computação – PMCC

Salvador-BA

2014

PMCC-DSc-0015

IVAN DO CARMO MACHADO

FAULT MODEL-BASED VARIABILITY TESTING

Tese apresentada ao Programa Multiinstitucional de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, Universidade Salvador e Universidade Estadual de Feira de Santana como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientador: *Eduardo Santana de Almeida*

Salvador - BA
2014

Sistema de Bibliotecas da UFBA

Machado, Ivan do Carmo.
Fault model-based variability testing / Ivan do Carmo Machado. - 2014.
187 p.: il.

Inclui apêndices.
Orientador: Eduardo Santana de Almeida.
Tese (doutorado) - Universidade Federal da Bahia, Instituto de Matemática. Universidade Salvador, Universidade Estadual de Feira de Santana, 2014.

1. Software. 2. Software - Reutilização. 3. Software - Testes. 4. Software - Validação. I. Almeida, Eduardo Santana de. II. Universidade Federal da Bahia. Instituto de Matemática. III. Universidade Salvador. IV. Universidade Estadual de Feira de Santana. V. Título.

CDD - 005.1
CDU - 004.4

IVAN DO CARMO MACHADO

FAULT MODEL-BASED VARIABILITY TESTING

Esta tese foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa Multi-institucional de Pós-Graduação em Ciência da Computação da UFBA-UEFS-UNIFACS.

Salvador, 21 de julho de 2014.

Prof Dr. Eduardo Santana de Almeida (orientador).
Universidade Federal da Bahia

Prof^{a} Dr^{a}. Christina von Flach Garcia Chavez.
Universidade Federal da Bahia

Prof Dr. Cláudio Nogueira Sant'Anna.
Universidade Federal da Bahia

Prof Dr. Marco Tulio de Oliveira Valente.
Universidade Federal de Minas Gerais

Prof Dr. Vander Ramos Alves.
Universidade de Brasília.

Aos meus pais, Serafim e Joselice!!

Agradecimentos

A Deus, em primeiro lugar! A Ele toda honra e toda a Glória! Nada faria sentido sem a Sua presença em minha vida! Muito obrigado meu Senhor e meu Deus, por tudo!

Expresso aqui nestas poucas porém significativas palavras, imensa gratidão a minha família, que sempre me apoiou, incondicionalmente, na busca dos meus sonhos, objetivos e ideais. Muito obrigado por compreenderem a importância da conquista desse importante título acadêmico. A certeza de saber que poderia sempre contar com vocês foi fundamental para sobreviver aos altos e baixos do período de doutoramento. O fruto de todo este esforço é a vós que dedico! Muito obrigado meu pai, Serafim, minha mãe, Joselice, meu irmão Ivo, e minha segunda mãe, Benedita (Dil). São vocês o meu porto seguro.

Meus sinceros agradecimentos a minha noiva, Edna Telma, que com muito amor e carinho me fez ter forças para seguir, por ter suportado tanta ausência e eventuais estados de mal-humor. Agradeço-lhe por ter aderido a condição de parceira de um doutorando, o que por si só demonstra persistência e resiliência. Muito obrigado por acreditar que toda a espera terá valido a pena!

"A mente que se abre a uma nova ideia jamais voltará ao seu tamanho original". Tal frase, proferida outrora por um gênio, retrata fielmente a jornada de um doutorando. Sou bastante grato pela oportunidade de aprender com pesquisadores tão renomados na área de pesquisa sobre a qual me debrucei para realizar o presente trabalho. Uma série de encontros que permitiram fantásticas troca de experiências, que de uma forma ou de outra, contribuíram para a pesquisa desenvolvida.

Assim, agradeço ao meu orientador, Eduardo Almeida, pelo acolhimento e pelas oportunidades concedidas, pelas conversas francas e as experiências transmitidas ao longo desses pouco mais de seis anos de convivência.

Agradeço também ao professor John McGregor, da Universidade de Clemson, nos EUA, por sua importante contribuição a minha pesquisa. Durante o doutorado, passei sete meses em sua universidade, onde aprendi muitas coisas. Sou profundamente grato pelo acolhimento e também por esta experiência de vida, muito além do que meramente uma experiência acadêmica ou profissional.

Agradeço ao professor Klaus Schmid, da Universidade de Hildesheim, Alemanha, que, durante período sabático em nossa universidade, pode compartilhar um pouco da sua experiência com os membros do grupo RiSE *(Reuse in Software Engineering)*. Agradeço-lhe por disponibilizar a plataforma EASy-Producer, para a realização de um dos estudos empíricos, bem como o auxílio na definição do experimento.

Obrigado também aos membros da banca, Christina von Flach Garcia Chavez, Vander

Ramos Alves, Marco Túlio de Oliveira Valente, e Cláudio Nogueira Sant'Anna. Seus valiosos comentários durante a defesa ajudaram a enriquecer muito este trabalho, bem como direcionar pesquisas futuras.

Grandes mestres, os professores são essenciais em nossas vidas. Desta forma, agradeço profundamente aos professores do programa de pós-graduação em computação PMCC/UFBA, pelo aprendizado. Foi uma honra ter recebido atenção e ensinamentos de pessoas tão qualificadas.

Sou muito grato aos funcionários do CEAPG-MAT/UFBA pela atenção, disponibilidade e presteza ímpares, ao longo desses pouco mais de quatro anos, e diversas solicitações. Obrigado a vocês Davilene, Solange, Marcio, Gustavo e Kleber!

Agradeço a todos os amigos e parceiros que fizeram ou fazem parte do grupo RiSE e do Laboratório de Engenharia de Software da UFBA. Sinto-me privilegiado por ter convivido com as pessoas que passaram por esses grupos de pesquisa, em especial a Luanna Lobato, Yguaratã Cavalcanti, Paulo Silveira, Renato Novais, Raphael Oliveira, Alcemir Santos, pessoas que deram significativa contribuição em diversas oportunidades ao longo do processo de elaboraçao desta tese. Evidências empíricas indicam que a obtenção de bons resultados de pesquisa é algo diretamente proporcional a qualidade de boas parcerias estabelecidas ao longo do processo. Que bom que pude contar com vocês!

Agradeço também ao Instituto Recôncavo de Tecnologia - IRT, em especial a Marcio Magalhães, Eduardo Trzan, Paula Ferreira, por proporcionarem um ambiente de colaboração ímpar. A parceria trouxe bons resultados. Que possamos produzir ainda mais em conjunto!

Agradeço ao apoio financeiro da FAPESB - Fundação de Amparo a Pesquisa do Estado da Bahia, CAPES, CAPES-PDSE, sem as bolsas, em seus respectivos momentos, não seria possível realizar este trabalho. Agradeço ainda a IBM Research, por ter reconhecido a minha dedicação e assim ter concedido um importante prêmio, o IBM Ph.D. Fellowship 2012-2013.

Por fim, muito obrigado a todos os demais que passaram pelo meu caminho, nestes pouco mais de quatro anos. Alguma lição ficou em cada encontro ou desencontro!

Vamos em frente que a caminhada está só começando! ;)

Imperfect tests, run frequently, are much better than perfect tests that are never written at all.

—MARTIN FOWLER

Abstract

Software Product Line (SPL) engineering has emerged as an important strategy to cope with the increasing demand of large-scale product customization. Owing to its variability management capabilities, SPL has provided companies with an efficient and effective means of delivering a set of products with higher quality at a lower cost, when compared to traditional software engineering strategies. However, such a benefit does not come for free. SPL demands cost-effective quality assurance techniques that attempt to minimize the overall effort, while improving, or at least not hurting, fault detection rates. Software testing, the most widely used approach for improving software quality in practice, has been largely explored to address this particular topic.

State of the art SPL testing techniques are mainly focused on handling variability testing from a high level perspective, namely through the analysis of feature models, rather than concerning issues from a source code perspective. However, we believe that improvements in the quality of variable assets entail addressing testing issues both from high and low-level perspectives.

By carrying out a series of empirical studies, gathering evidence from both the literature and the analysis of defects reported in three open source software systems, we identified and analyzed commonly reported defects from Java-based variability implementation mechanisms. Based on such evidence, we designed an approach for building fault models for variability testing, from two main perspectives: test assessment, which focuses on the evaluation of the effectiveness of existing test suites; and test design, which aims to aid the construction of test sets, by focusing on fault-prone elements.

The task of modeling typical or important faults provides a means to coming up with certain test inpus that can expose faults in the program unit. Hence, we hypothesize that understanding the nature of typical or important faults prior to developing the test sets would enhance their capability to find a particular set of errors.

We performed a controlled experiment to assess the test effectiveness of using fault models to provide SPL testing with support to design test inputs. We observed promising results that confirm the hypothesis that combining fault models in an SPL testing process performs significantly better on improving the quality of test inputs.

Keywords: Software product line enginering, software testing, variability testing, fault model.

Resumo

A Engenharia de Linhas de Produtos de Software (LPS) surgiu como uma importante estratégia para lidar com a crescente demanda de customização de produtos de software em larga escala. Por sua capacidade de gerenciar variabilidade de forma sistemática, o paradigma de LPS tem proporcionado às empresas métodos eficientes e eficazes para alcançar a entrega de produtos de software com maior qualidade, a um custo de produção reduzido, quando comparado a estratégias tradicionais de desenvolvimento de software. No entanto, a obtenção de tais benefícios não é trivial. O paradigma impõe a necessidade de técnicas de garantia de qualidade eficazes, com bom custo-benefício, que tentem minimizar o esforço global, ao tempo em que se alcance melhorias nas taxas de detecção de falhas. Assim, a disciplina de testes de software, abordagem comumente utilizada na busca por melhoria na qualidade dos produtos de software, tem sido largamente explorada no contexto de LPS.

As mais relevantes técnicas de testes em LPS estão focadas principalmente no gerenciamento de testes de variabilidade sob uma perspectiva de alto nível, notadamente através da análise de modelos, em sobreposição aos aspectos de mais baixo nível, isto é, sob o ponto de vista do código fonte. Entretanto, acreditamos que melhorias na qualidade dos artefatos de software variáveis implica na investigação de aspectos da disciplina de testes, em ambas as perspectivas, quer seja alto nível quer seja baixo nível.

Através da realização de uma série de estudos empíricos, evidências foram obtidas a partir da análise de textos publicados na literatura, e a partir da análise de defeitos reportados em três sistemas de software de código aberto. Neste último caso, identificamos e analisamos defeitos provenientes do uso de mecanismos de implementação de variabilidade em Java. Com base nas evidências, construímos uma abordagem para construir modelos de falhas que auxiliem o teste de variabilidade, sob duas perspectivas principais: avaliação de teste, que incide sobre a avaliação da eficácia dos casos de testes existentes; e o projeto de teste, que visa auxiliar a construção de casos de teste, concentrando-se em elementos propensos a falhas.

A tarefa de modelagem de falhas típicas ou importantes fornece um meio para identificar certas entradas de teste que podem expor falhas na execução do programa. Desta forma, a nossa hipótese é que a compreensão da natureza das falhas típicas, ou importantes, como tarefa anterior ao desenvolvimento dos casos de teste, tende a aumentar a capacidade dos testes em encontrar um determinado conjunto de defeitos, quando executados.

Para avaliar a eficácia da abordagem proposta nesta tese, planejamos e executamos um experimento controlado. Os resultados mostraram-se promissores, provendo indícios de que a ideia de se combinar modelos de falha em um processo de teste de LPS pode trazer ganhos significativos a atividade de teste, bem como melhorar a qualidade dos dados de entrada de testes.

Palavras-chave: Linhas de produtos de software, testes de software, teste de variabilidade, modelos de falhas.

Contents

List of Figures

List of Tables

List of Acronyms

APFD	Average Percentage of Fault Detected
CBD	Component-Based Development
CR	Change Request
FODA	Feature-Oriented Domain Analysis
GQM	Goal Question Metric
LLD	Low-Level Design
ODC	Orthogonal Defect Classification
OVM	Orthogonal Variability Model
OOP	Object-Oriented Programming
OS	Operating Systems
PCA	Principal Component Analysis
PUT	Program Under Test
QDF	Quality of Defects Found
SDLC	Software Development Life Cycle
SLR	Systematic Literature Review
SOA	Service-Oriented Architecture
SPL	Software Product Lines
SUT	Software Under Test
SVM	Suport Vector Machine
TCE	Test Case Effectiveness
TDD	Test-Driven Development
V&V	Verification and Validation

1 Introduction

Software is a key element in many of the devices and systems that pervade most facets of modern life. It is inevitable to ensure the important role it plays in the society, both economically and socially. Software systems control from small gadgets to the largest civil aircraft built so far (Burger *et al.*, 2013). Together with a perceived ever-increasing complexity of software systems, software engineering has called for building better and more cost-effective methods, to cope with tight deadlines and market pressure.

SPL has proved to be an efficient and effective strategy to deal with the aforementioned demands (Clements and Northrop, 2001). As a major source for competitive advantage, SPL engineering may lead companies to achieve economies of scale and scope, and remarkable results such as substantial cost savings, reduction of time to market, and large productivity gains (Clements and McGregor, 2012).

SPL engineering is based on the idea that, by exploiting the *commonalities* and managing *variabilities* among related products, it is possible to establish a common platform on top of which a set of assets could be systematicaly reused and assembled into different products, thus meeting particular customer and/or market demands (Pohl *et al.*, 2005).

While the capability of accomodating commonalities and variabilities in a plataform may ease the process of assembling assets into a multitude of products, particular attention should be paid to the quality of the developed assets. The reason is that, as the platform may hold a high degree of variability, an uncovered defect in a single asset may be propagated to the many products that include the asset. As verifying every individual product configuration in an exhaustive manner is not always feasible in practice, more affordable strategies must be taken into account. From a software testing perspective, there are some particular characteristics that SPL engineering must accommodate. Whereas commonality testing may undertake traditional software testing techniques, and as such it is, to a certain extent, a mature field, variability testing improvements are at the core of testing in SPL engineering (McGregor *et al.*, 2004).

In this thesis, we investigate such a problem in details and propose a contribution to SPL testing by employing fault modeling concepts. This Chapter contextualizes the focus of this work and starts by presenting its motivation and a clearer definition of the research problem, in Section 1.1. Next, Section 1.2 provides details of the thesis statement, highlighting the research goals. We present the steps taken to conduct this work in Section 1.3. The main contributions are listed in Section 1.4, and finally Section 1.5 outlines the Thesis structure.

1.1 Motivation

Software testing is an important and practical technique to efficiently detect faults in complex software systems (Ammann and Offutt, 2008). A mandatory activity in SPL engineering is to ensure that an artifact holds an adequate level of quality, as it is likely to be used in a range of different product configurations (Pohl *et al.*, 2005).

Software testing techniques usually encompass one or more of the following topics (Myers *et al.*, 2011): *test case design*, when a new test set is expected; *test case selection*, whenever an existing test suite is available and it is ready to run; *test case prioritization*, a means to improve the test selection, which aims at increasing the effectiveness of testing effort within a scenario with limited resources; and *test case assessment*, employed to evaluate the capability of a test case, or a suite of test cases, to detect errors.

In traditional Software Engineering, hereinafter called *single system development*, in which every new product is built from scratch, there are techniques supporting all of the aforementioned topics. Although not as mature as in single system development, software testing has evolved in SPL engineering, in a range of topics. The initial investigation on the topic was carried out by Harrold (1998), who reported on the underlying concepts of testing evolving software systems, that would be further addressed and formalized to fit testing in the SPL engineering field. Some years later, McGregor (2001) published a Technical Report entitled *Testing a Software Product Line*, a work that served to introduce the fundamental concepts underlying SPL testing. Based on a long-term practical experience, the author pointed out a need for specific SPL testing strategies, given that common strategies from single system development could not yield the expected results in SPL engineering.

The demand for particular SPL testing approaches has led the research community to increasingly propose novel techniques, methods, and tools, as earlier discussed by Neto *et al.* (2011a) and Engström and Runeson (2011), who reported on the state-of-the-art SPL testing approaches. In general, there are two *groups of interests* addressed by the existing approaches (Machado *et al.*, 2012b): *selecting instances of products for testing (focusing on verifying if the features of a product work properly together)*, which handles how the assets can be combined so that valid products can be composed, and *testing actual products (veryifying if features fulfill their specifications)*, which cares about the actual functionalities of the produced assets. The concepts surrounding these interests are further discussed in Chapter 4.

While we recognize that both interests are important and should coexist in an SPL testing process, the latter is more closely related to the purpose of our investigation, which is aimed at improving SPL testing by employing the concept of fault models. Among the approaches dealing with such an interest, we noticed that the techniques do not take into consideration fine-grained variability, i.e., at source-code level, when desigining the SPL test cases. The studies have been focused on modeling variability in the problem domain, over the support of variability at the solution domain (Machado *et al.*, 2014a). They are mostly concerned with modeling domain variability in a higher level of abstraction, as a means to represent the common and variable features of products, and how testing can be managed thereupon. We might enlist two main likely reasons to explain this:

- The underlying assumptions of most SPL testing techniques is that, handling tests at source code level is a straighforward task, and as such, techniques from single system development suffices.
- The challenge of handling variabilities is more easily addressed at higher levels of abstractions (e.g., through models), rather than at low levels (i.e., at source code). Given that models are more abstract and less detailed than source code, variability is likely less scattered, what simplifies its management.

However, while it can facilitate understanding how products could be composed, in terms of features, it is rather important to manage variability at source code level, given that it holds important role in establishing variable behavior. Furthermore, no empirical evidence can be found in the literature to ensure that such a statement is not a simple guess.

In this effect, considering existing testing support for SPL engineering, and the SPL demands, as exemplified above, the central problem addressed in this thesis is the *lack of adequate support for the low-level variability testing in SPL engineering*.

More than verifying either if the features fulfill their specifications, or verifying if the features of a product work properly together, what usually do not have to do with source code, but high-level models instead, a challenge is to establish an understanding on how testing in an SPL could benefit from the variability awareness, i.e., it is important to use the powerful capability of the variability mechanisms to improve the quality of the SPL assets, considering large-scale gains. That is, not only using variability to define high-level tests, but also to deal with the source code particularities, so as to enable better fault coverage still while developing the artifacts.

1.2 Objectives

Predicting future is infeasible, but establishing trends based on historical data and other is rather possible. This statement is particularly important to the software testing field. The analysis of historical data enables the construction of statistical models for estimating fault-proneness of software modules before testing (Denaro *et al.*, 2002). As a consequence, testing activities can be better planned and monitored. The estimation of software fault-proneness is important for assessing costs and quality and thus better planning and tuning the testing process.

Counting on historical data to modeling the fault-prone elements can be referred to as fault modeling. It is indeed not a new idea in the software testing field. Morell (1990) early introduced a theory of fault-based program testing, stating that a fault model describes the set of known defects that can result from the activities leading to the test point. McGregor (2008) introduced the idea of fault models in SPL engineering. He presented an initial version of a fault model, which considered faults originated in the developments of assets such as requirements and design. He considered some of the characteristics of an SPL and its development organization that are the basis for the faults.

Building on top of McGregor (2008) work, and inspired by the ideas from Morell (1990), we developed a method for building fault models for variability testing. The goal is to establish an affordable strategy to design effective test cases that prioritize the fault-prone elements in variability implementation.

On the basis of such a goal, we established the research question that drives this investigation:

Can fault modeling support improve test effectiveness in variability testing?

In general, a fault model is an engineering model of something that could go wrong in the construction or operation of a piece of equipment, structure, or software (Martin and Xie, 2007).

In our case, we are modeling source code statements that could be faulty when implementing variations in an SPL project.

The expected benefits of using fault models for test case design is twofold:

- By focusing on the most fault-prone elements of the variability implementation, the testing activity should benefit from this *risk-based* capability, so as the priorities are directed to what might go wrong, first. Boehm and Basili (2001) earlier presented the results of an empirical investigation on software testing that almost 90% of software failures come from 10% of the faults. Even by considering a more flexible set of values, e.g., the 80/20 rule (The Pareto Principle (Juran *et al.*, 1979)), a broadly accepted measure to illustrate that most defects are found in a small percentage of the code (Gittens *et al.*, July), we could generalize such a statement, and have a clearer indication that, by firstly covering what might go wrong, we improve the fault detection capability of the designed test cases.

- Given that a fault model provides testers with the information about what can go wrong, it is likely that such a test case design should cover not only the expected scenarios, but also what is unexpected, so as to improve the fault detection capability of test cases. It is useful in case of software changes, which is often regarded as the common scenario in the development of software-intensive systems.

1.3 Research Method

This Section describes the research design employed as the basis for this work. Based upon the research goal, we applied a combination of methods, to both gain a fuller understanding of the research problem, and to fortify and enrich our study's conclusions (Hesse-Biber, 2010).

The present investigation can be split into two main parts: ***1.*** *Literature review* and ***2.*** *Concept establishment and evaluation*. These are detailed next.

Part 1. Literature review

This initial part of this thesis comprises the analysis of existing literature on the topics involved in this investigation, as a means of devising our research questions, and narrowing down the possibilities to consider. Besides, it also served to guide and focus the early steps of our research.

Regarding the literature on variability mechanisms, a set of reviews and empirical studies exist that provided us with the current state-of-the-art in such a field. We mainly considered the studies of (Svahnberg *et al.*, 2005) and (Chen and Babar, 2011). Both provide an extensive analysis of literature.

As part of our preceding investigation, we analyzed existing literature on the SPL testing field, and reported it as a means of a Systematic Mapping Study (Neto *et al.*, 2011a). Such a publication served us to perform a more in-depth and current analysis of existing knowledge, as a means of a Systematic Literature Review (Machado *et al.*, 2014b). This latter comprised the analysis of 273 primary studies available in the most important venues in the field of Software Engineering. We complemented the first investigation by analyzing prominent techniques for product testing in SPL engineering, pinpointing the variability potential used in those techniques. Chapter 4 presents the results of such an investigation.

Part 2. Concept

The second part represents the core of this investigation. It comprises four subparts: *a preliminary evaluation on the use of a systematic process for testing SPL projects*, *identifying common faults in variability implementation*, *building a fault classification scheme for variability implementation* and *building fault modeling support*.

Achieving better test results can be tied to the planning of the whole testing activity. On the other hand, test planning might demand systematic methods to guide test assets management. In the light of such an assumption, we investigated the role of a systematic process for testing SPL projects at improving test effectiveness. In a scenario containing low-level variability, we aimed at understanding whether the use of a systematic process could lead to better test results. As opposed to a systematic use of fault models, further addressed in the thesis, the preliminary evaluation counted on a naive approach to simulate the presence of problems in the source code under testing.

Next, we identified the common faults found in popular variability implementation mechanisms, particularly in the Java programming language, pinpointing the most commonly occurring ones. For such, we gathered data from two main sources: *(i)* the analysis of bugs from a set of *open source projects* that contain variability implementation; *(ii) informal* literature analysis, given the existence of a set of studies which report on common errors in different implementation techniques, it is possible to sketch a relationship between those and the variabitiy mechanisms. The overall goal of this task is to establish a fault classification scheme, which subsumes commonly occurring faults in variability implementation mechanisms.

The latter task, *building fault modeling support*, was backed up by data obtained from the preceding task. It is the main goal of this investigation, and mainly consists of proposing fault modeling support for variability testing.. Next, we proceeded with an evaluation, which comprised a controlled experiment performed with software engineers as participants.

1.4 Contributions

The main expected contributions of this work can be listed as follows:

- **Body of knowledge about testing in SPL engineering.** The number of studies in the SPL testing field is increasing year by year. We performed a series of structured literature reviews aiming at synthesizing state-of-the-art evidence, that can be used in a range of investigations in the field, as also to serve as comparisons with other approaches.
- **Commonly occurring defects in variability implementation.** The main goal of the thesis is to build fault model support for variability testing. In order to accomplish that, it was necessary to enlist the commonly occurring defects that are faced by variability mechanisms. We built a fault classification scheme, based on three widely used variability mechanisms for the Java programming language, that might be helpful to others that are not intended to use our proposed approach, but still work with variability implementation.
- **Fault model support for helping engineers to prioritize tests.** The core of our investigation is to propose the fault model support for variability testing. The results are promising, in that it shows it is possible to improve testing effectiveness.

1.5 Thesis outline

Figure 1.1 shows a schematic overview of the thesis structure. Besides the Introduction Chapter, the remainder can be outlined as follows:

- **Background.** This part provides background concepts on the topics involved in this investigation, namely *software testing, SPL engineering* and *SPL testing*. In addition to the basic concepts, this part also presents a state-of-the-art review of the field of testing in SPL engineering, which includes current development, controversies and breakthroughs.
  - **Chapter 2 (Software Testing Fundamentals)** presents basic fundamentals about software testing. In addition, we discuss the fundamentals of fault models, shedding light on its usage and likely benefits for software testing.

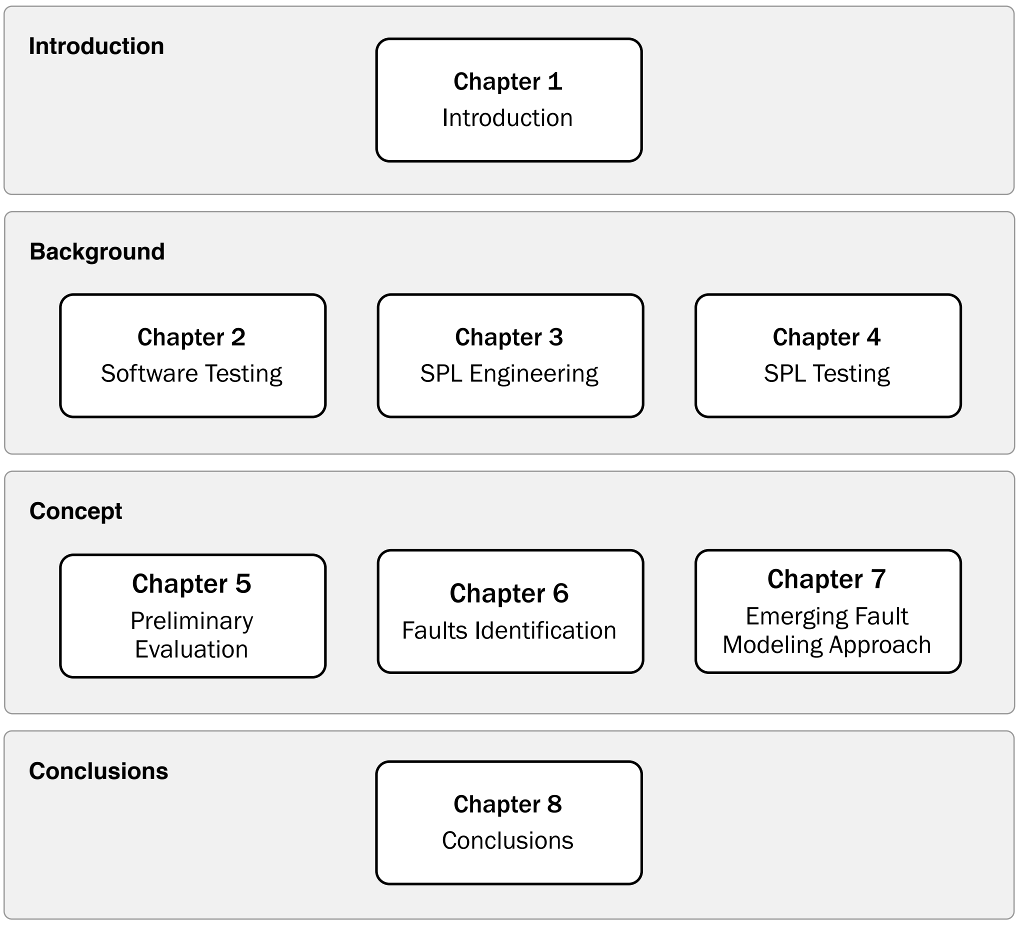


Figure 1.1: Schematic overview of the thesis structure.

- **Chapter 3 (Software Product Line Engineering)** presents the SPL engineering field, and focuses on variability and reuse, and how these concepts are managed. A clear understanding on how variability should be represented and implemented in both logical and implementation views is vital for our test approach. We left out organizational concerns an SPL demands, and rather focused on technical aspects of variability.
- **Chapter 4 (Software Product Lines Testing)** surveys the state-of-the-art of software testing strategies for SPL engineering. We reviewed and analyzed a range of studies and summarized the main findings, that should be relevant to sketch the current scenario in the field, stressing out where our approach fits.

- **Concept.** This part motivates and define in details the novel concept of fault-model based variability testing in software product line engineering.

  - **Chapter 5 (A Preliminary Evaluation of the Effects of Unit Testing in SPL Engineering)** reports on an experimental study aimed at analyzing and understanding the capabilities and effects of a proposed unit testing infrastructure for SPL projects.
  - **Chapter 6 (Defining a Fault Classification Scheme Towards Variability Testing)** introduces the effort to build up an empirical picture of commonly occurring faults from different variability implementation mechanisms.
  - **Chapter 7 (Fault Modeling for Variability Testing)** details the fault model support for variability testing, and reports on the empirical evaluation.

- **Conclusions.** In this final part, Chapter 8 (Conclusions), we summarize the achieved contributions and discuss the perspectives on future research directions.

2 Software Testing Fundamentals

Software testing is the discipline aimed at raising the quality and reliability of the software systems. Testing plays an important role in the software quality assurance. It aims at demonstrating that the system works as expected, therefore meeting customer needs, and also, by assuming the premise that any program contains errors, testing aims at finding those errors.

Indeed, the main concern of software testing is to find as many errors in the software system as possible, especially before delivering the systems, leading to reduced repair costs. This is how software testing might add some value to the software system. Boehm and Papaccio (1988) first presented a logarithmic increasing cost representation for bug fixing throughout the software development life cycle. Such a representation indicates the importance of preventing defects or detecting and removing them early, as it can realize significant cost and schedule benefits.

Myers *et al.* (2011) define testing as *"the process of executing a program with the intent of finding errors"*. Such a statement can be considered as an overall and accurate goal of software testing. For this reason, testing can only be successful if it can detect faults in a given time. If it is incapable of detecting any fault, it fails.

Extensive and *philosophical* discussions about definitions of testing goals are available in well-known references that use a vocabulary widely accepted in academia and industry (Binder, 1999; Burnstein, 2003; Ammann and Offutt, 2008; Naik and Tripathy, 2008). Along this Chapter we describe the foundations of software testing explicitly discussed in such studies, by emphasizing the elements that are important to this research.

The organization of this Chapter is as follows. Section 2.1 gives an overview of the software defect terminology. Section 2.2 introduces a typical software testing process. Section 2.3 presents the commonly used testing techniques. Section 2.4 focuses on testing levels, namely unit, integration, and system, and Section 2.5 provides an overview of regression testing. Fault models are introduced in Section 2.6, and Section 2.7 concludes the Chapter.

2.1 Fault-Error-Failure chain

The terms *fault, error*, and *failure* should not be used interchangeably, as they actually have different meanings. The understanding of both their definitions and the relationship among them is a key feature for the purpose of this thesis. This section summarizes the fundamental concepts of faults, errors, and failures, according to a commonly used taxonomy (Avizienis *et al.*, 2004).

- **Fault.** A fault is a structural imperfection, as judged against the requirements. A *fault* is a defect within the system. It may be an incorrect program instruction, an incorrect requirement, or other incorrect artifact. Faults are often referred to as *software bugs*.
- **Error.** An error is a deviation from expectation based on encountering a fault during execution. An *error* is the result of executing the portion of the program that contains a fault. A fault may lead to an error. The error may be an incorrect result due to a faulty algorithm, it may be a value that arrives too late in a time sensitive design, or it may be that no value is produced and a variable has a bad value as a result. It is important to highlight that a fault may stay *inactive* for a long time before it manifests itself as an error. For example, a software bug in a subroutine is not visible while the subroutine is not called.
- **Failure.** A failure is the inability of a system or component to perform its required function within the specified requirements. When an error occurs and propagates to the point that it causes a deviation from the expected to either a human user or another system, a *failure* has occurred. For example, the user interface shows a computed value. If the value is different from the expected, the program has failed.

2.2 Software testing process

A typical software testing process encompasses four phases: *(i) Planning, (ii) Specification, (iii) Execution* and *(iv) Reporting* (Myers *et al.*, 2011). The purpose of the *Planning* phase is to plan all test activities to ensure project success, as a means to properly manage testing from this phase until project termination. This phase is ruled out by an artifact called *test plan*, later discussed in this Section. The *Specification* phase involves the use of test strategies to develop (write, generate, fix, rewrite) the test cases. Next, the *Execution* and *Reporting* phases are responsible for executing the Software Under Test (SUT) with the test cases, by employing input data under a particular set of conditions, and to verify whether the outputs are correct for all test inputs, reporting the observed behavior of the SUT, respectively. Knowing the proper outputs for a given set of inputs and execution conditions is a mandatory aspect of the software testing discipline (Burnstein, 2003).

A software testing process is associated to the test artifacts defined in the Standard for Software Test Documentation (IEEE, 1998). These are described next:

- **Test plan.** This artifact is the main output of the *Planning* phase. A test plan details all tasks to perform, which includes the test strategies to employ (approach) - including both the techniques and the coverage criteria to reach, the items to be tested (scope) - modules, features, and the project estimates and resources - risks and dependencies, schedule, personnel, etc. The test plan should be updated throughout the project until the completion of closure activities. A well though-out test plan will provide the test team with a clear direction and understanding of the procedures taken to achieve a certain result.

- **Test specification.** The specification defines *how* testing will be performed. The artifacts surrounding test specification are managed at the *Design* phase of a software testing process. These can be defined as follows:

  – **Test design specification.** It refines the test approach and identifies the features to be covered by the design and its associated tests. Besides, it identifies the test cases and test procedures, if any, required to accomplish the testing and specifies the pass/fail criteria.

  – **Test case specification.** It documents the actual values used for input along with the anticipated outputs. A test case also identifies constraints on the test procedures resulting from the use of that particular test case. Test cases are separated from test designs to allow for use in more than one design, and to allow for reuse in other situations. A collection of test cases that are grouped for test execution purposes is often referred to as a *test suite*.

  – **Test procedure specification.** This artifact identifies all steps required to operate the system and exercise the specified test cases, in order to implement the associated test design. Test procedures are separated from test design specifications as they are intended to be followed step by step and should not have extraneous details.

- **Test execution.** It consists of running the test cases, either manually or automatically. This phase is usually bundled with the test reporting. It covers the following artifacts:

  - **Test log.** This artifact is responsible for recording what occurred during test execution, i.e., the result of a test case execution - pass/fail.
  - **Test incident report.** It describes any event that occurs during the test execution which requires further investigation, i.e., events that cannot be handled at a particular execution.

- **Test reporting.** It comprises the artifacts from the last phase of a test process, as follows:
  - **Test item transmittal report.** It identifies the test items being transmitted for testing in the event that separate development and test groups are involved or in the event that a formal beginning of test execution is desired.
  - **Test summary report.** It summarizes the testing activities associated with one or more test design specifications.

While the description above claims each testing phase to precede another one, by following a sequence, from *Planning* to *Reporting*, it is worth mentioning that such a *waterfall* structure is not encouraged. Instead, the phases should communicate with each other, providing continuous feedback.

Figure 2.1 shows an activity diagram that illustrates a general software testing process workflow, with its main tasks, and the communication among them. By following such a flow, after being planned, the tests are designed, and run afterwards. The design is represented by a fork node, with two possibilities, i.e., manual and automated test cases. A manual test case is a sequence of test steps written in natural language, whereas the automated scripts are mechanically interpretable representations of manual test cases (Thummalapenta *et al.*, 2012). When executing a test and finding an issue, a report should be created to capture what is already known about the issue. Next, if the established pass/fail criteria is not achieved, it is necessary to write new test cases, or even make some adjustments in the test plan, and the flow repeats from the beginning, until the condition of a test success is achieved.

The above description specifies the common information to consider in a software testing process. Despite the importance that all testing artifacts and phases hold, more attention should be given to the **test case** artifact. It is far the most important one in the testing process, and main focus of research in the software testing field, independently on the major. Its definition is the core task of a testing strategy.

Many authors claims software testing to be a costly activity in software development, taking over fifty percent of development budget (Harrold, 2000). Vegas and Basili (2005) argue that one of the factors that influence such a cost is the number of test cases used. The problem is

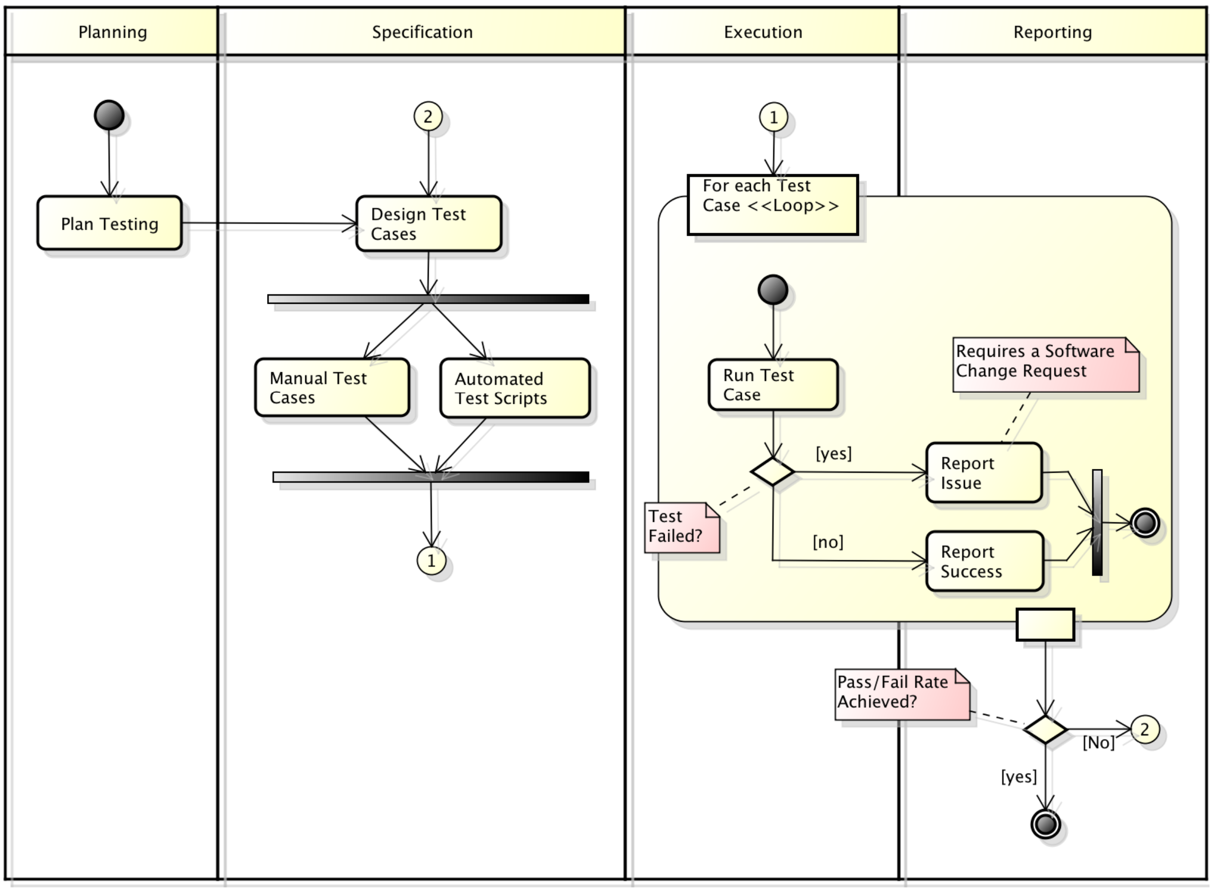


Figure 2.1: A general software testing process workflow.

that, when a large number of test cases is created, a long time is required to specify, execute, and analyze them, what demands a high amount of resources. It would seem to make sense to limit the test scope to the most important test cases in the test suite. Therefore, the major concern in software testing is to effectively use the resources available by developing a set of test cases that yields the maximum amount of defects for the time and effort spent (Burnstein, 2003).

2.3 Test specification

Test cases are designed to cause faulty programs to make errors which result in program failure (Ammann and Offutt, 2008). A test case is defined as a success if it can detect faults in a given amount of time, and as a fail if it does not detect any fault. The capability a test case holds in identifying a defect shows how effective it is.

Test case specification is concerned about designing effective test cases (Myers *et al.*, 2011).

In general, fixing a bug is easy, but finding the bug could be a nightmare. A developer may spend hours and days in front of the debugger, trying to identify the faults from which the errors emerged. Besides, fixing one bug could break other parts of the code, and also the bug, once fixed could appear again later.

In addition, effective test cases should also consider their capability to uncover the most important input data values, i.e., those that are likely to hold defects, in an efficient fashion, with a minimum amount of time and effort.

Therefore, adequate testing techniques should be employed towards designing effective test cases. Along this section we discuss the commonly applied techniques to achieving this goal.

2.3.1 Testing techniques

Software testing techniques provide different criteria to designing the set of test cases. These criteria allows grouping testing techniques in families according to the source of information used to define the test requirements. These can count on *specification* (functional) or *source code* (structural) information (Juristo *et al.*, 2004). We next describe the role of each group of techniques.

- **Functional testing.** This family of techniques is used by considering the system as a *black-box*, i.e., the test case generation counts on inputs and associated outputs, as defined in the requirement or design specifications. It focuses on the **external behavior** of the software entity under test, and the internal structure is not taken into account. The most common techniques in this family are *equivalence class partitioning, boundary value analysis*, and *cause-effect graphing*.

  The main idea behind *equivalence class partitioning* and *boundary value analysis* is to divide the system inputs into subsets, termed equivalence classes, where each class element behaves similarly. On the other hand, *cause-effect graphing* explores the combinations of input circumstances (Myers *et al.*, 2011), not covered by the preceding techniques. That is, in a cause-effect graph, *causes* are the input conditions and *effects* are the results of those input conditions. Causes and effects are identified by reading the specification word by word and underlining words or phrases that describe causes and effects. By methodically tracing state conditions in the graph, the graph can be converted into a limited-entry decision table. Each column in the table represents a test case.

  Given that testing all possible inputs is not feasible in practice, functional testing techniques apply some strategies aimed at reducing the most part of the software, while keeping relevant input data (Ammann and Offutt, 2008).

- **Structural testing.** As opposed to the prior, structural testing requires knowledge of source code. It is often referred to as *white-box* technique, by emphasizing the **internal structure** of the software. The goal of selecting such a family is to cause the execution of specific spots in the software entity, such as specific statements, program branches or paths, determining if all the logical and data elements in the software unit are functioning properly. The expected results are evaluated on a set of structure-related coverage criteria, such as *control-flow*, and *data-flow* coverage (Ammann and Offutt, 2008).

  Control-flow criteria works by examining the branch and loop structure of a program. The logic elements to consider for coverage based on the control flow in a source code unit are the following: program statement, branches (decisions - if/then/else and for/while statements; conditions - true/false expressions), and combinations of decisions and conditions (Burnstein, 2003). In program statement coverage, every statement must be executed at least once, and in branch coverage, every branch must be traversed. The majority of systematic white-box testing approaches consider the control-flow of the program and try to cover as many aspects as possible (Fraser and Zeller, 2012).

  Data-flow criteria focuses on how variables are bound to values, and how these variables are to be used (Rapps and Weyuker, 1985). As opposed to selecting program paths based solely on the control structure of a program, the data flow track input variables through a program, following them as they are modified, until they are ultimately used to produce output values.

A meaningful measure of testing technique is its fault-detection ability, i.e., its *effectiveness*. A good means to accomplish that is for a software testing strategy does not only consider either functional or structural testing, but instead it might feature elements of both, to ensure that a rigorous program testing was designed. In practice, testing techniques are much more effective used in combination than used separately (Wood *et al.*, 1997).

A test case can be generated for both manual or automatic execution. The first is more intended to functional testing, while the structural testing should be automated. However, even functional testing should be as automated as possible, so as to enable rapid feedback, and cost and effort reductions. Research on automated test case generation has resulted in a great number of approaches for test case derivation, either from models or source code, and using different test objectives such as coverage criteria, and using different techniques and algorithms (Fraser and Zeller, 2012).

Some authors also include another testing technique, that also holds a great importance to achieve the test case goals, the so-called **error-based testing**, introduced by Morell (1990), and further explored in the software testing field (Delamaro *et al.*, 2007). In this technique, the test criteria and requirements come from the kwnoledge about the common and recurring errors made in the software development process. Therefore, it seeks to demonstrate that prescribed faults are not in the program.

The technique has some components. One is the identification of general classes of errors that occur during program construction. Another is the assumption that the expert tester knows how to interpret general error classes in particular contexts, and choose relevant tests and test methods (Howden, 2011).

Error Seeding (Meek and Siu, 1989) and *Mutation Analysis* (DeMillo *et al.*, 1978; Mathur, 2002; Jia and Harman, 2011), both error-oriented techniques, are typical testing criteria applicable to this testing technique. They are usually applied to assess the quality of the test cases, i.e., whether they can reveal certain types of faults.

In error seeding, a predefined number of artificially generated errors is inserted into the system. After that, test runs are used to detect errors and to examine the ratio between actual and artificially "sown" errors, based on the total number of detected errors. The testers do not know the artificially generated errors.

Mutation testing is a kind of systematic method of error seeding. The criteria considers all simple faults that could occur. Hence, certain statements in the source code are "mutated", i.e., purposely modified, to introduce single faults, one at a time, to create "mutants" of the original program. The test set is then applied to each mutant program, to check if the test code is able to find the errors. The test adequacy is measured by the amount, or percentage, of "mutants killed".

In practice, mutation testing would be extremely difficult without a reliable, fast and automated tool that generates mutants, runs the mutants against a test suite and reports the mutation score of the test suite (Madeyski, 2010). Judy[1] is a well-accepted mutation test tool for Java.

2.3.2 Test case prioritization

Given the usually limited resources in software projects, it is rather important to establish priorities to test case execution, in order to increase the likelihood of revealing defects in the program under test (Haidry and Miller, 2013).

Test case prioritization is the process of establishing test case priority order, with respect to a given goal, in such a way that test cases with high utility are given higher priority, as they are

[1]`http://mutationtest.com/`

expected to present better fault detection results (Gonzalez-Sanchez *et al.*, 2010). It means that if known which test cases are more likely to detect faults, these should be run earlier than others.

Catal and Mishra (2012) analyzed existing literature on the topic, covering about 120 research papers. The study leveraged the most important aspects of test case prioritization, and pointed out which techniques have caught researchers' attention in the last years. The investigation elaborates on the work of Yoo and Harman (2012), that discussed about existing techniques for *test case prioritization*. They classified the existing techniques into the following categories: *coverage-based prioritization, distribution-based approach, human-based approach, probabilistic approach, history-based approach, requirement-based approach, model-based approach*, and *cost-aware approach*. Among these, there is a higher prevalence of coverage-based prioritization techniques.

Another classification (Haidry and Miller, 2013) reduces to three the number of categories of techniques, as follows: *History-based:* prioritizes by using information from previous execution cycles, like data about code-coverage, i.e., the higher the achieved coverage by a test set, the higher the priority; *Knowledge-based:* uses the human knowledge power to perform the prioritizations; and *Model-based:* uses models of the system to determine the priorities. It considers the models to establish "weights" of importance, for a given entity, thus enabling prioritization based on the entity's importance, e.g., the number of dependencies an entity has.

2.4 Test levels

Software testing activities are commonly categorized into different levels (Ammann and Offutt, 2008). The definition of different levels has been used to characterize an activity that occurs at a specific timepoint in a software development process. Each level has a particular interest, such as analyzing the incorrect use of a given algorithm, or to expose deviations from the requirements, or even evaluating robustness to stressful load conditions. These are distinct *fault* types that can be found in different levels of abstraction, e.g., the prior could be found when analyzing the implementation routine, the second should consider if the system functionalities were developed accordingly, and the latter counts on the whole application usage to analyze whether system performance is in compliance with the specified non-functional requirements. The levels are described next.

- **Unit Testing.** It tests the smallest part of a program. It usually consists of testing methods in a component (in Object-Oriented Programming (OOP)), both to ensure the *unit* is working properly, and to find internal logic implementation errors. As a piece of code is implemented, a test should be designed that test the new functionality. It is also possible to consider a whole component as a unit. The level of granularity will depend on the test strategy employed. Automated unit tests are very popular, probably because in the majority of the projects, developers are responsible for testing (Greiler *et al.*, 2012). Developers create unit test cases in the form of small programs, typically in a framework such as the JUnit [2], or the TestNG framework[3]. A great advantage of unit testing is that a suite of tests can be run as often as desired without any manual intervention (Thummalapenta *et al.*, 2012).

- **Integration Testing.** It is carried out when two or more tested units are ready. It tests the integration of units to prove they also work properly when put together. Integration tests can be performed by accessing the source code, and also using their external interfaces. Alike unit testing, a framework such as the JUnit can be used to automate the integration tests. Indeed, there are specific tools for this level, however their usefulness is dependent on the type of system components to integrate.

- **System Testing.** After finishing the components that will compose a product, it is necessary to test the behavior of the entire system, i.e, the interactions of all components, to verify whether the actual system meets the specifications. Besides, at this level, quality attributes (i.e., non-functional requirements, e.g., availability, performance, usability, security, reliability, etc.) may be tested, given that the whole system is expected to be functional. Unlike the preceding levels, system testing does not require access to the source code. Furthermore, system tests are typically written by testers, not the developers of the application.

Besides the three standard test levels, a fourth test level can also be considered, the **acceptance testing**. This is the only level in which the testing goal is more to prove the system is working rather than finding errors (Burnstein, 2003). Acceptance testing consists of an *enhanced* system testing, usually performed with the presence of end-users or customers, to ensure that the system is performing what they initially defined in the requirements specification document, and that the system does not operate outside of its specifications as well (Lewis, 2008). Some projects do not formalize acceptance tests, especially when end-users have been involved continuously throughout the development cycle and have been implicitly applying acceptance testing as the system is developed (Lewis, 2008), or due to time constraints in the project.

[2]JUnit is a unit testing framework for the Java programming language. Available at `http://junit.org/`
[3]Another unit testing framework for Java. Available at `http://testng.org/`

There is some attempt to accurately reflect the software development lifecycle phases with the test levels, such as in the V-model for system development, initially presented by Boehm (1979), and largely discussed in the literature afterwards. However, in practice, implementing unit tests and integration tests as single, separated phases results in a thoughtless approach for testing. For example, a single unit test will require a custom test harness. Each unit may require a different test harness. For a large project containing several units, this could prove to be costly and problematic. A better idea could be testing a unit when connected to the actual system, using the system to deliver test messages (Marick, 1999).

Therefore, it is way more interesting to look at the test levels from another perspective, other than only trying to establish a path in which each level must only be adopted during a particular development phase. As the test levels enable to understand which kind of faults they are more likely to uncover, it could be helpful for a fault management process, which should determine where faults are found, and also where they are introduced.

2.5 Regression testing

Whenever software changes, it is necessary to analyze whether that modifications are correct, and that already tested functionalities, that previously worked properly, have not been negatively affected. **Regression testing** is the testing activity aimed at providing confidence that the newly introduced changes do not harm the existing behavior of the software (Rothermel and Harrold, 1997). It constitutes the vast majority of testing effort in commercial software development, being regarded as an essential part of the maintenance activity, especially when multiple software releases are developed (Ammann and Offutt, 2008). As such, it demands special attention in the software testing process.

The most straightforward regression testing strategy is to *rerun* all existing test cases, in a strategy called *retest-all* approach (Yoo and Harman, 2012). However, such a strategy cannot be considered as feasible, due to its high cost. Including every existing test sets entails a likely unmanageably large regression test set (Ammann and Offutt, 2008). This limitation forces consideration of *selective* strategies that aim at reducing the required effort.

Rothermel and Harrold (1996), the most influential authors in the regression testing field, introduced two decades ago, a detailed study on existing techniques for selecting regression tests[4]. In the investigation, the authors analyzed thirteen techniques, which spans a variety

[4]Although this study (Rothermel and Harrold, 1996) is not a recent source, it may still be considered as a well-accepted source of information regarding regression test selection techniques. As early as in March 2012, the google scholar indicates that about 450 studies have cited it so far. Two years later, in March 2014, the number

of topics, such as: *linear equation, symbolic execution, path analysis, dataflow, program dependence graph, system dependence graph*, among others.

The authors proposed a framework to categorize the regression techniques according to a set of criteria (inclusiveness, precision, efficiency, and generality), thus enabling comparisons of their fault-detection and cost reduction abilities, conditioned on criteria. They argued about a required trade-off analysis between the costs of selecting and executing test cases, when choosing a technique for practical application.

Another investigation held by the same group of authors and their colleagues (Graves *et al.*, 2001) reported on an empirical evaluation of the cost and benefit analysis of five techniques for reusing test cases, focusing on their relative abilities to reduce regression testing effort and uncover faults in changed programs.

Engström *et al.* (2010) carried out a thorough systematic literature review on the topic. They analyzed thirty-two techniques, in order to compare available evidence on regression **test selection**. Similarly to the aforementioned investigations, the statement about the trade-off analysis continues to hold true in this study, given that authors claim about the lack of common definition of what criteria defines a good regression test selection technique.

2.6 Fault models

Software engineers, when designing test sets, have in mind what problems are more likely to occur, given they usually have accumulated those in memory, from years of experience. While it is possible to design test sets without fault modeling support, it is not true that a common testing strategy does not involve fault models, at least implicitly. That is, an experienced developer often knows where errors are more likely to occur, and as such may put more effort into exercising the “hot spots" than others.

An important steps towards the improvement of the testing process is the ability of estimating software fault-proneness, i.e., the probability of the presence of faults in the software (Fenton and Neil, 1999). The idea is to estimate to what extent a software module is expected to be faulty both *before* and *after* testing. A realistic estimation might aid testing as understanding the nature of software fault-proneness allows to beter focus the testing activities, thus improving the allocation of resources. Conversely, after testing, the estimation of software fault-proneness can provide feedback on testing and help in defining maintenance actions.

increased to 518, what demonstrates an increasing trend. Rothermel and Harrold remain the most influential authors in the field. They have authored or co-authored the most cited publications in the field.

A significant research effort has been dedicated to defining specific measures and building quality models based on those measures (Arisholm *et al.*, 2007). Several software metrics have been proposed to characterize software both statically and dynamically, and many evidence prove the existence of correlation between values and fault-proneness, as well as many other non-directly measurable software attributes have been empirically proved by many authors (Basili and Hutchens, 1983; Gill and Kemerer, 1991; Frankl and Iakounenko, 1998).

In theory, if relations are found between the software attributes and fault-proneness, better and more accurate predictive models can be defined. On the other hand, the attempts to define models for computing fault-proneness indicators based on software metrics have not succedded in producing convincing general models (Denaro *et al.*, 2002). There exists little evidence of the economic viability of such models (Briand and Wüst, 2002).

The fault prediction models support the identification of faulty modules in a software system. These are modules candidate for verification and testing, or at least where priority should be given. Building such predictive models demands investigating sets of historical observations so as to understand the nature of faults in the software systems, rather than only counting on structural measures.

To build such models there exists a large number of modeling techniques to choose from, including standard statistical techniques such as logistic regression (Denaro *et al.*, 2002), or *optimized set reduction* (Briand *et al.*, 1992), data mining techniques such as *decision trees* (Porter and Selby, 1990; Selby and Porter, 1988), machine learning techniques such as *support vector machines* (Elish and Elish, 2008), and *neural networks* (Khoshgoftaar *et al.*, 2006).

It is rather important to build models that reflect the occurrence of faults in a given software system, so that testing activities can be planned effectively. Associated with the understanding of the nature of faults, another line of action in a testing process may emerge. It consists of fault modeling. It is tied to the idea of building fault dictionaries to augment the efficacy of testing process and to improve the overall quality of the delivered software.

In terms of what actually constitutes a fault model, the literature provides a plethora of definitions for the term, as Reinecke *et al.* (2010) identified in their literature review on the topic. Among the commonly accepted definitions, Burnstein (2003) claims a *fault model* as *"a link between the **error** made, and the **defect** in the software"*. In order for such a simplistic definition to be intelligible, recall that defects/faults arise because errors are made, in a sort of cause-and-effect relationship. Even under the best of development circumstances errors are made, resulting in faults being injected in the software during the phases of the Software Development Life Cycle (SDLC). Recall also that a failure is an observed deviation from the expected behavior (specification).

By following the basic concepts of fault-tolerant computing (Cristian, 1991) and the demonstration from (Reinecke *et al.*, 2010), let us employ the notion of software components (provider) that implement a set of functionality to be used by other software components (consumer). This consumer in turn can be thought of as a provider of functionality to other consumers, and so on. The functionality provided by a component is correct if it complies with the specification for the software component. By hypothesizing that a functionality-failure in a component may be the cause of a failure of the functionality the client in turn provides to its own consumers. Consequently, the failure in the provider is a fault for the consumer. As we may observe, different failures in the provider constitute different faults to the consumer and may result in different consumer failures.

This scenario motivates the importance of knowing the sources of faults and understanding how they could affect the behavior of a system, not just the faults themselves, as an opportunity to eliminate them at their source. According to many sources, the earlier within the SDLC a problem is discovered, the cheaper, and often easier, it is to fix (Basili and Perricone, 1984; Ammann and Offutt, 2008; Myers *et al.*, 2011).

We could rephrase the definition of a fault model as *a description of the behavior of, and assumptions about, how components in a faulty system behave*. A fault model provides testers with specific fault types for which to search based on the types of technologies used and the activities that have preceded the tests. Tests or review scenarios are written to search for each possible type of fault.

A fault model describes the space of erroneous behaviors which can be expected as a result of a error. Such a description consists of a fault list or dictionary. From the fault list/dictionary, faults can be selected, and test inputs can be developed (McGregor, 2008). It is an effective means to design test cases that have a high probability of revealing faults (Martin and Xie, 2007).

The tester with access to the fault model and the frequency of occurrence of fault types could use this information as the basis for generating fault hypotheses and test cases, aiming at building a minimal complete test suites for a fault model.

Determining the frequency occurrence should rely on the extensive analysis of historical data, encompassing a collection of experience, in terms of common faults, about the scenario under analysis in a range of criteria (e.g., application domain, programming language, structural complexity of the software, etc).

Furthermore, the role of a fault model relies on specifying the fault types to be detected by a test. This is strongly tied to the SDLC phases, as there are different fault categories for each work product (e.g., requirements, design, code, and test artifacts) (Chillarege *et al.*, 1992; Seaman *et al.*, 2008; Strecker and Memon, 2012).

In practice, there is no a generic fault model that can be used by every scenario. Fault models are usually dependent on particular domains and types of applications. When understanding the issues of a particular domain, the interest is in the behaviour of the system under various common faults. While the fault model support cannot guarantee the absence of a specific type of fault, it can be used in assurance arguments that specific procedures have been used to search for specific faults.

A measurement of the efficiency of a fault model can be defined as the percentage of the existing actual faults covered by testing the modeled faults (Bengtsson and Kumar, 2005; Jansen, 2010). It is worth to mention that, since more thorough fault models need higher test effort, because more possible faults have to be considered, a trade-off between quality and cost of a test must be found in practice.

Although the relationships between software failures, software faults, and their origins are not easily mapped, fault models concept and fault lists are useful to design tests and for diagnosis tasks during fault localization activities. Such a scenario might pose a challenge for defining fault models for software engineering.

2.6.1 Fault models in systems and software engineering

A fault model is an engineering model of something that could go wrong in the construction or operation of a piece of equipment, structure, or software (Martin and Xie, 2007).

Fault models have been largely used in testing of physical systems, such as integrated circuits, microcontrollers, and Operating Systems (OS) device drivers. In such application domains, testing today typically consists of generating test patterns based on multiple fault models that emulate manufacturing defects. A seminal study dates back to 1982, where Malaiya and Su (1982) proposed a fault model to improve fault coverage in testing CMOS[5] devices. Since then, the use of fault models has been growing increasingly, that several models have become sort of standards to follow when test cases are generated (Botaschanjan and Hummel, 2009). It is the case of the “classic" fault models *stuck-at faults, multiple stuck-at faults, bridging faults, interconnect open faults*, and *delay faults*, that have long been used for fault diagnosis in the development of electronic circuits (Pomeranz and Reddy, 2009; Jansen, 2010).

In the domain of OS device drivers, the research community has also defined a detailed representation of commonly occurring faults, and modeled these as fault models, such as *bit flipss, data type dependent parameter corruptions,* and *parameter fuzzing*. These are mandatory assumptions when test cases are generated that aim at revealing robustness vulnerabilities of an

[5]CMOS stands for *Complementary metal oxide semiconductor*, a technology for constructing integrated circuits.

OS device driver (Winter *et al.*, 2011).

From the software engineering perspective, fault modeling has also served to improve software testing efficiency and effectiveness for the detection and removal of faults, of which a not exhaustive list is provided below:

Offutt *et al.* (2001) presented a model for the appearance and realization of OOP faults and defined specific categories of inheritance and polymorphic faults. The investigation was aimed at providing support to empirical evaluations of OOP testing techniques, and to improve design and development of OOP software. They observed that, rather than only taking corrective measures to eliminate the faults, it is necessary to understand the problems that can be caused by method overriding and polymorphism and document them as a fault model.

Martin and Xie (2007) employed a fault model approach to modeling things that could go wrong when constructing an *access control policy*. They used a fault model to measure the fault-detection effectiveness of automatic test generation and selection techniques. To this end, it was necessary to create a broad categorization of faults in the domain under evaluation.

Babu and Krishnan (2009) applied the fault modeling concept to support the identification of faults during aspect composition in the early design stage. They provided a list of faults that can occur during aspect composition at a shared join point. Such a support would help in adopting better coding strategies that result in modular, reusable and maintainable code.

Reinecke *et al.* (2010) surveyed the use of fault-models for *Quality of Service* studies of Service-Oriented Systems. They provided an overview of the fault-models available in the literature, using a fault-classification scheme based on architectural properties of a Service-Oriented Architecture (SOA)-based system, and review schemes that have been proposed for classifying faults in SOA-based systems. Such a study is another clear indication that fault models subsume a range of benefits for early fault detection in software systems development.

2.6.2 Fault models in the software development life cycle

Fault modeling may be an important element for several activities in the software development life cycle, ranging from requirements to maintenance. These activities deal with different inputs, at different abstraction levels. A fault model provides testers with specific fault types for which to search based on the types of technologies used and the activities that have preceded the tests.

Different fault models helps detecting defects that were introduced during different phases, so that they could be fixed in the earlier stages of the projects avoiding rework (Kumaresh and Baskaran, 2012). Thus, a fault management process could be defined as a means to determine where faults are found and where they are introduced.

A template for a development phase description can be defined to support the process. Figure 2.2 shows a template we defined. It comprises four elements of interest for the development phase, as explained next:

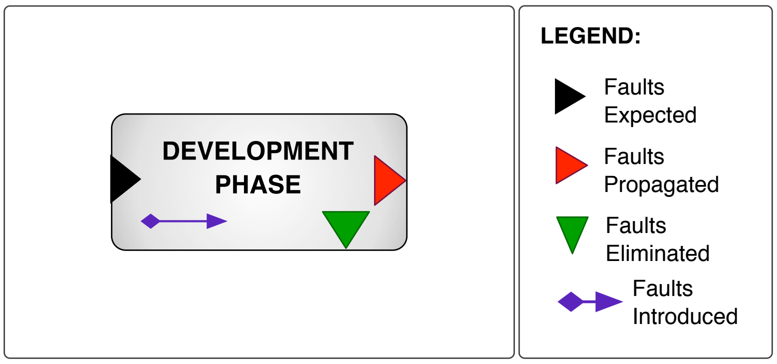


Figure 2.2: Process template.

- **Faults expected** - When defining a process each phase will propagate some faults on to the next phase. The activities in the second phase should be created to identify and handle these faults.
- **Faults eliminated** - The activities in each development phase may eliminate faults that have been previously introduced.
- **Faults introduced** - Each phase of development has the potential to inject faults into the product or its supporting artifacts. The nature of the phase determines what is possible.
- **Faults propagated** - Faults that are either anticipated or introduced must either be eliminated or they will be passed on to the following phase.

Artifacts created at one phase are passed on to, and used by, later phases. As a consequence, any fault injected and not detected by the verification activities in that particular phase is still in the artifact when it is used by the later phase. This action is referred to as *fault propagation*.

When a fault is injected into a development artifact, such as architecture or program code, that fault remains until it is recognized and removed. As the development proceeds and the artifacts are used by later phases, a fault may cause errors that result in additional faults being created in other artifacts. For example, a faulty requirement may result in a fault in the architecture and several faulty test cases thereupon.

The Verification and Validation (V&V) activities of a development process are intended to identify those faults. These activities should be planned with specific faults in mind. Developing

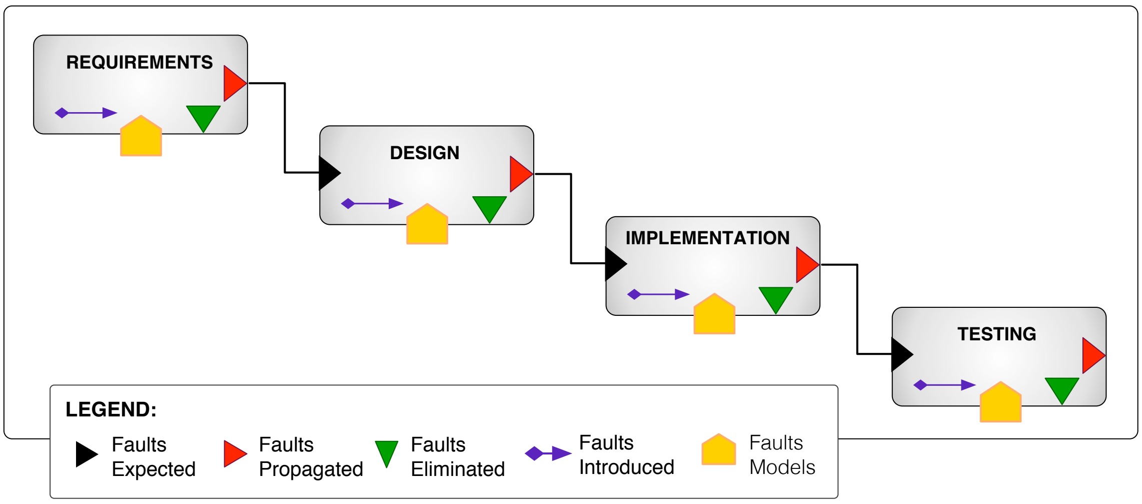


Figure 2.3: Software process lifecycle with fault model support.

the V&V plan for a project is the point at which specific fault types related to the technologies being used are mapped to the development process. This is the point at which fault modeling becomes particularly interesting. At each phase, the V&V activities will be defined in conjunction with the fault models. A process definition will be a composition of phase definitions, as Figure 2.3 shows. The V&V activities in each phase are the first line of defense. Expected fault types and introduced fault types are searched for.

Figure 2.4 illustrates a typical requirements phase definition. This phase comprises requirements elicitation and writing. Considering this as the initial phase in the development life cycle, there are no input faults. However, a number of faults can be introduced. Seaman *et al.* (2008) listed a number of defect types inherent to requirements definition: *clarity, completeness, compliance, consistency, correctness,* and *testability*. Thus, the fault models should take into consideration the *common mistakes in requirements statement definition*.

We should be aware that, even applying an effective set of fault models, it is likely that some faults might still be propagated. The role of fault models will be to reduce the probability of such a propagation.

2.7 Chapter summary

Software testing can be regarded as the most important quality assurance technique. Employing a testing strategy is crucial to any software project success. In this background chapter, we reviewed some of the basic concepts of software testing.

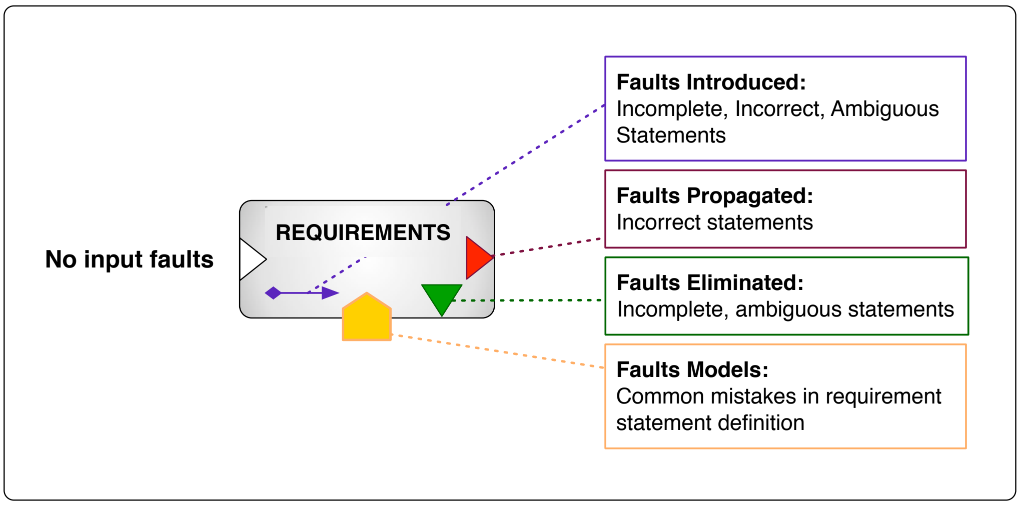

No input faults
REQUIREMENTS
Faults Introduced:
Incomplete, Incorrect, Ambiguous Statements
Faults Propagated:
Incorrect statements
Faults Eliminated:
Incomplete, ambiguous statements
Faults Models:
Common mistakes in requirement statement definition

Figure 2.4: Fault model support within the requirement phase.

Besides, we introduced the concepts of fault modeling. As a commonly applied strategy in testing of physical systems, we discussed the inherent aspects that make fault models also relevant to improving the software testing practice.

Next Chapter presents a general introduction of software product lines, presenting their principles, and discussing its benefits and drawbacks.

3 Software Product Line Engineering

Traditional Software Engineering is usually focused on building individual software systems, one system at a time. It includes some sort of specification gathering, proceeding with the design, implementation, and testing. By the end of the construction process, a software system is deployed. A range of practices have been well established that support this software development strategy (Sommerville, 2011).

Such a typical software development strategy might not cope with the increasing market demands for variability and customizability. That is, similar products that share functionalities and might accommodate some adaptations, to meet the needs of particular customers. As each product is treated as an individual unit, the designed assets are not variable enough to be reusable in different products. Therefore, in order to meet customer needs, several products have to be built from scratch. From both engineering and economic perspectives, it does not make sense to develop each product separately, as it would demand setting up different development teams to cope with the creation of different products. In effect, the support for **variability** and **mass customization** might not be cost effective. Both these aspects have become important attributes of modern software development practices (Rashid *et al.*, 2011), and due to become a must for market success (Benavides *et al.*, 2010).

Svahnberg *et al.* (2005) define **variability** as *the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context.* From a product line development perspective, such a definition could be rephrased as *the ability to derive different products from a common set of artifacts* (Apel *et al.*, 2013). Variability of a software component is related to its reuse as increased variability increases the likelihood of reuse (Fazal-Amin *et al.*, 2011).

Further, the concept of **mass customization** is tied to the definition of *mass production*. As Benavides *et al.* (2010) state, the latter stands for *"the production of a large amount of* *standardized products using standardized processes that produce a large volume of the same product in a reduced time to market"*. Therefore, **mass customization** could be defined as *the production of goods and services to meet individual customers' needs with near mass production efficiency* (Tseng and Jiao, 2001). This is widely applied in the automotive industry, which focuses on efficiently producing and maintaining multiple similar products, exploiting what they have in common and managing what varies among them (Krueger, 2001). This strategy is an important player in market competition (Alford *et al.*, 2000).

As opposed to the creation of "customized" individual software from scratch, SPL engineering provides efficient means to develop variable software, and manage variability across a very large number of similar products. Clements and Northrop (2001) defines an SPL as *"a set of software-intensive systems sharing a common, managed set of features that satisfy 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"*. It is based on the idea that assets, built on the basis of a common design, can be systematically configured and composed in different ways, so as to enable the creation of a diversity of products, in a shortened building period, instead of either developing each product from scratch, or copying the software from a similar system and changing it when necessary, in an *ad-hoc* fashion.

SPL engineering provides a series of benefits over any *ad-hoc* reuse strategy (Clements and McGregor, 2012), such as savings in development and maintenance costs, reductions in time to market, and improvements in the quality of the product delivered to customers (Pohl *et al.*, 2005). SPL engineering leads companies to achieve remarkable economies of scale and scope, leading to large productivity gains. Examples of SPL are manifold and can be found in different application domains (Rashid *et al.*, 2011). There are lots of companies that report successful stories about the application of SPL engineering in their development processes, as listed in (Pohl *et al.*, 2005, chap. 21) (Weiss *et al.*, 2006) [1] (van der Linden *et al.*, 2007, chap. 8-17).

In order to achieve the potential benefits of SPL engineering, it is necessary to understand and control common and distinguishing characteristics between the systems that are part of the product line. Given that SPL engineering demands software assets with high variability, there is a minimally necessary set of steps to accomplish (Svahnberg *et al.*, 2005), namely identifying, constraining, implementing, and managing variability. These steps are subject to discussion in the following sections, as follows: Section 3.1 introduces the particular aspect of

[1]The Carnegie Mellon Software Engineering Institute (SEI) maintains the *Software Product Line Hall of Fame* web site - c.f. http://splc.net/fame.html. It contains a list of sucessful stories about the application of SPL engineering in software-intensive systems development. These stories serve as models of what an SPL should be, according to the view of a board committee, formed by active players in the SPL (research & development) field.

SPL engineering, namely, the division of development into two processes. Section 3.2 elaborates on variability, the key element of the SPL development. The commonly adopted SPL strategies are discussed in Section 3.3. Finally, Section 3.4 summarizes the Chapter.

3.1 Domain and application engineering processes

Unlike traditional Software Engineering (henceforth named as single systems development), SPL engineering splits the software development into two distinct processes: ***domain engineering*** and ***application engineering***. Figure 3.1 shows a simplified version of a commonly accepted SPL engineering framework, encompassing the both processes and their associated lifecycle disciplines. The framework was originally introduced by Pohl *et al.* (2005), but a range of modified versions can be found in the literature, which consider another set of disciplines in the SDLC. It is the case of an extension proposed by our Research Group (Neiva *et al.*, 2010; Machado *et al.*, 2011; Cavalcanti *et al.*, 2011a), which includes aspects of Scoping (Moraes *et al.*, 2011), Evolution Management (Oliveira, 2009) and Risk Management (Lobato *et al.*, 2012), orthogonal disciplines that should be considered in both processes.

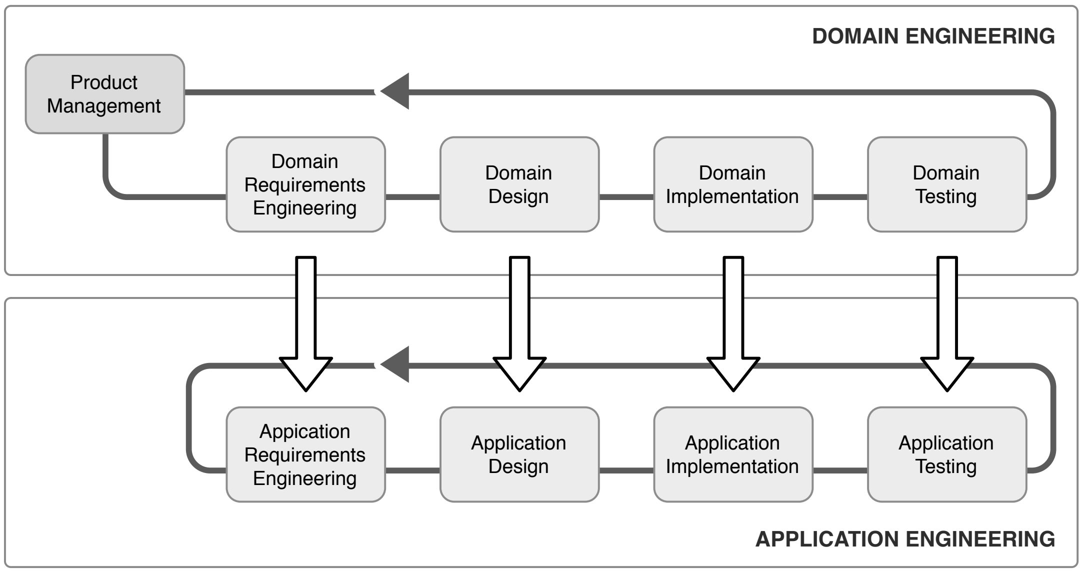


Figure 3.1: SPL Engineering Framework.

Domain engineering (development for reuse) is in essence, the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, on top of which all products are built (Czarnecki and Eisenecker, 2000). It includes the implementation of a set of common assets, and base technologies which allow the derivation of products. The derivation of products describes the process of their creation, carried out during *application engineering* (Rashid *et al.*, 2011). In ***application engineering (development with reuse)***, products are then assembled, by reusing the commonalities, binding the variability defined in the domain artifacts, according to customer- and/or market-specific needs, and implementing product-specific parts (Pohl *et al.*, 2005).

SPL engineering is strongly dependent on a common product line architecture, that is also called reference architecture (Pohl *et al.*, 2005). It must be designed as early as in domain engineering. It specifies the common structure of the products and centers in the development and evolution of both domain and application assets (Wu *et al.*, 2011).

3.2 Handling variability

The explicit division of interest into domain and application engineering provides an infrastructure for developing highly customized software systems. However, to enable a large scale reuse, SPL engineering explicitly represents the variations for managing dependencies among variants and supporting their instantiation throughout the SPL life cycle (Schmid and John, 2004). The management includes identifying and managing commonalities and variations across a set of system artifacts such as requirements, architecture, code components, and test cases (Babar *et al.*, 2010).

To this end, a product line can be divided into **features**, characteristics that are used to differentiate among members of the product line, and hence to determine and define the common and variable functionality of an SPL (Gomaa, 2005). Kang *et al.* (1990) define *feature* as *"a user-visible aspect or characteristic of the domain"*. Griss (2000) introduces a similar definition. He claims that a feature is *"a product characteristic that users and customers view as important in describing and distinguishing members of the product-line"*. Batory (2006) adopts a more simplistic definition for features, and assumes that *"a feature is an increment in program functionality"*.

Features in an SPL may be classified into two types: **mandatory** and **variable** features. The mandatory features determine the degree of commonality in the SPL. These features must appear in the same way in every product in the SPL. The variable features determine the degree of variability in the SPL (Gomaa, 2005), that is, the products in an SPL are distinguished from each other on the basis of their variant features. Variable features can be either **optional** or **alternative**. Optional are those that need to be provided by only some SPL members. When two or more features are alternatives to each other, only one of them can be provided in a given

SPL member. These are the commonly represented relationships among features.

Variability management encompasses the activities of eliciting and representing variability in software artefacts, in both logical and implementation views. The logical representation is tied to the *problem domain*, and consists of structuring how the product line features can be combined to each other. It is more the view that should be presented to/handled by non-developer stakeholders. On the other hand, representing variability in source code refers to the *solution domain*, and consists of implementing the features using a specific mechanism (Jaring and Bosch, 2004; Svahnberg *et al.*, 2005). The source code of an SPL is usually more complex than that of single systems because of the increased variability, what makes it harder to understand (Kästner *et al.*, 2008).

Furthermore, variability management also encompasses establishing and managing dependencies among different variabilities, and supporting the exploitation of the variabilities for building and evolving an SPL (Chen and Babar, 2011). Also, as the management is usually performed at different levels of abstraction and across all generic development assets, there is a need of effective methods, techniques, and tools to provide adequate support (Bosch *et al.*, 2002; Sinnema and Deelstra, 2007; Chen and Babar, 2011).

3.2.1 Variability modeling

Numerous approaches have been proposed for modeling variability in an SPL (Kang *et al.*, 1990; Czarnecki and Eisenecker, 2000; Gomaa and Eonsuk Shin, 2002; Dashofy *et al.*, 2002; Schmid and John, 2004; Gomaa, 2005; Pohl *et al.*, 2005; Dhungana *et al.*, 2011). Among them, **feature modeling** is perhaps the most popular technique. It was firstly proposed in the Feature-Oriented Domain Analysis (FODA) (Kang *et al.*, 1990) method, and has been used in many different software development paradigms, like model-driven development (Trujillo *et al.*, 2007), feature oriented programming (Batory, 2003), software factories (Greenfield and Short, 2003), generative programming (Czarnecki and Eisenecker, 2000), and so on. A feature model describes the type of features, the relationships and dependencies among features and their interrelationships, i.e., defining the way decisions about variable features influence each other, such as stating that one variable feature requires another, or that one feature be incompatible with another. These decisions are made for a given SPL member by configuring the feature model. Voelter (2009) states that a great advantage of configuration is its simplicity, in a sense that it is not necessary for an engineer to learn complex formalisms for defining an SPL variant, but rather to simply select from a predefined set of alternatives.

Feature models are tree-like structures in which features are depicted using labeled boxes.

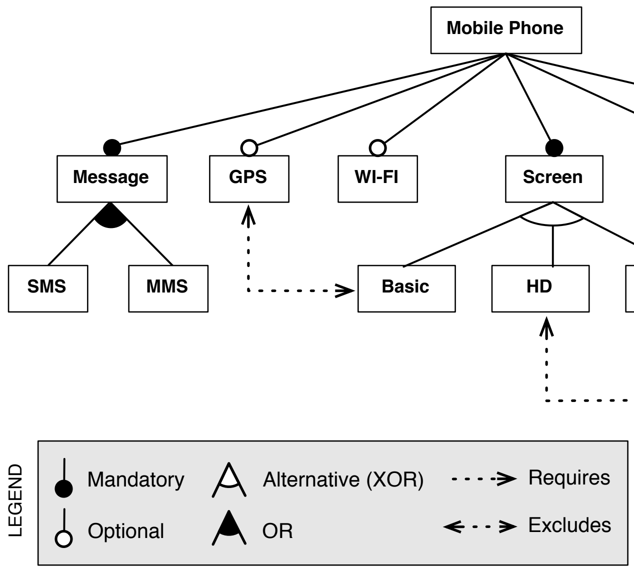


Figure 3.2: Sample Feature Model.

Each feature in the model may have a set of child features with which it can interrelate in a given type of relationship: *mandatory, optional*, or *alternative*. They describe the features that can appear in a member of the product line, to separate the common features from the variable ones, and to indicate how the variable features can appear (Czarnecki and Eisenecker, 2000). Features can be activated and deactivated in order to create a specific SPL variant[2].

Figure 3.2 illustrates a feature model, an excerpt of the **mobile phone** domain. It uses a widely used notation (Czarnecki and Eisenecker, 2000), which includes all types of relationships among features, and **`parent»child`** relations. Dependency constraints are also illustrated. Besides the root feature, that describes the application domain under analysis, the figure presents *three* mandatory features, *three* optional features (variation point), *three* OR-group features (variation point), from which one or more (variant) features may be selected for an SPL member, and *one* alternative group of features, from which exactly one feature must be selected. Regarding constraint dependencies, whenever the feature `Camera»Front` is selected, the feature `Screen»HD` must be selected as well. It represents the *requires* constraint. On the other hand, whenever the feature `GPS` is selected, the feature `Screen»Basic` must not be selected, and vice-versa.

[2]Powerful tools exist to manage features and their relationships, e.g. FeatureIDE (Thüm *et al.*, 2014), pure::variants (Beuche, 2012). Lisboa *et al.* (2010) systematically reviewed the literature aiming at finding out available tools that offer support to such a task. They analyzed the main characteristics of the selected tools, pinpointing their potential benefits, and leveraged still existing gaps to the complete support.

From this sample feature model, we could generate a number of 168 valid configurations[3]. A valid configuration is one in which all rules are satisfied, or, in terms of the concrete syntax used in Fig 3.2, a valid subtree that satisfies all cross-tree dependencies.

There are complex analyses that can be made over feature models, in addition to determining whether a configuration is either valid or not. Benavides *et al.* (2010) provide state-of-the-art information on the analysis of feature models, leveraging the operations employed in model verification. The operations are used to identify anomalies in the models, and to detect structural and integrity constraint violations as well. They both help verifying the ability of the product line model to generate all the desired products, and only them, and measure the quality of the model, as a means to identify and correct the defects in the model, a vital task for efficient management and exploitation of the SPL (Salinesi and Mazo, 2012).

Besides using feature models to expose user-visible features as requirements, they can also be used for design, implementation, testing, and deployment features visible only to developers. Some scenarios enable linking these features, so that user-visible features match a (sub-)set of developer-visible features. An illustrative example could consider linking user-visible front-end features to developer-visible solution-layout features. It could enable the generation of a default solution layout based on the type of front-end chosen by the user.

In SPL engineering, *feature models* can be stated as the *de-facto* standard to model variability (Kang *et al.*, 1990; Czarnecki and Eisenecker, 2000; Chen and Babar, 2011). However, feature modeling is just one of many ways that can be used to describe variability. Others include forms, tables, wizards, designers, templates, patterns, scripts, and code. They can be used alone, and in various combinations.

A well-known alternative approach to feature models is the Orthogonal Variability Model (OVM) technique, introduced by Pohl *et al.* (2005). The OVM technique provides variation points, variants, variability dependencies, and constraint dependencies to define the variability of an SPL. Unlike feature models, OVM is not hierarchical. Technically, it does not describe features and their relationships but rather variation points. It is mainly aimed at explicitly defining and managing the variability of an SPL without considering the common features. However, the two representations are semantically equivalent (Roos-Frantz, 2009).

Furthermore, some researchers proposed the so-called *Constraint Based Product Line* language. They aimed at translating the relationships between features into semantics, so as to denote cross tree constraints by using propositional logic formulas (Czarnecki *et al.*, 2005; Thüm

[3]We calculated such a value in the SPLOT tool (Mendonca *et al.*, 2009). This sample feature model (Mobile-Phone) is available at the SPLOT's feature model repository: `http://gsd.uwaterloo.ca:8088/SPLOT/index.html` and `http://gsd.uwaterloo.ca:8088/SPLOT/models/model_20140503_110898123.xml`

et al., 2009; Batory *et al.*, 2011; Passos *et al.*, 2011). Consequently, they can capture constraints that are not covered by the feature model tree. Besides, with the automated analysis of the models, it is possible to identify and automatically eliminate inconsistencies, thus reducing the scope of products to handle.

3.2.2 Variability implementation

As presented in Figure 3.1, the SPL development life cycle makes the asset goes through a number of phases. Each phase has its own representations, and it is possible to state that development consists of transformations of these representations (van Gurp *et al.*, 2001). For example, a SPL requirement specification can be transformed into a feature model. Next, the feature model and the requirements will serve as the basis to design the architecture. After that, this will form the basis of the detailed design, from which the source code development will be based upon. The source code is compiled, linked and finally run. The variability contained in the source code must reflect the variability early defined in the previous assets.

In order to make the final product satify the variability requirements, variability mechanisms are employed in the development process. This is the way to enable multiple configurations of an SPL.

In the simplest case, configuration can be achieved by simply setting flags in a configuration file. The literature offers a large amount of variability mechanisms, that consider variability in several steps during development life cycle (Svahnberg *et al.*, 2005; Kim *et al.*, 2005; Mohan and Ramesh, 2007; Deelstra *et al.*, 2009). An important project decision concerns to the variability mechanisms to use in the artifacts development in order to encapsulate the variable parts and provide appropriate support for instantiating the variation mechanism. As the purpose of this thesis encompasses variability issues at the source code level, hence the focus is on how variability corresponding to functionality can be managed at this level. Well-established techniques like conditional compilation, inheritance, or parameterization can be used as variability implementation mechanisms (Svahnberg *et al.*, 2005; Bosch and Capilla, 2013). Table 3.1 lists a set of common mechanisms, just to name a few.

In the early days of SPL adoption, in industrial practice, the variability used to be often realized using macro preprocessors (conditional compilation). A reason for this is the widespread availability and know-how resulting from the distribution of such a tool with C and C++ languages (van Gurp *et al.*, 2001). The C preprocessor is often the standard tool to introduce variability to software, especially for its simplicity and flexibility. Despite the use of proprocessors has been criticized in academia, as it usually introduces problems surrounding

Table 3.1: Variability Implementation Mechanisms.

Variability Mechanisms			
• Aggregation/Delegation	• Aspects (AOP)	• Conditional Compilation	• Dynamic Class Loading
• Dynamic Link Libraries	• Frames Reflection	• Inheritance	• Overloading
• Parameterization	• Properties	• Static Libraries	

comprehensibility and correctness (Le *et al.*, 2011), novel techniques have been proposed to cope with such problems, and keep the use of preprocessors a cost-effective strategy to implement variability (Feigenspan *et al.*, 2013; Machado *et al.*, 2014a).

Therefore, over the years, object-oriented development has become an alternative product line implementation technology (Fazal-Amin *et al.*, 2011), with a large set of variability mechanisms available at the code level. Other technologies such as aspect-oriented software development (Alves *et al.*, 2008) also gained interest by the research community, as an alternative to macro preprocessors.

Whereas these more recent variability technologies aim at reducing the problems identified in the use of preprocessors, there are some inherent problems, as follows. Given that some of the mechanisms are used both in ordinary algorithmic implementation, as well as for managing variation points (Bosch and Capilla, 2013), it is sometimes rather difficult to establish a seamless traceability relation between problem space (feature model) and solution space (source code), especially if the SPL is barely documented. Some research initiatives, e.g., (Anquetil *et al.*, 2010; Cavalcanti *et al.*, 2011b; Santos *et al.*, 2012; Linsbauer *et al.*, 2013; Tsuchiya *et al.*, 2013), however, have proposed solutions to produce reliable traceability relations at low levels of precision and reasonable levels of recall, based on ideas from single system development (Antoniol *et al.*, 2002).

Variability mechanisms are architectural patterns, design patterns, idioms, or guidelines for coding (Fritsch *et al.*, 2002). There are three main indispensable characteristics a mechanism must offer: *implementation of the specified options*, *a technique to select the options for a certain product configuration*, and *the binding time*. This latter specifies when the variability described by a specific feature is bound in the development proces. The binding time refers to either the time at which a variation was assigned to a variation point, or the latest time during the development when a variation can be bound to a variation point (Deelstra *et al.*, 2009).

The binding time of variability restricts the use of a variability mechanism, that is, if the variability is to be bound at run time, it is not possible to implement it with a mechanism which

Table 3.2: Typical Binding Times with respect to Implementation.

Binding Time	**When Variability is Resolved**	**Example Mechanisms**
`Compile-time`	The variability is resolved before the actual program compilation or at compile time	Function overloading, precompiler (preprocessor directives), template evaluation, static aspect weaving
`Link-time`	The variability is resolved during module or library linking	DLLs, class loading
`Runtime`	The variability is resolved during program execution	Virtual functions, inheritance & polymorphim, factory-based instance creation, delegation, meta programming, data driven (tables, interpreters)
`Post-runtime`	The variability is resolved during program updates or after program execution	Update utility which adds functionality to existing modules

is bound at compile time. Krueger (2004) provides an extensive list of binding times. From these, Table 3.2 presents typical binding times with respect to the *implementation level*, together with a brief description and example variability mechanisms, based on the work of Gacek and Anastasopoules (2001).

3.3 SPL adoption strategies

An orthogonal issue to the SPL development methods refers to the adoption strategy (Alves *et al.*, 2006; Bastos *et al.*, 2011). Krueger (2001) termed three broad SPL adoption strategies: *proactive*, *reactive* and *extractive*. In the proactive strategy, the organization analyzes, designs, and implements a complete SPL to support the full scope of products needed on the foreseeable horizon, that will pay-off at some point after enough applications within the SPL are generated (Rashid *et al.*, 2011). As opposed to this strategy, both the reactive and extractive strategies capitalize on existing systems within the company. These can be enacted by the application of program refactorings (Alves *et al.*, 2006). In the former, the organization incrementally grows their software product line when the demand arises for new products or new requirements on existing products. All involved assets, i.e., feature models, source code, and so on, are incrementally extended in reaction to new requirements. In the latter, the organization capitalizes on existing custom software systems by extracting the common and varying source code into a single production line.

In all strategies, variability management must be addressed in the domain: while focusing on exploiting the commonality within the products, adequate support must be available for composing SPL core assets with product-specific artifacts in order to derive a particular SPL instance (Alves *et al.*, 2006).

A proactive strategy might be the most desirable one for any organization, due to its capability of defining the variability early in the SDLC, thus enabling controlling the traceability on what is likely to vary within the assets (Cavalcanti *et al.*, 2011b). However, it may be less frequent in practice than the other strategies, due to its incurred high upfront investment and risks (Alves *et al.*, 2006), particularly for small to medium-sized software development organizations with projects under tight schedules (Bastos *et al.*, 2011). In this strategy, although not every asset should be implemented before starting to build products, it is necessary to design and plan for the development of all assets that will be part of the product line, what demands a high investment. In contrast, the inherently incremental nature of the reactive strategy offers a quicker and less expensive transition into SPL engineering. Similar benefits are provided by the extractive strategy, in which the high level of software reuse enables an organization to very quickly adopt an SPL. These two strategies can be more suitable.

Nevertheless, these strategies are not necessarily mutually exclusive. It is possible to bootstrap an SPL effort using the extractive approach, by applying program refactorings in existing software products (Alves *et al.*, 2006), and then move on to a reactive approach to incrementally evolve the production line over time (Krueger, 2001).

3.4 Chapter summary

This Chapter introduced the general principles of SPL engineering. In a nutshell, we could state that SPL engineering is the software version of an old manufacturing concept: to build a suite of products from common parts assembled under a common design in a common production facility.

While traditional software engineering practices are focused on developing and maintaining single products, such a strategy might not be efficient in cases where groups of products are related. SPL engineering exploits the commonalities among products to achieve economies of scale, by creating core assets, and accommodates the differences among products by explicitly identifying and planning for those variations in product behavior and qualities.

The special properties of software make SPL engineering a particularly high-payoff proposition. The management of variation allows the organization to achieve economies of scope and provides the capability of mass customization where every market is treated as a niche. Management ensures that core asset developers create effective core assets and product builders efficiently build products by using them.

However, despite the SPL promised benefits, guaranteeing the quality of the products in an SPL deserves special attention. Testing, as an important quality assurance technique, is especially chalenging in an SPL. Testing the differences in a cost-effective way can become complex because in realistic product lines, variability abounds. The next Chapter elaborates on the main concepts of testing in SPL engineering, in the light of existing knowledge. We discuss the current state-of-the-art, pinpointing the still existing gaps to bridge.

4 Software Product Lines Testing

Testing plays an important role in the quality assurance process for SPL engineering. There are many opportunities for economies of scope and scale in the testing activities, but techniques that can take advantage of these opportunities are still needed.

This Chapter reports on the state of the art aspects of the SPL testing field. It is the result of a systematic literature review carried out to identify SPL testing practices that have the potential to achieve these economies, and to synthesize available evidence on the strategies for testing SPL projects (Machado *et al.*, 2014b).

The analysis of the reported strategies comprised two fundamental interests for SPL testing: *the selection of products for testing*, and *the actual test of products*. In the former, the investigation addressed the generation of representative sets of products from domain models, in order to sketch which techniques could be used and what their properties are. The latter addressed the existing support for testing end-product functions, which takes advantage of commonality and variability.

The review encompassed the analysis of two hundred seventy-six studies published from the year 1998 up to the 1st semester of 2013. Several filters were used to focus the review on the most relevant studies, and detailed analyses of the core set of studies was provided.

With the review, we could leverage a number of strategies that applied to support both the selection of products, and the actual testing of products. However, the findings indicate that the literature offers a large number of strategies to cope with such aspects. However, a lack of reports on realistic industrial experiences might be observed, which limits the inferences that can be drawn.

This Chapter details how the review was conducted, and presents the main results. It is structured as follows. Section 4.1 introduces fundamentals concepts of SPL testing. Section 4.2 describes the research method used in the review. Section 4.3 presents the results of the review.

The main findings are discussed in Section 4.4, together with the threats to validity. Section 4.5 discusses related work and, finally, Section 4.6 concludes the Chapter.

4.1 Introduction

Guaranteeing that every feature in an SPL will work as expected, and ensuring that combinations of features will work in each product is often problematic because of the high costs involved. Exhaustive testing is seldom feasible in any development process. This is particularly infeasible in SPL due to the variability in features, due to the many input variables.

A scoping review, carried out as background to the present review (Machado *et al.*, 2012b), revealed two independent but complementary interests an SPL testing strategy should handle, as follows:

- Firstly, it is necessary to check the feature interaction coverage, i.e., when checking the properties or configurations of an SPL, every feature combination has to be consistent with the specification and must not violate the stated constraints.

- Secondly, it is necessary to check the set of correctness properties of each product. Given that a built software artifact can be used by a range of products, an uncovered defect may be propagated to the many products that include it.

The *first interest* considers testing generation as a systematic selection of a representative set of product instances, comprising a subset of all possible configurations in an SPL. The main idea is to reduce the testing space. Figure 4.1 illustrates the first interest. It shows that test cases refer to product configurations, i.e., a test case is responsible for testing whether an instance of the feature model (or whatever represents the variability in an SPL) is valid or not.

There are two main inputs to consider for this *interest*: **a set of product requirements** and the **quality of the variability model under test**. The role of requirements is to establish what functionalities a product instance should encompass. Regarding the quality of the variability model, through the analysis of its consistency, we could ensure that the produced models are complete and correct, in the sense that there are no conflicting restrictions between features, and that all the important distinctions and differences in a domain are covered by the model.

The *second interest* focuses on performing testing on end-product functionalities. Such an interest deals with the systematic reuse of test assets and results, as a means to reduce the overall effort, and avoid retesting of already tested features, while being effective at revealing faults. Test assets are designed to test the functionalities of features that will compose the products.

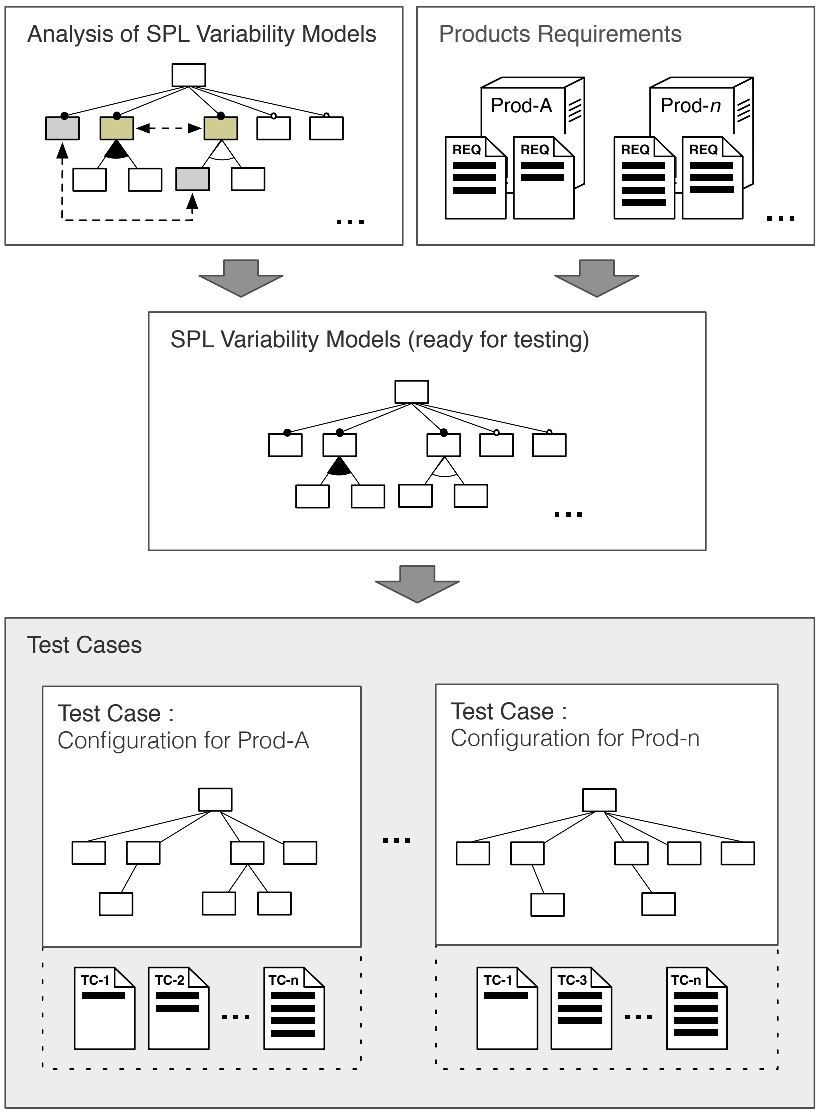

Analysis of SPL Variability Models
Products Requirements
Prod-A
Prod-n
REQ
SPL Variability Models (ready for testing)
Test Cases
Test Case : Configuration for Prod-A
Test Case : Configuration for Prod-n
TC-1
TC-2
TC-n
TC-3

Figure 4.1: SPL Testing interest: selection of product instances to test.

Figure 4.2 illustrates how this interest works. It comprises both *domain* and *application engineering* processes (Pohl *et al.*, 2005). From a testing standpoint, the former defines the variable test assets (test cases, test scenarios etc.), taking as input the variability defined for the SPL. In the later, when a product variant is instantiated, the variable assets are bound, according to the requirements expected for that particular product instance. Existing testing assets are also bound, as it is encouraged the reuse of test assets between product instances.

In both interests, a particular problem is the number of test inputs to consider, which can increase exponentially with the number of features that an SPL comprises (Perrouin *et al.*,

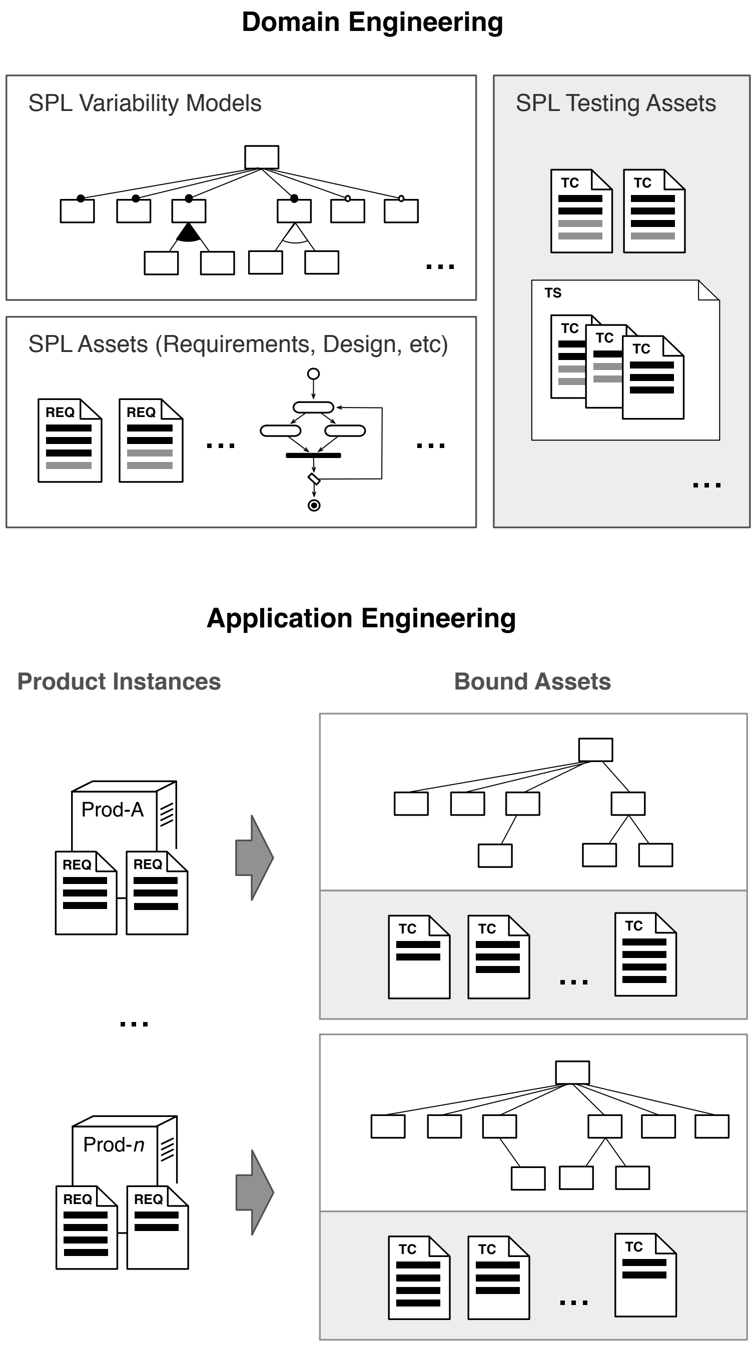


Figure 4.2: SPL Testing interest: actual test of products.

2010). Therefore, designing and/or selecting an effective[1] set of test cases, considering the likely amount of test inputs, play an important role in SPL testing.

Considering the importance of knowing which test case design and selection techniques

[1]By *effective* we mean the defect revealing ability of a test suite (Naik and Tripathy, 2008).

the current SPL practice adopts, a detailed insight from the point of view of the interests aforementioned would be valuable. To this end, we conducted a *systematic literature review*, aiming at identifying, assessing, and interpreting available research evidence in the SPL testing research field, for the purpose of clear categorization. We investigated how studies address the generation of representative sets of products, from domain models, in order to sketch which techniques could be used and what their properties are. Yet, we investigated the existing support for testing end-product functions by taking advantage of the specific features of a product line, i.e., commonality and variability. We intend to collect evidence about current research that suggests implications for practice, and to identify open problems and areas that need attention.

4.2 The review method

A Systematic Literature Review (SLR) is a rigorous, systematic, and transparent method to identify, appraise, and synthesize all available research relevant to a particular research question, or topic area, or phenomenon of interest, which may represent the best available evidence on a subject (Cruzes and Dybä, 2011). A SLR may serve as central link between evidence and decision making. They provide the decision-maker with best available evidence. This evidence, in combination with field expertise and the customer-related values, characteristics, and circumstances, are necessary ingredients for making good decisions.

The importance of SLR for software engineering has been addressed by a reasonable amount of studies, as deeply discussed in (Cruzes and Dybä, 2011; Kitchenham *et al.*, 2009; Dybå and Dingsøyr, 2008b; Brereton *et al.*, 2007). Kitchenham and Charters (2007) describe a set of reasons for undertaking a SLR, as follows:

- to review the existing evidence concerning a treatment or technology;
- to identify gaps in the existing research, which may indicate areas for further investigation;
- to provide a context/framework in order to properly position new research activities.

The review was carried out by following Kitchenham's guidelines for performing SLR in software engineering (Kitchenham and Charters, 2007), which comprises the following steps: *development of a review protocol*, *conducting the review*, *analyzing the results*, *reporting the results* and *discussing the findings*. The protocol specifies all steps to take in the review, and increases its rigour, transparency, and repeatability, while establishing a means to reduce risk of bias. The protocol includes: *(i)* the strategy employed to define the research questions, based upon an explicit research objective; *(ii)* the systematic searching methods that reduce the risk of selective sampling of studies, which may support preconceived conclusions, thus reducing risk of bias, and *(iii)* the method employed to evaluate available information.

The methodology employed in the SLR included formulation of research questions to attain the objective, as next detailed in Section 4.2.1; identification of sources from where research papers were to be extracted, described in Section 4.2.2; and also the search criteria and principles for selecting and assessing the relevant studies, discussed in Section 4.2.3.

4.2.1 Research questions

The *Population, Intervention, Comparison, Outcome* and *Context* (PICOC) structure was used to define the research questions, as defined by Petticrew and Roberts (2006). This structure describes the five elements to consider when defining a searchable question. Articulating a research question in terms of its elements facilitates searching for a precise answer.

This review was not intended to sketch any comparison of interventions, therefore the element *comparison* is not applicable. Table 4.1 shows the PICOC structure.

Table 4.1: PICOC structure

Element	Description
Population	Software product line testing research.
Intervention	Approaches, i.e., methods, strategies, techniques, and so on, that support testing in SPL engineering.
Comparison	n/a.
Outcome	The effectiveness of the testing approaches.
Context	Within the domain of SPL engineering, with a focus on testing approaches.

Considering that the two SPL testing interests discussed in Section 4.1, hold different overarching goals, studies from both categories should be analyzed in a proper manner. Hence, our SLR aims to answer the following research questions:

- **RQ1.** What SPL testing strategies are available to handle the selection of products to test?
- **RQ2.** What SPL testing strategies are available to deal with the test of end-product functionalities?
- **RQ3.** What is the strength of the evidence in support of these proposed SPL testing strategies?

- **RQ4.** What are the implications of these findings for the software industry and the research community?

We defined the RQ1 to get an in-depth view on how existing SPL testing techniques cope with the *selection of product instances for testing* (first SPL testing interest). It considers the configuration of features as the main input for the design of test cases.

In addition, RQ2 aimed at carrying out tests of end-product functions, as we intended to gather the SPL strategies used to handle the *actual testing of SPL assets* (second SPL testing interest). The RQ2 considers core assets as the input for designing the test cases. Recall that core assets are those assets that form the basis for the SPL. They are produced in Domain Engineering phase. They often include, but are not limited to, the architecture, domain models, requirements statements, reusable software components etc.

Within the set of strategies identified in RQ1 and RQ2, we observed whether tool support was available to practitioners.

Furthermore, RQ3 helps researchers assess the quality of existing research. The results of this question are critical for researchers to identify new topics for empirical studies, and for practitioners to assess the maturity of a proposed strategy. RQ4 help us outline directions for future research and identify areas that need work in order to make strategies more applicable in industrial practice.

4.2.2 Identification of relevant literature

The process of gathering and selecting relevant studies involved three phases, as Figure 4.3 shows. Our initial set of candidate papers was provided by previously published literature reviews on SPL testing (Neto *et al.*, 2011a; Engström and Runeson, 2011; Lamancha *et al.*, 2009). Therefore, the first phase consisted of collecting the primary studies listed in each of these literature reviews, which include research papers published up to the year 2009. For the second phase, we performed a search for studies published from the year 2009 up to the year 2013. The third phase consisted of the screening process to exclude studies that are not relevant to answer the research questions.

Along this Section, we expand on each gathering phase, and detail the study selection procedure.

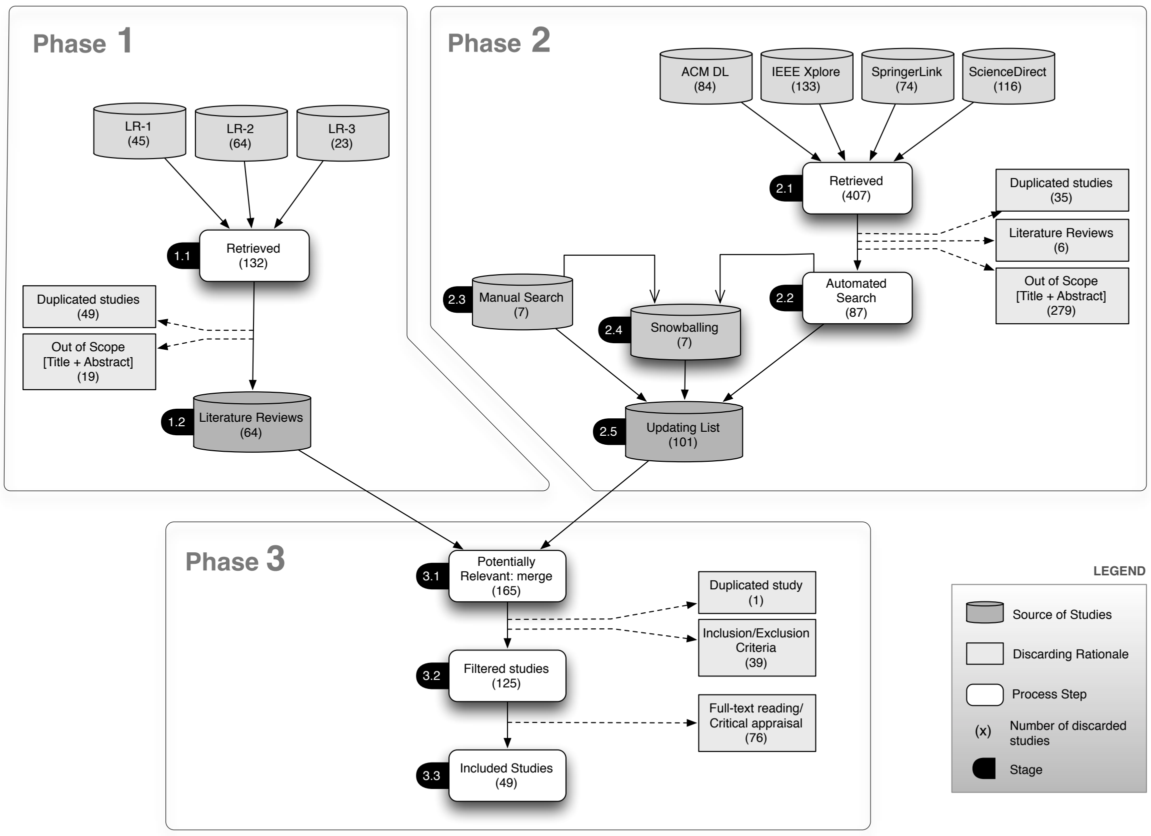


Figure 4.3: Study selection procedure.

4.2.2.1 Phase 1: analysis of existing reviews

In earlier research, we reported on a systematic mapping study of SPL testing (Neto *et al.*, 2011a). By analyzing the literature published up to the year 2009, we identified the major existing practices in the field, setting up *clusters* of studies that could support a fuller review. The systematic mapping study investigated 45 unique publications. The findings were in accordance with an analogous study (Engström and Runeson, 2011), which also systematically mapped out the existing literature on SPL testing, in order to identify useful approaches and needs for future research. This latter analyzed a set of 64 unique publications.

In both reviews, the overall focus was to enlighten researchers and practitioners with a broad picture of research and practice in the field of SPL testing, without providing in-depth analysis of any nature whatsoever.

A third study was performed with similar goals (Lamancha *et al.*, 2009). In a systematic literature review, a group of reseachers analyzed 23 unique publications. Unlike the two previously mentioned mapping studies, published as journal papers, this third review was published as a conference paper, which, due to space constraints, might have limited the treatment of some required details, as a reader could expect.

These three studies complement each other in terms of research questions investigated. All of them have in common their overall goal, namely to provide an overview of the SPL testing field, pointing out achievements, opportunities, problems and available resources.

Thus, the initial set of primary studies for this SLR was gathered from these literature reviews, respectively *LR-1* (Neto *et al.*, 2011a), *LR-2* (Engström and Runeson, 2011), and *LR-3* (Lamancha *et al.*, 2009). Given that they followed systematic processes of gathering, selecting, and assessing the studies (Kitchenham and Charters, 2007; Petersen *et al.*, 2008), we acknowledge that they are a representative sampling of all primary studies in the SPL testing field, for studies published up to early 2009.

Within the reviews, we identified 132 potentially relevant papers, illustrated as **stage (1.1)** in Figure 4.3. We read the *titles* and *abstracts* of the publications, to identify and exclude those which bear no relation to our investigation, i.e., studies that are not suitable to answer the RQs. This stage of screening was carried out by two independent reviewers who screened and then met to compare their results. Any disagreement or uncertainty was discussed and arbitrated by a third independent reviewer.

Besides, as some studies were included in more than one literature review, we removed the overlap. In the end of such a screening, we had a set of 64 studies, illustrated as **stage (1.2)**.

4.2.2.2 Phase 2 - gathering recent publications

The second phase of the *search process* consisted of an update on the list of primary studies. We analyzed the literature published between *2009* and the first semester in *2013*.

We performed an automated search in the following search engines and indexing systems: *ScienceDirect, ACM Digital Library, IEEE Xplore*, and *SpringerLink*. These are very useful online databases, since they index IEEE, ACM, Springer and Elsevier publications which, together, provide many of the leading publications in the Software Engineering field. Therefore, they are likely to include major SPL testing-related venues. Here we use the word *venue* as a generic name for journals or conferences (including workshops).

From the stated research questions, and known terms of the SPL testing domain, we identified the *keywords* to use in the search process. We applied variants of the terms *"software product lines"*, *"software product family* [2]*"*, *"software testing"* and *"testing techniques"*, to compose

[2]Software product family is a commonly used synonym for SPL (Clements and Northrop, 2001).

the search query. This was coded to fit the syntax requirements and capability of the search engines of each data source used. Table 4.2 lists the search strings applied in each search engine. The Table also shows the number of results retrieved from each.

Table 4.2: Detailed search strings applied in the automated search engines.

Engine	URL	Search string	Results*	
			Raw	Refined
IEEE Xplore	http://ieeexplore.ieee.org	(((software product line) OR software product family) AND test)	158	133
Springer	http://www.springerlink.com	'software product line' AND '(test, OR testing)' within 2009 - 2013	77	74
ScienceDirect	http://www.sciencedirect.com	pub-date >2008 ((“software product line” OR “software product lines” OR “software product family”) AND (“testing” OR “test”)) AND LIMIT-TO(topics, “software, product line”) AND LIMIT-TO(yearnav, “2013,2012,2011,2010,2009”)[All Sources(Computer Science)]	129	116
ACM DL	http://dl.acm.org	(“software product lines test”) OR (“software product line test”) OR (“software product line testing”) OR (“software product lines testing”) OR (“software product family testing”) OR (“software product family test”)	92	84

(*) **Raw results** means the whole set of entries listed by the search engines. Inasmuch as not only research papers are retrieved, but also *cover letters, foreword, preface, guest introductions etc.*, hence, we filtered out those entries, and list the total number of research papers retrieved in the column **refined results**.

From this task, we obtained a set of 407 publications, depicted as **stage (2.1)** in Figure 4.3. As we considered studies retrieved from different search engines, 35 articles were excluded because they were duplicates, i.e., retrieved in more than one search engine. This search also retrieved related work, such as the literature reviews analyzed, and similar studies. We discarded these 6 publications, since they have been considered in other respects in this study, such as either source of studies or related work.

Next task included reading the *title* and *abstract* of each remaining paper, similarly as in the previous phase. In this sense, a set of 279 articles were found to be irrelevant, as they did not consider testing from an SPL standpoint, but instead addressed issues from single-system software development. In the end, we had a pool of 87 publications from the automated search, depicted as **stage (2.2)** in Figure 4.3.

Upon completion of the automated search, we carried out a manual search aiming at increasing coverage and quality (Zhang *et al.*, 2011; Kitchenham *et al.*, 2010), considering the same time span as in the automated search. Some important journals and conference proceedings were individually searched. Table A.1 in Appendix A lists the venues, and their usual abbreviations. They are clearly representative of the Software Engineering field. Such a claim is grounded in observations in related work, and also in discussions with colleagues. The task retrieved an additional 7 publications, as shown as **stage (2.3)** in Figure 4.3.

In addition, reference lists of all identified publications were manually scanned, in order to identify missing relevant papers. This is the process called *snowballing* (Webster and Watson, 2002). This task resulted in an additional 7 articles, as Figure 4.3 illustrates in **stage (2.4)**. With such a small number of additional papers, we are convinced that we have a representative sampling and, furthermore, that the pool of the papers we have are relevant from a SPL engineering perspective. At the end of Phase 2, we revealed a universe of 101 new publications.

4.2.2.3 Primary study selection strategy

By merging the results from Phases 1 and 2, the list of potentially relevant studies was then composed of 165 publications. **Stage (3.1)** in Figure 4.3 shows such an amount. We identified one duplicate, a study listed in *LR-1* that was also retrieved in the automated search. This was due to the year 2009 was considered in both *LR-1* and in the automated search phase.

We established a set of *inclusion* and *exclusion criteria* to assess each potential primary study. They were applied to the titles and abstracts of identified articles.

The criteria were specified based on the analysis scope of found papers to guarantee that only works really related to the context of testing approaches[3] for SPL should be selected as the primary studies of this SLR.

This task aims to ensure that relevant studies are included and no study is excluded without thorough evaluation. At the outset, studies are only excluded if they clearly meet one or more of the exclusion criteria.

To be included in the review, studies had to present an approach to cope with SPL testing and at least one empirical evaluation to demonstrate the approach's feasibility. It is worth mentioning that only studies produced in English were included. Table 4.3 presents the exclusion criteria.

At the end of the screening process, we ended up with a pool of 125 studies, to be subject of full-text reading, depicted as **stage (3.2)** in Figure 4.3.

[3]We herein generalize the term *approaches* to include not only actual *approaches* but also *strategies, methods, techniques,* and *processes*, given that authors sometimes interchangeably use those terms to define their proposal.

Table 4.3: Exclusion criteria and list of excluded studies.

Exclusion Criteria	Rationale
Related Work	Secondary studies were not considered in this review. Such a kind of study were analyzed as related work.
Abstracts	We excluded prefaces, editorials, and summaries of tutorials, keynotes, panels and poster sessions.
Doctoral Symposium	Studies published in *doctoral symposia* were also discarded. To the best of our knowledge, this kind of study does not bring information other than a *status report* on the doctoral thesis work, and usually make reference to more complete studies, published elsewhere.
Extended Studies	When several duplicated articles of a study exist in different versions that appear as books, journal papers, conference and workshop papers, we include only the most complete version of the study and exclude the others.
Position Papers	Position or philosophical papers, i.e., papers only presenting an anecdotal evidence of the SPL testing field, were excluded from the literature review, but some of the position papers showing future directions are mentioned in the conclusion and future work section.
Comparative Papers	Comparative papers, with no additional contribution, but rather only analyzing existing literature, that eventually were included in our primary studies list.
Out of scope	By analyzing the introduction section of a study, it is possible to figure out what the topic under investigation is about. Based on this statement we discarded studies which did not deal directly with testing, but instead that consider SPL in a general viewpoint.

4.2.3 Data extraction

After using the criteria to select relevant papers, we undertook a comprehensive analysis of the 125 filtered studies, to collect data necessary to answer the research questions, and to measure their quality.

The data were extracted and stored in a spreadsheet after reading each paper using a data extraction form. The form included the following attributes: **title, authors, corresponding email address, year of publication, source of publication, publication type, notes**, and an additional set of attributes, as listed below:

- **Research result.** We analyzed the outcomes of each primary study, and classified their main findings according to the types of software engineering research results (Shaw, 2002): *procedure or technique, qualitative or descriptive model, empirical model, analytic model, notation or tool, specific solution, answer or judgment*, or *report*.

- **Evidence gathering method.** We evaluated the evidence level (Lev1-6) reported in the study. Such assessment might lead researchers to identify new topics for empirical studies, and for practitioners to assess the maturity of a particular approach. Kitchenham et al. classified six levels of study design, based on the evidence hierarchy suggested from medical research (Kitchenham and Charters, 2007). Later on, Alves *et al.* (2010) described a tailored classification that could be fully applicable to the interest of this review. The classification includes the following hierarchy (from weakest to strongest):

1. No evidence.
2. Evidence obtained from demonstration or working out toy examples.
3. Evidence obtained from expert opinions or observations.
4. Evidence obtained from academic studies.
5. Evidence obtained from industrial studies.
6. Evidence obtained from industrial practice.

- **Industry.** In case a study was evaluated in industry settings, matching evidence levels 5 and/or 6 above, this attribute was used to identify the application domain in which authors carried out the evaluation.
- **SPL testing interest.** As a means to analyze techniques matching either one of the SPL testing interests, earlier discussed in this article, we decide to verify which interest is the main concern in each analyzed primary study.

As well as for the initial screening stage, the procedure of reading and completing the extraction form for each paper was again conducted by two independent reviewers, and any discrepancies were resolved by calling upon a third reviewer.

4.2.4 Quality assessment

We also quality appraised each study remained for data extraction, using a set of quality criteria. We extracted the criteria mainly from the questionnaire for quality assessment proposed by Dybå and Dingsøyr (2008a), which is based on principles of good practice for conducting empirical research in Software Engineering (Kitchenham and Charters, 2007), and also on the Critical Appraisal Skills Programme (CASP)[4].

[4]http://www.casp-uk.net/

The questionnaire used to critically appraise the quality of the selected studies contained 11 closed-questions, as listed in Table 4.4. Taken together, the criteria provided a measure of the extent to which we could be confident that findings of a particular study could provide the review with a valuable contribution. The criteria covered the four main issues pertaining to quality that need to be considered when appraising the studies identified in the review (Dybå and Dingsøyr, 2008a):

- **Reporting.** Reporting of the study's rationale, aims, and context.
- **Rigour.** Has a thorough and appropriate approach been applied to key research methods in the study?
- **Credibility.** Are the findings well-presented and meaningful?
- **Relevance.** How useful are the findings to the software industry and the research community?

From the criteria obtained from Dybå and Dingsøyr (2008a), we only made a few changes to customize the criteria for *relevance* assessment, as a means to evaluate the relevance of the study for the software industry at large and the research community.

Each of the 11 criteria was answered with either "yes" (1) or "no" (0). Then, a quality assessment score was given to a study by summing up the scores for all the questions for a study. The resulting total quality score for each study ranged from 0 (very poor) to 11 (very good).

We used quality assessment criteria for both synthesis purposes and filtering papers. For the second matter, the first criterion was used as the minimum quality threshold of the review to exclude non-empirical research papers. Hence, 76 papers were excluded as part of this screening process.

Upon completion of the quality evaluation assessment, the set of selected primary studies was composed of 49 studies, illustrated as **stage (3.3)** in Figure 4.3. See Table A.3 in Appendix A for a list of selected primary studies.

Results of the quality evaluation are presented in Table A.2 in Appendix A. Again, as QC1 was used as the basis for including or excluding a study, the table shows the assessment for the set of included papers.

4.3 Results of the systematic review

We used the extracted data to answer our research questions. In this section, we initially give an overview of the selected studies with respect to their publication venues. Then, we answer each

Table 4.4: Quality assessment questions.

No.	Question	Issue
QC1.	Is the paper based on research and it is not merely a “lessons learned” report based on expert opinion?	Reporting
QC2.	Is there a clear statement of the aims of the research?	Reporting
QC3.	Is there an adequate description of the context in which the research was carried out?	Reporting
QC4.	Was the research design appropriate to address the aims of the research?	Rigour
QC5.	Was there a control group with which to compare treatments?	Rigour
QC6.	Was the data collected in a way that addressed the research issue?	Rigour
QC7.	Was the data analysis sufficiently rigorous?	Rigour
QC8.	Has the relationship between researcher and participants been considered to an adequate degree?	Credibility
QC9.	Is there a clear statement of findings?	Credibility
QC10.	Is the study value for research or practice?	Relevance
QC11.	Are there any practitioner-based guidelines?	Relevance

research question, based on the extracted information.

4.3.1 Characteristics of the studies

Table 4.5 shows the temporal distribution of the selected 49 primary studies, encompassing the years 2003 through 2013. The table also shows the distribution of the studies based on their publication channels, along with the number of studies from each source: workshops, conferences, journals, and the grey literature[5]. We observe that only 3 out of 49 studies were found in journals, whereas most was published in conferences.

Such a distribution give us the initial impression that most relevant studies in the field were only found in recent publications, i.e., as of the year 2010. Regardless the number of studies removed from our final list for matching any of the exclusion criteria, we should recall that we only included studies which presented any kind of empirical evaluation. Thus, we may notice a trend curve in the data, showing an increasing attention on the use of scientifically rigorous evaluation methods as a means to assessing and making explicit the value of the proposed approaches for the SPL testing field.

[5] *Grey literature* comprises technical reports and book chapters.

Table 4.5: Summary of selected primary studies by publication type and publication year.

Venue	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	Total
Workshop	1	-	1	3	2	-	-	1	-	-	3	11
Conference	-	1	1	1	-	2	1	8	7	4	6	31
Journal	-	-	-	-	-	1	-	1	-	1	-	3
Grey Literature	-	-	-	3	-	1	-	-	-	-	-	4
Total	1	1	2	7	2	4	1	10	7	5	9	49

Table 4.6 lists the venues of all selected studies and the publication count. A list comprising all manually searched venues can be found at Appendix A.1.

The 49 selected primary studies were gathered from 20 conferences, 6 workshops, and 3 journals. The remaining 4 studies were published as book chapters, 3, and one technical report. As expected, the greater amount of studies in a single vehicle was found in the SPLC[6] (10 studies), considered the most representative conference for the SPL engineering area, followed by the SPLiT[7] (4 studies), a workshop dedicated to the SPL testing topic, co-located with the SPLC. The workshop was held yearly for five years, between 2004 and 2008.

During the data extraction process, we collected the SPL testing interest each selected primary study addressed. As the both interests are not mutually exclusive, i.e., a comprehensive process for testing SPL might encompass both, in this analysis we could group the studies according to their central proposal. The histogram in Figure 4.4 shows the number of studies addressing each interest by publication year, where *Interest 1* means *selection of product instances to test*, and the *Interest 2* means the *actual test of products*, as earlier discussed in this article. Table 4.7 shows what studies addressed what SPL testing interest.

This histogram shows a recent growing interest in the proposals towards overcoming the first SPL testing interest. Despite the observed trend, we can state that both interests holds the same importance. Regarding the second one, much has been proposed along the years, so as to enable a detailed analysis of existing practices and describe the inherent challenges.

Besides, despite of the importance of both interests working together in a same technique, only in the year 2013 we found studies dealing with both. Although this is a rather small number of studies, it might be some indication that the research community has realized the benefits of proposing approaches which encompass both interests.

[6]SPLC stands for *International Software Product Line Conference*.

[7]SPLiT stands for *International Workshop on Software Product Line Testing*.

Table 4.6: Study distribution per publication sources.

Source	Count
Conferences	
Intl. Software Product Line Conference (SPLC)	10
Intl. Conference on Model Driven Engineering Languages & Systems (MODELS)	3
Intl. Conference on Software Reuse (ICSR)	2
Intl. Symposium on Software Reliability Engineering (ISSRE)	2
Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)	1
European Software Engineering Conference / Foundations of Software Engineering (ESEC/FSE)	1
Intl. Conference on Modularity (AOSD)	1
Intl. Conference on Advanced Information Systems Engineering (CAiSE)	1
Intl. Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)	1
Intl. Conference on Fundamental Approaches to Software Engineering (FASE)	1
Intl. Conference on Engineering of Complex Computer Systems (ICECCS)	1
Intl. Conference on Enterprise Information Systems (ICEIS)	1
Intl. Conference on Software Testing (ICST)	1
Intl. Conference on Testing Software and Systems (ICTSS)	1
Intl. Conference on Information Technology - New Generations (ITNG)	1
Intl. Conference on Software Engineering and Formal Methods (SEFM)	1
Intl. Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)	1
Intl. Conference on Tests and Proofs (TAP)	1
Genetic and Evolutionary Computation Conference (GECCO)	1
Haifa Verification Conference (HVC)	1
Workshops	
Intl. Workshop on Software Product Line Testing (SPLiT)	4
Workshop on Advances in Model Based Testing (A-MOST)	1
Concurrency, Specification, and Programming workshop (CS&P)	1
Intl. Workshop on Combinatorial Testing (IWCT)	1
Intl. Workshop on Product Line Approaches in Software Engineering (PLEASE)	1
Intl. Workshop on Variability & Composition (VariComp)	1
Journals	
IEEE Transactions on Software Engineering	1
Journal of Software (JSW)	1
Software Quality Journal (SQJ)	1
Book Chapters and Technical Reports	4
Total	49

4.3.2 Strategies to handle the selection of products to test (RQ1)

Variability in features may lead to diverse products composition possibilities (Dordowsky *et al.*, 2011; Tischer *et al.*, 2011). Although it is necessary to test and analyze all possible feature

Table 4.7: Selected studies *vs.* SPL testing interest addressed.

Interest	Studies	Count
1	[P05], [P15], [P18], [P19], [P20], [P21], [P23], [P24], [P25], [P26], [P28], [P29], [P30], [P31], [P33], [P36], [P37], [P38], [P40], [P43], [P44], [P47]	22
2	[P01], [P02], [P03], [P04], [P06], [P07], [P08], [P09], [P10], [P11], [P12], [P13], [P14], [P16], [P17], [P22], [P27], [P32], [P34], [P35], [P39], [P41], [P45], [P48], [P49]	25
Both	[P42], [P46]	2

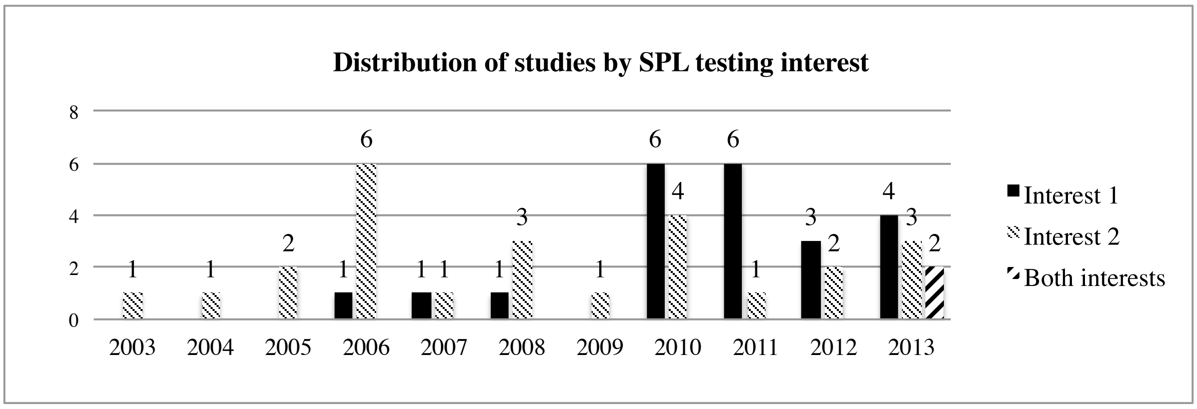


Figure 4.4: Distribution of studies by SPL testing interest and publication year.

combinations, this is unrealistic for variability models of a reasonable size. In this sense, the core problem of the **first SPL testing interest** is to reduce the set of possibilities to a reasonable and representative set of product configurations. While keeping a small sample is critical to limit the effort necessary for testing each selected configuration (Hervieu *et al.*, 2011), this is particularly a combinatorial problem, that should be handled accordingly.

Optimization techniques may be used to prune redundant configurations that need not be explored. From the set of 24 selected studies that cope with this SPL testing interest, we found *combinatorial interaction testing (CIT)* to be the *de-facto* standard to handle test selection in such an interest. CIT enables the selection of a small subset of products where the interaction faults are most likely to occur. This is a cost-effective strategy for SPL engineering, aimed at reducing the test set by selecting the possible combinations of features that will be present in most products. That is, the representative subset of sample products under test. Within the selected studies, this selection is usually based on *combinatorial criteria on feature models*, *coverage criteria on variable test models*, and *coverage criteria on feature interactions*.

A prevalence of formal approches could be noted in those selected studies. For instance, *t-wise* feature coverage has been applied in conjunction with *SAT solvers*, by dividing the set of clauses, transformed from a feature diagram, into solvable subsets. The idea is to automatically generate the test products from a feature diagram that satisfy the *t-wise* SPL test adequacy criteria. *Alloy* can be applied to generate the test cases, and works by capturing all dependencies between features in a feature diagram as well as the interactions that should be covered by the test cases. Some algorithms to translate feature diagrams into Alloy specifications have been proposed, e.g., *FeatureDiagram2Alloy* - P31 -, and *Kesit* - P36. Alloy specifications are used to examine the semantics of features and their interactions, as a means to cope with the problem of scale.

A number of studies - P05, P18, P19, P20, P23, P24, P28, P29, P30, P38, P043, P044 (50% out of the total from the first SPL testing interest group) - applied *pairwise testing*, a specialized notion of *t-wise* coverage, as the main heuristic both for designing and selecting test cases. Pairwise has proven to be most effective when solving problems of large complexity. The underlying assumption is that, by describing the input model as a combination of two features each other, obeying the constraints between them, it might be easier to find inconsistencies, rather than trying to combine all features at once. Establishing pairwise as a test case reduction heuristic may lead to a practical means of sampling all the relevant combination in features models. As a test minimization technique, it aims at identifying and eliminating redundant test cases from test suites in order to reduce the total number of test cases to execute, thereby improving the efficiency of testing.

Pairwise feature interactions serve as an input for test generation based on *constraint programming*, i.e., a paradigm which enables the design of a tailor-made pairwise constraint. This paradigm is well suited for optimization problems such as those related to the minimization of the size of test sets. It maps a feature model into a finite domain constraint model.

All selected studies provide either a process or an algorithmic implementation, aiming to improve test coverage while reducing the overall test effort. However, the observed heterogeneity prevents us from sketching any kind of categorization.

Furthermore, we analyzed the tool support in the proposed approaches. In 17 studies, which represents 71% out of the total amount of selected studies handling the first SPL testing interest, we found a descriptive information about automated tool support. Among them, there are 8 studies - P23, P24, P30, P33, P36, P37, P42, P44 - which provide details about the proposed tool support, in terms of their goals and implemented requirements. They are aimed at analyzing feature models and automatically generate a set of test configurations that cover interactions (usually *pairwise*) between features in a feature model. Furthermore, the study P42 handles test case selection. In all of these studies it is possible to check the algorithm that implement the approach.

While this was expected to be a relevant aspect to enhance transparency in their proposals, an important downside might be observed, that is, from such a set, only 2 studies make reader aware of how the tool could be obtained. The main reason is that, the proposed algorithms and tools are often developed solely to demonstrate a particular nuance of a methodology, mostly within the context of a research group.

Furthermore, in the remaining studies, authors from P05, P20, P25, P28, P29, P31, and P38 claimed that their approaches should be implemented in a tool support; however, they only described the algorithms associated to the proposal. No other information could be found. In one study - P18 -, carried out as an industrial case study, authors stated that an *in-house* tool was used to support their proposal. For this reason, they could not provide readers with further details. In P21, authors stated that their proposal works in conjunction with a tool that was developed beforehand, and discussed in a preceding publication.

4.3.3 Strategies to handle the test of end-product functionalities (RQ2)

Whereby research on configuration-aware software testing for SPL focuses on the combinatorial problems during interaction testing by detecting valid and invalid combinations of configuration parameters, the SPL testing interest covered in the RQ2 reveals testing practices and problems for the actual testing of functionalities.

The leveraged techniques work either by performing testing at the *domain engineering*, or testing the concrete assets at *application engineering*. Domain engineering is by definition the SPL process where commonality and variability analysis take place, leading to the definition of reusable test assets, i.e., by representing the points in which assets will accommodate different properties, that will be further exploited during application engineering.

As software reuse plays a fundamental role in SPL engineering, we analyzed the selected primary studies in the light of a set of characteristics a technique should cover. Table 4.8 presents the characteristics, and Table 4.9 sketches a relationship between the selected studies and the addressed characteristics.

Regarding the characteristic *variability*, our interest is to understand which, and how, the approaches define test cases (TC) and test scenarios (TS) by expressing variability from domain models in such artifacts. We hypothesize that, for each feature, there should be test cases present in the test suite to validate whether the feature has been correctly implemented. We identified 12 out of 27 studies (44%) proposing strategies that use the variability models to define reusable test assets.

Table 4.8: Leveraged SPL testing characteristics

Characteristic	Rationale
Variability	SPL test assets are explicitly designed and modeled making their variation points explicit.
Test reuse	Test assets from domain engineering may be systematically reused in application engineering, as assets from a product instance may serve as input for a next instance.
Automation	Employing an automated strategy for generating SPL test assets leads to significant effort reduction.
SPL process	Tests can be performed in domain engineering, application engineering, or both.

In the following characteristic, *asset reuse*, the focus was to understand whether the study explicitly provided a technique to reuse test cases (TC), test scenarios (TS), test results (TR), and test data (TD), either between products, or from a core asset base. This was found to be the most common characteristic within the selected studies. The amount of 23 out of 27 studies (85%) made any contribution to this group of characteristics, with a larger amount dedicated to establish strategies to handle TC and TS reuse.

There is an initiative to improve test asset reuse by focusing on the differences between product instances, in a so-called delta-oriented testing technique, based upon regression testing principles and delta modeling concepts. The idea behind the technique is that an SPL can be represented by a core module and a set of delta modules. While the core modules will provide an implementation of a valid product, the delta modules specify changes to be applied to the core module to implement further products, by adding, modifying and removing code (Schaefer *et al.*, 2010). In this effect, test models for a new product instance will be generated by considering the deltas between this and the preceding product instances. That is, testing will focus on the deltas, what enables an increased reuse of test assets and test results between products. The tecnhique was investigated in both P41 and P48.

Next, we analyzed how the studies addressed the automated generation of test cases (TC), test case selection (TCS), and test inputs (TI). A small number of studies provided any description on how they could automate such important tasks so as to make SPL testing a feasible practical approach. Only 5 out of 27 studies (19%) explicitly provide tool support to handle the listed characteristics, as summarized in Table 4.9. They are: P01, P11, P27, P42, P49. In all of them, the studies only state they developed a tool, but they do not make their tool available. Another 5 studies - P03, P12, P14, P17, P22 - point out the need of tool support to handle those characteristics. However, instead of proposing new tools, they use already established ones.

In another 7 studies (26%) - P06, P08, P32, P35, P39, P41, P45 - authors state their proposed

Table 4.9: Relationship between selected studies and characteristics

Study	Variability		Asset reuse				Automated gen.			Process		
	TC	TS	TC	TS	TR	TD	TC	TCS	TI	DE	AE	Both
P01	·	·	●	·	·	·	●	·	·	·	·	●
P02	●	·	●	·	·	·	·	·	·	●	·	·
P03	⊖	●	·	●	·	·	⊖	·	·	·	·	●
P04	●	·	●	·	·	·	·	·	·	·	·	●
P06	·	·	●	·	·	·	·	·	·	·	●	·
P07	●	·	●	·	·	·	·	·	·	·	·	●
P08	·	·	·	·	●	·	·	·	·	·	●	·
P09	·	●	·	·	·	·	·	·	·	·	·	●
P10	·	·	●	·	·	·	·	·	·	●	·	·
P11	·	·	●	·	·	·	●	·	●	·	●	·
P12	·	·	●	·	·	·	⊖	·	·	·	●	·
P13	·	●	·	●	·	·	·	·	·	●	·	·
P14	·	●	·	●	·	·	⊖	⊖	·	●	·	·
P16	·	·	·	●	·	·	·	·	·	●	·	·
P17	·	●	⊖	●	⊖	·	⊖	·	·	·	·	●
P22	●	·	·	·	·	·	⊖	·	·	●	·	·
P27	·	●	·	●	·	·	·	·	●	●	·	·
P32	●	·	·	·	·	·	·	·	·	●	·	·
P34	●	·	●	·	⊖	·	·	·	·	·	·	●
P35	·	·	·	·	●	●	·	·	·	·	·	●
P39	·	·	●	·	·	·	·	·	·	·	●	·
P41	·	·	●	·	●	·	·	·	·	·	●	·
P42	·	·	·	·	·	·	·	●	·	●	·	·
P45	·	·	·	·	●	●	·	·	·	·	●	·
P46	·	·	·	·	●	●	·	·	●	●	·	·
P48	·	·	●	·	●	·	·	·	·	·	●	·
P49	·	·	●	·	·	·	·	●	·	·	●	·

Legend: [●] Characteristic clearly addressed by the study, [⊖] The study encourages the use of such characteristic, but do not provide any implementation (e.g., it states an external tool is used, but no detail is provided), [·] Characteristic not mentioned in the study, [TC] Test case, [TI] Test input, [TD] Test data, [TCS] Test case selection, [TR] Test result, [DE] Domain Engineering, [AE] Application Engineering.

approaches are supported by automated tools. However, they do not provide readers with detailed information about the tools, neither make clear whether the tool was built for the particular purpose of the investigation.

The last characteristic observed in which SPL process the approaches fit, *domain engineering*, *application engineering*, or both. Testing the common aspects early in domain engineering is essential, since an undetected error in the "framework" assets can be spread to all instances depending on those assets. However, as it is unfeasible to test all possible input values for the domain assets, since different variants might depend on specific applications, testing must also be performed in application engineering. Besides testing input data not covered yet in domain engineering, even the common assets previously tested might not work properly when bound in a specific product instance. Thus, some of domain assets should be retested again in application engineering, considering the particular behaviour of a product instance. In this effect, we found 10 approaches working at *domain engineering*, while 9 approaches handle testing at *application engineering*. Finally, we found 8 approaches that can be applied in both SPL processes.

4.3.4 Strength of evidence in support of available strategies (RQ3)

According to the evidence evaluation scheme described in Section 4.2.3, Table 4.10 presents detailed results on how much evidence is available to adopt the proposed approaches. Figure 4.5 provide a summarized data representation of the results.

Data showed that all selected studies present any kind of preliminary evaluation. However, this apparent benefit is diminished when we consider the low level of evidence of the proposed approaches. The most commonly employed evaluation methods are *academic studies* (Lev4) - 53% out of the total, followed by *demonstration* (Lev2) - 33%. Only a very small number of the approaches claimed to be evaluated in industry - 14%, what lead us to consider an overall low-level of evidence in the SPL testing field.

The scenario is even more worrying as data from the SPL testing interests are considered in a separate way. Figures 4.6 and 4.7 show data from interests 1 and 2, respectively. Only one study from the first interest set was evaluated in industry settings. We found no studies with both academic and industrial evidence.

It is worth mentioning that we found no studies applying more than one evaluation method. The combination of empirical evaluation methods to assess the feasibility of an approach is desirable because it increases external validity for findings.

Table 4.10: Evidence level of selected studies.

Study	Lev1	Lev2	Lev3	Lev4	Lev5	Lev6
P01				●		
P02		●				
P03		●				
P04						●
P05				●		
P06		●				
P07		●				
P08		●				
P09						●
P10				●		
P11				●		
P12		●				
P13				●		
P14		●				
P15		●				
P16					●	
P17						●
P18		●				
P19				●		
P20		●				
P21		●				
P22		●				
P23				●		
P24				●		
P25				●		
P26		●				
P27		●				
P28				●		
P29				●		
P30				●		
P31				●		
P32		●				
P33				●		
P34				●		
P35				●		
P36				●		
P37				●		
P38		●				
P39				●		
P40				●		
P41				●		
P42				●		
P43				●		
P44					●	
P45				●		
P46				●		
P47				●		
P48					●	
P49					●	

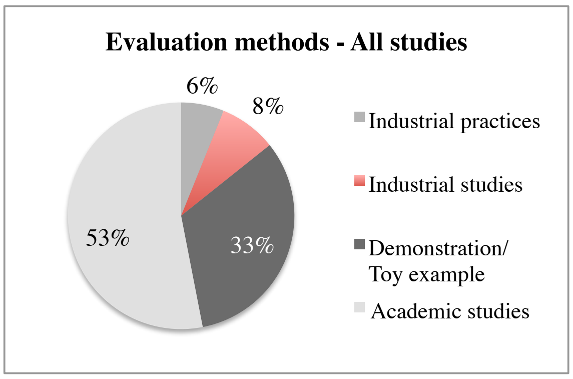


Figure 4.5: Evidence available to adopt the proposed methods - All studies.

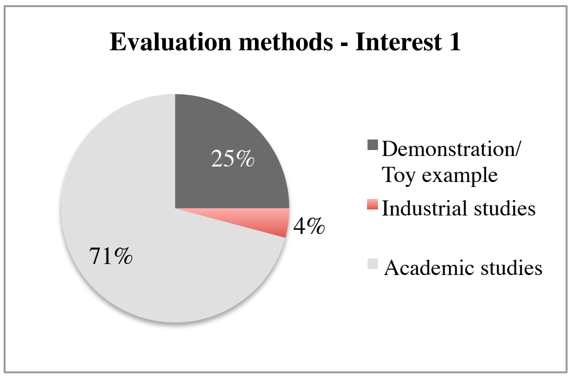


Figure 4.6: Evidence available to adopt the proposed methods - Interest 1.

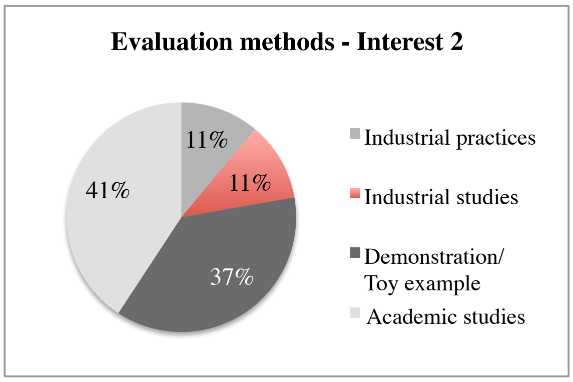


Figure 4.7: Evidence available to adopt the proposed methods - Interest 2.

4.3.5 Implications for research and practice (RQ4)

To determine the implications of the approaches found in the literature for both research and practice, we have to initially figure out the limitations of selected studies. We used the quality criteria established for appraising the selected studies, presented in Table 4.4, to determine the strength of inferences. The criteria were categorized in four groups of issues: reporting (QC1-3), rigour (QC4-7), credibility (QC8-9), and relevance (QC10-11).

Table A.2 lists the quality assessment results for all selected studies. Most of the selected studies perform fairly well on *reporting* issues. We recall that, as the first criterion (QC1) was used as the basis for either including or excluding a study, meaning that all selected studies are based on research and not merely a "lessons learned" report based on expert opinion. The same positive result could be observed in the QC2, while roughly 98% of the selected studies clearly state the aims of the research. While excelent results were obtained in the first two criteria, only 71% of the studies provided an adequate description of the context in which the research was carried out. That is, authors provide readers with limited information on the application domain(s) in which the approach was used, or the software process(es) involved, or even the skills of involved engineers, necessary to seamlessly use the proposed approach etc.

Four criteria relate to the *rigour* of the research methods employed in the primary studies. In roughly 49% of them, the research design is appropriate to address the aims of the research. In only 29% out of the total, there was a control group with which to compare treatments, so as to enable authors to compare the outcomes of the proposed approach against an external entity. In 43% of the studies, data was collected in a way that addressed the research issue, and in only 16% of the studies, data analysis was sufficiently rigorous. The results point out to a frustrating lack of rigour, as it is likely that some of the findings from the selected studies are probably accurate and usefully generalisable. However, the apparent shortcomings in methodology seriously limit their usefulness.

In relation to *credibility* of the study methods, for ensuring that the findings are valid and meaningful, we found a rather small amount of studies (12%), in which the relationship between researcher and participants was considered in the evaluation. However, for 65% of the studies the findings were explicitly stated, in which they discussed the results on the basis of the research questions, and an adequate discussion of the evidence, thus including the limitation of the approach, was provided.

In terms of *relevance* for both research and practice, we considered that 84% of the selected studies as valuable, for they describe the strengths and weaknesses of their proposal, and point out open rooms for improvement. As opposed to this high value, only 16% of the studies present any practitioner-based guideline, which seems to be a barrier to providing optimal usage of the proposed approach, especially for industry-scale application.

To understand the impact of the selected studies for the research field, we gathered their citation counts. A simple way to gather the citations is using the Google Scholar Citations[8]. While some academics have been very critical of Google Scholar (Aguillo, 2012), as it may lack the quality control needed for its use as a bibliometric tool, other authors (Harzing, 2013) argue that its coverage is very comprehensive, and as such it might provide a less biased comparison across disciplines than tools such as Web of Science or Scopus. These present an insufficient coverage, as they barely include conferences and/or workshops, but are focused on journals instead. Besides, Google Scholar has displayed considerable stability over time, increasing the coverage for disciplines that have traditionally been poorly represented.

Figures 4.8 and 4.9 shows scatterplots for the citation counts[9] and quality scores, respectively

[8]http://scholar.google.com

[9]The citation counts were gathered by the end paper selection process, late in the first semester of 2013.

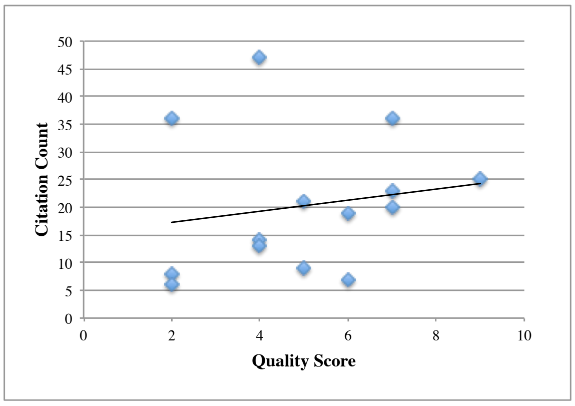


Figure 4.8: Studies published between 2003 and 2009.

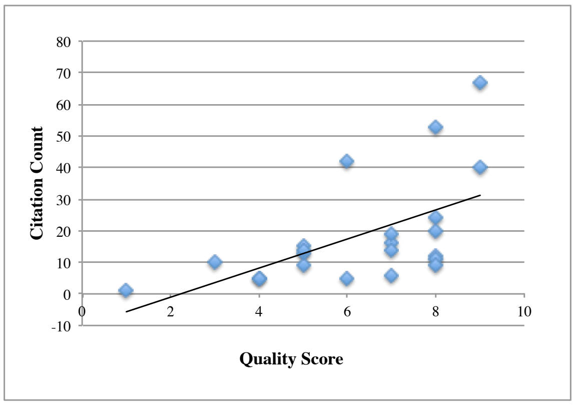


Figure 4.9: Studies published between 2010 and 2012.

for the set of included studies published between 2003 and 2009, and the studies published between 2010 and 2012. Such a division was arbitrarily established as a means to mitigate the likely *timing* effect in the analysis, i.e., as time passes, it is likely that more citations a paper will hold. Hence, it would not be fair to use observations from all studies in a same sample.

It is worth to mention that we excluded selected studies published in the year 2013. The reason is that it usually takes time for any automated citation engine to index all publications and the citations thereof. In a briefly performed search, we barely found a citation to any paper published in the year 2013 that was considered in this systematic review.

The y-coordinate corresponds to the number of citations of a paper. The x-coordinate corresponds to the quality score achieved in our assessment. The point representing the observation for a given paper is placed at the intersection of the two coordinates. In the first set we list 14 observations (we found no results for 4 studies). In the latter, we list 23 observations.

In both sets we observe an uphill linear pattern, as we move from left to right. It is a brief indication of a positive relationship between the citation count and the quality score attributed to the selected studies, especially in the second set. However, it is hard to infer whether any cause-and-effect relationship exists, due to the rather limited number of observations.

4.4 Analysis and discussion

This systematic review identified a set of testing approaches that are relevant for the SPL engineering field. The main observation in this study is the emerging need of supporting future development of an evidence-based and effective testing pratice designed to address the special SPL engineering quality needs, thereby improving the overall SPL practice. Along with such an observation, we describe throughout this section other important aspects that may arise from this research.

This review has to do with software testing in SPL engineering. We should recall that testing is a validation technique employed to measure the product correctness and efficiency, identifying early in the development life cycle points in the program that need fixes, before delivering it to the final customer. An ideal purpose for SPL testing is to evaluate the correctness of features' assembly by gathering information on the sources of faults and statistics on how many errors are generated with certain volumes.

Much of the work on reducing test effort for SPLs has been performed from the *feature-based* (first interest) or *product-based* (second interest) perspectives. An overall impression we had is that, while approaches in both interests are concerned about minimizing the test effort, there is little discussion on the subject of achieving higher defect detection rates. There is rather limited evidence describing which are the commonly found faults in either interest. As each interest has a specific goal in the SPL testing task, knowing common faults can be helpful. It might guide an engineer to better identify the faults that are more likely to occur, and provide him/her with recommendations for the correct repair actions.

As our work attempts to provide research directions to SPL testing engineers to design and develop more effective SPL testing approaches, besides the previously stated observation, we next discuss the main open issues we found under each of the interests.

- *First SPL testing interest*

Within the challenges in this SPL testing interest, the main concern is in how to establish a strategy to test all the feature interactions that result in testing of all variations across all dimensions of variation. As pointed out in P46, current techniques handling the combinatorial problem can be categorized as *sampling* and *exhaustive* exploration techniques. While the former emphasizes the selection of configurations, likewise in pairwise coverage, to reduce the combinatorial space involved, the latter intends to consider all possible configurations, with the aid of specialized techniques to eliminate - or even to minimize - redundant configurations that do not need to be explored.

Within the selected studies, that fit into either one or another category, we observed the use of diverse optimization algorithms. Although the proposed solutions are usually claimed to achieve better levels of effectiveness than others, it is usually hard to figure out which can be suitable for an average scenario so as to enable generalizations. Despite the large amount of algorithms, we could not identify which could yield better results for a SPL project, due to the lack of reliable comparative assessment.

Another important highlight is that comparative studies of existing approaches are barely published. Perrouin *et al.* (2011) empirically analyzed the capabilities of studies P29 and P31, in an investigation that can be considered one of the few and relevant reports providing comparisons between SPL testing techniques. Such an endeavor in highlighting the strengths and weaknesses of their approaches could be followed by other researchers, as a means to make practitioners aware of which testing approaches could be feasible to a given application domain and/or scenario.

A task strongly related to the automated generation of product configurations involves determining the *quality* of the product line variability models. For instance, in a feature model, it is necessary to analyze the relationships among the features that determine the composition rules and the cross-tree constraints, and some additional information, such as trade-offs, rationale, and justifications for feature selection (Thörn, 2007; Pohl *et al.*, 2011). Such analysis determines which configurations are feasible in practice.

SPL modeling approaches seek to better and more effective strategies to ensuring consistency of the variability models. This verification effort concerns the technical quality of the variability structure, such as checking that two features that are mandatory are not also mutually exclusive. This type of check usually employs some forms of logic or rule checker on the constructs provided by the methodology.

Figure 4.1, earlier presented in Section 4.1, illustrates the role of the analysis of variability models in the first SPL testing interest. The *analysis* layer represents an input activity to the selection of product configurations to test. That is, before selecting the test cases - the valid and representative products - the SPL testing approaches should perform a series of analysis operations on feature models, such as finding if a product is valid, obtaining all products, calculating commonality and variability, detecting dead and false optional features, etc. (Rincón *et al.*, 2014), through the extraction and understanding of the feature model semantics, as earlier introduced in Chapter 3 (c.f. Section 3.2.1).

Despite the importance of model verification for the SPL testing field, the set of primary studies analyzed in this review, that discuss fundamental and practical aspects of testing in SPL engineering, do not provide details about how they perform the model verification task. They usually use the operations on feature models aimed at generating sets of valid products from them, instead of using the ones aimed at measuring the quality of the models. Therefore, model verification aspects are not subject to discussion in this systematic review. Salinesi and Mazo (2012) surveyed the literature on verification of product line models, and as such their work can be used as a means to understand the gaps and challenges ahead.

- *Second SPL testing interest*

Most of the approaches dealing with specification of variability in the test assets work by annotating variability in test models. UML activity and sequence diagrams have been used as the standard models. The variable scenarios include specifications to the whole set of admissible products, plus other test cases which instead will vary for each specific product, depending on how the variant characteristics are instantiated. Indeed, not all admissible test cases are derived, but rather they derive the SPL test specification and leave it unfolded. The test cases will actually be derived for a specific product after having instantiated the tags in each SPL use case to the appropriate values.

While those approaches seems to work for any SPL project, improving testing practice, no discussions on the maintenance effort for those variable test assets is provided. A variable test suite is typically developed to test the whole SPL and the test suite will be modified as new products come into play, or the current products need to be improved. However, as the number of products increases, the number of test cases for testing the SPL will also increase. Therefore, it becomes practically impossible to execute all the test cases of the product line due to limited available time and resources for each new product. Therefore, it is essential to seek a solution to minimize test suites for a specific product efficiently before execution to reduce the cost of testing. Besides, an efficient testing process should systematically exploit the reusability of test artifacts between the products under test.

Besides, we observed a strong concern about handling test design at a very high level of abstraction, regardless the importance of also coping with variability at lower levels, such as at source code. While the existing approaches can be applied to any SPL project, with only minor adjustments, satisfying system testing of end-product functionalities, no studies could be found that consider the likely particularities of unit testing, performed in conjunction (either before, or right after) with the actual implementation of features. We have no indications about the impact of employing different mechanisms to implement variability in the domain assets, and its consequence for the unit testing, the lowest abstraction level for testing. This observation proceeds from the fact that testing techniques relevant to single-system engineering might not deal with the variability intrinsic to the SPL domain. Thus, a deeper evidence-based discussion on such an issue would be valuable.

Another important aspect to consider is the potential an SPL testing strategy has to cope with traceability issues. In a SPL, establishing traceability between feature models to actual implementation artifacts is a key task to a number of tasks, such as program comprehension, maintenance, requirement tracing, impact analysis, and reuse opportunities (Antoniol *et al.*, 2002). From a SPL testing perspective, the links between problem space and solution space entities enable both the definition of the test assets, and the capability to cope with evolution, in a sense that it is possible to identify the work products affected by a proposed change.

The selected primary studies address part of the aforementioned tasks. The use of annotated UML models (activity and sequence diagrams), in which features are associated to stereotypes, makes it possible to establish the relationship between a given feature and its corresponding test models and implementation - P07, P09, P11, P12, P13, P14, P17, P34, P35. It is also possible to use traceability-mining algorithms (Linsbauer *et al.*, 2013) to recover the tracing information, even though no primary study discussed about this feasible opportunity.

Traceability information between features and test source code should also be established. For each product, it is important to know the functionalities it provides and hence the features that it contains and its source code (Machado *et al.*, 2014a). However, such low-level traceability information is barely mentioned in the analyzed primary studies. The studies P45 and P46 propose a means to handle low-level variability in test cases. Variability is implemented as simply Java Boolean variables, i.e., an SPL is an ordinary program with if-conditions. This known limitation prevents us to generalize such a technique to scenarios in which other variability implementation mechanisms are employed.

Furthermore, in practice, as systems evolve, traceability links between the models and the test code artifacts may become broken or outdated. Thus, it is necessary to keep the consistency between the artifacts. The study P22 contributes an approach to provide traceability links in a way that ensures consistency between the feature model and the code artifacts, enables the evolution of variability in the feature model. The approach provides information on the artifacts that are actually impacted by a change, and provides immediate feedback on the actual impact of that change, reflecting on the tracing between feature and code assets. In addition, in P49, test cases are augmented with an attribute to associate it to the feature, or to a set of features. This is a promising strategy that makes it possible to keep the tracing between both entities updated.

As a matter of fact, SPL testing strategies still have lots to do in terms of traceability and evolution. Currently, external tools handle most traceability and evolution aspects, as mentioned in P09. Just a few research efforts could be retrieved that incorporate those aspects into their proposals.

Another real concern in this SPL interest refers to the strength of evidence in the selected studies, as earlier discussed in RQ3 and reinforced hereinafter. There is ample evidence that many SPL testing approaches are methodologically weak. Despite the amount of approaches, just a few of them have benchmarked their results against other approaches. It may hinder any inference on highlighting the benefits of selecting and/or using an approach over another. Good research requires not only a result, but also clear and convincing evidence that the result is sound. This evidence should be based on experience or systematic analysis, not simply persuasive argument or small examples (Shaw, 2002).

Empirical studies comparing the effectiveness of existing approaches, in a range of scenarios, thus involving project of different sizes and domain, is encouraged. Furthermore, there are some obstacles in the current tool support that should be addressed before further empirical studies can be conducted. As mentioned earlier, although some studies attempt to describe their tool support, no tools are readily available. When some algorithms describing how the approach would work in an automated scenario, readers are not provided with any discussion on their real benefits, and the required effort for its implementation.

The observed lack of empirical studies does not allow to evaluate what are the best ways to test an SPL. This is an important research issue for the area.

Besides, no guideline or methodology is provided to train test engineers to handle test design and selection. This means the current practice of test selection largely depends on the expertise of test engineers and it might not scale when more components are developed and are to be tested.

This analysis has considered individual research reports, but major results that influence practice rely on accumulation of evidence from many projects. Each individual paper thus provides incremental knowledge, and collections of related research projects and reports provide both confirming and cumulative evidence.

4.4.1 Limitations of this study

Our systematic review has some limitations. To the extent that we performed a systematic literature review, the potential for incomplete identification of relevant studies and publication bias are always consideration.

A potential risk that we might have missed relevant papers is due to lack of agreed terminology in the SPL field, leading to the possible existence of relevant papers that do not explicitly mention the keywords we specified. Hence, there is always a risk that important studies are omitted. To minimize this possibility, the search for potentially relevant studies encompassed a bibliographic search of published literature reviews on the topic under investigation, a search with multiple databases, and also bibliographic searches of the reviewed articles to identify additional studies. Thus, by combining the list of retrieved papers, we might assume a good coverage of publications and venues in the SPL testing field.

With respect to publication bias, the heterogeneity in studies' design, interventions, and outcome measures, and the absence of statistically reliable effects preclude any strong claims for the effectiveness of the analyzed approaches. In most cases, data collection and analysis were poorly described. Overall, empirical studies are not supported by rigorous evidence. Hence, due to there being insufficient data in the papers, necessary for the computation of effect sizes, we could not assess publication bias through meta-analysis. However, we acknowledge that, given the goals of this study, meta-analysis would not have been appropriate.

The effect of publication bias might have affected the data extraction, in terms of inaccuracy and bias. We had some difficulties to extract relevant information from the selected papers. For instance, several papers do not explicitly mention in which domain the proposed approaches could be used, or there is a lack of information regarding the empirical methods employed by the studies to carry out their evaluation. In situations like these, we had to make subjective interpretations of information. Therefore, the researcher's bias could affect the final extracted data. Hence, we acknowledge that there is a possibility of having misunderstandings in the way we have extracted data from the primary studies. The data extraction form was designed to obtain consistent and relevant information for answering the research questions. Besides, we performed quality assessment on relevant studies to ensure that the identified findings and implications came from a credible basis.

We next discuss the potential threats to the validity of this systematic literature review, in accordance with the following taxonomies: *construct validity*, *internal validity*, *external validity*, and *reliability* (Wohlin *et al.*, 2012).

- *Construct validity* concerns establishing a relationship between the theory behind the study and the observations. It covers issues that are related to the design of the study, to analyze its ability to represent what reseachers have in mind, and what is investigated according to the research questions. To avoid threats to construct validity, we applied the concepts of "testing in SPL" and "systematic literature review" as the main constructs. In the former, we leveraged the key characteristics in the studies, following the division of interests.. As for the second construct, we followed the guidelines to design the research questions, search and assessment criteria. Another important aspect is to ensure the discovering of all relevant studies in the topic under investigation. For this purpose, we combined automated and manual searches, to increase the coverage.

- *Internal validity* concerns establishing a causal relationship, whereby certain conditions are shown to lead to other conditions. As threats to the internal validity we can consider the subjective decisions that might have occurred during primary studies selection and data extraction, and individual bias in assessment. The decision process for including a study into the analysis may be considered the most important influential factor on the resulting conclusions. Given that some primary studies did not provide a clear description or proper objectives and results, it was difficult the objective application of the eligibility criteria or the impartial data extraction. The strategy to minimize selection and extraction mistakes was to consider several stages in the review in order to incorporate the most complete primary studies possible to increase reliability of the conclusions. Besides, to minimize threats regarding data analysis, that arise from individual researchers bias when assessing her assigned primary studies, we followed a pre-defined protocol, carrying out several dry runs individually, and consolidating the differences collaboratively. However, it is possible that some papers that do not contribute to the understanding of the study goals are included, as it is possible that some excluded papers might present useful characteristics that might affect the review conclusions.

- *External validity* concerns establishing the extent to which results of the studies provide a correct basis for generalization to other scenarios and application domains. Most of the papers included in this investigation refers to approaches that have not been used in industry. Even most of the evaluation and assessment performed on the approaches do not refer to real-world practice. This prevents us from asserting that the classification provided in this analysis is fully generalizable.

- *Reliability* is concerned with issues that affect the ability to draw that the operations of a study can be repeated with the same results. To attain this goal, we defined the search strings and procedures in such a way that other researchers could directly and objectively replicate this study. However, we cannot guarantee that other researchers could achieve the exact same outcomes, as subjectivity is a major criticism levelled at primary studies analysis. However, we believe that the underlying trends should remain unchanged.

4.5 Related work

Early in 2003, Kolb and Muthig (2003) published one of the initial analysis of existing SPL testing practices, and pointed out a set of directions for further improvement. Although the study was not organized as a literature survey, the authors discussed about the use of techniques that were commonly used to test a SPL, and also highlighted the common problems of applying techniques from traditional software development in the SPL scenario. Despite the importance of such a study to the field, since its publication, much more investigation has been carried out.

One year later, Tevanlinna *et al.* (2004) published a survey on product family testing. In the light of available studies at the time of publication, the authors provided research community with an overview on the established practices and the challenges surrounding the SPL testing field. Authors focused on discussing methods developed for or that could be applied to test product families, while disregarding particular characteristics of a given method. The paper served for a long time as a good roadmap for researchers intended to investigate the field.

Johansen *et al.* (2011) presented a survey of product line testing, focusing on the investigation of strategies employed towards developing test suites for a SPL. They followed a formalized literature review process (Dybå *et al.*, 2005). After analyzing existing publications, the authors focused on the analysis of three studies that contained empirical evaluations on their data. The authors claimed that the reason to only include and analyze empirically assessed studies was to perform a more reliable assessment of SPL testing practices. However, the small amount of studies prevents the generalization of research findings.

There are some other related research we could include in this section, such as a series of literature reviews on the topic (Neto *et al.*, 2011a; Engström and Runeson, 2011; Lamancha *et al.*, 2009). Such studies provide state-of-the-art evidence on the SPL testing field. In a systematic way, these surveyed existing research trying to identify useful approaches, and synthesize the achievements, identify gaps, and propose research directions, based on studies published up to the year 2009. They complement each other well in terms of research questions addressed.

As a next step, in this present study, we go further, and analyze the current research on strategies for handling testing in SPL, by assessing provided evidence about current research regarding how far it can convince practitioners, and also try to identify open problems and areas for improvement.

4.6 Chapter summary

The goal of SPL engineering is to optimize effectiveness and efficiency by capitalizing on the commonality and managing the variation that exists between multiple software systems. In order to achieve the benefits of a software product line, testing of the assets that will compose the products is a critical activity. Given that an asset can be reused by multiple products, it is not economically meaningful to test every interaction between assets in every product instance derived from the product line. A new paradigm demands new strategies that result in new improvements.

This chapter reported on the results of a systematic literature review of testing in software product line engineering. It aims at understanding how products are selected from the very large set of possible products for asset testing, and how each selected product is tested. Twenty-four papers were found to provide the material for the discussion surrounding the first research question, while twenty-seven papers described strategies that matched the second.

Despite an observed increasing interest in the SPL testing topic, the study led us to claim the need for more effective methods and techniques for testing SPL, issue stated a decade ago by Kolb and Muthig (2003), that still holds true as probably the main rationale for current research. Given the current available evidence, we noted that research has advanced in terms of strategies to handle testing each selected product, but it is still in its initial stages when considering strategies to cope with the selection of representative products. Additionally, in either topic, we observed a lack of generalization of existing techniques, which demands further investigation.

Based on the findings of this systematic literature review, we suggest that research be undertaken to expand on the strategies that have been investigated. Further research into this topic should include some form of empirical assessment of existing strategies, as a means to improve their accuracy, and enable generalizations. This may entail investigating more than just small usage scenarios, but rather large-scale and industry-side scenarios.

This Chapter closes the second part of this thesis, which discussed the underlying concepts for this work. Next Part presents the thesis *concept*, encompassing the steps to build the proposed fault model support for variability testing.

5 A Preliminary Evaluation of the Effects of Unit Testing in SPL Engineering

In a previous investigation we defined a process for testing SPL projects (Machado *et al.*, 2011). The process establishes a systematic relationship between the common test levels and the SPL processes - domain and application engineering, exploring the usage of testing-related resources in the overall SPL life cycle.

By following the process, we carried out an empirical evaluation aiming to analyze and understand the capabilities and effects of the unit testing infrastructure, as well as capture lessons learned and practices that could be used by other SPL teams for their unit testing.

Unit testing is an important part of the development stage. As earlier discussed in this thesis, the earlier a defect is uncovered, the cheaper it may be to fix the problem. This statement continues to hold true in case of SPL engineering. However, there is little evidence available on carrying out empirical studies in SPL testing, which encompasses low-level testing.

Therefore, as the proposed fault model support aims at improving testing carried out at a low level of abstraction, we consider the empirical study reported along this chapter as an initial endeavor towards building the concept underlying this thesis. The collected insights are reported for further analysis and the gathered data will serve as baseline values for next evaluations.

Along this chapter we describe the empirical study, consisted of a controlled experiment, conducted in an academic environment with students from a V&V course as participants[1]. The study was carried out by following the guidelines for conducting experiments in software engineering defined in (Wohlin *et al.*, 2012), and reported as suggested by the guidelines described in (Jedlitschka *et al.*, 2008).

The remainder of this chapter is organized as follows. Section 5.1 briefly introduces the

[1]Portions of the material presented in this chapter have appeared previously in (Machado *et al.*, 2012a).

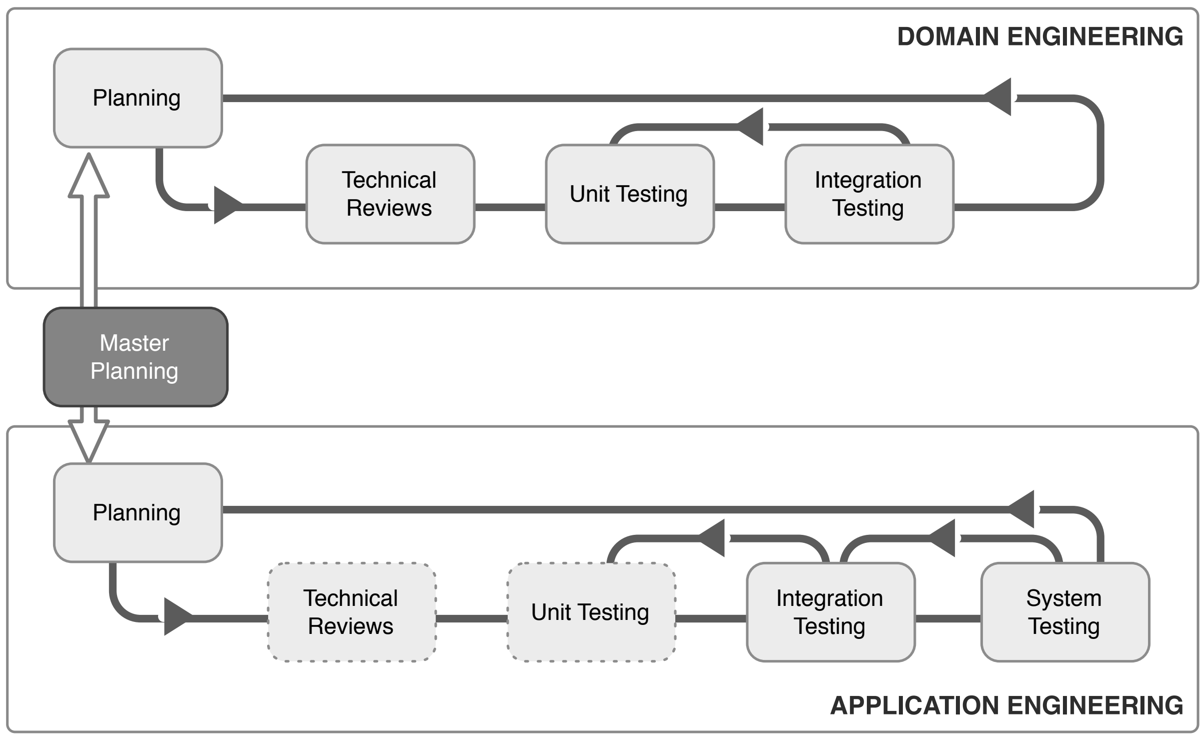


Figure 5.1: SPL testing process overview.

proposed approach for SPL testing, used in the experiment. Section 5.2 presents the experiment planning and definition, and Section 5.3 describes its operation. Next, Section 5.4 discusses the results. Given the insights from the experiment, Section 5.5 discusses the potential implications of following the proposed unit testing process, and Section 5.6 concludes the Chapter.

5.1 SPL testing process

Testing an SPL encompasses domain and application engineering phases. In a broad view, a process in an SPL must consider particularities of each phase, since they hold different, although complementary, goals.

Figure 5.1 presents an overview of the process subject to assessment in this empirical study (Machado *et al.*, 2011). The figure shows the test levels to handle in each SPL phase. Feedback is a closure activity common to all of these.

In domain engineering, when assets are developed with a special attention to the forthcoming reuse, the focus is on the *unit* and *integration* test levels. Thinking of Component-Based Development (CBD), firstly, in order to ensure that a component may be reused, it should be tested, under planned conditions. Therefore, planning should be mandatory for both domain and application engineering processes, as illustrated in the Figure 5.1. After performing unit tests in a component, and ensuring that it fulfills what it was specified to, *integration tests* are performed.

Recall that *unit* and *integration* testing are responsible for detecting different types of faults (Juzgado and Vegas, 2003). On one hand, *unit testing* independently tests methods, classes, and the interaction among these pieces that comprises a component. On the other hand, *integration testing* is responsible for validating the interaction among components interfaces and the integration between modules, that is, architectural information is a mandatory input for integration testing (Neto *et al.*, 2012).

A preceding box presented in the figure, named technical reviews, has also its very particular V&V goals. The process has included such an element seeking to increase the importance of statically verifying the entities, as a complementary V&V task. However, as there are several techniques available in the literature to cope with static analysis and verification, the process does not delve into sufficient detail to give exhaustive guidance for selecting an appropriate technique.

Taking the SPL testing interests earlier discussed in Chapter 4, and reported in (Machado *et al.*, 2012b), the proposed process fits into the second one, where the concern about variability surrounds how the test cases responsible for testing the end-product functionalities could be optimized. Thus, instead of considering a test case as a possible product instance, as the first interest does, the variability perspective employed in this process concerns on how to design and implement variable test cases that can be reused by a range of products, as they share a set of features.

Unit testing concentrates on testing the code from the inside, exercising the code logic. This activity is directly linked with coding, and as such, it should be held right after a piece of code is developed, or even before developing the source code, when employing a Test-Driven Development (TDD) approach. The strategy to use is a project decision. In either case unit testing serves to ensure that the unit (1) does everything that its specification claims, and that it (2) does not do anything it should not do. Carrying out unit testing may be particularly productive because the visibility into the code under test is at its maximum. The degree of visibility is related directly to the testability (the ease with which the code under test can be tested) (Clements and Northrop, 2001).

Unit testing continues to hold the same role in SPL as in traditional development. However, the variability contained in the units should be managed so that the effort devoted for testing a unit can be reused in next units, if representing characteristics that suffice. The reuse potential should be exploited in this early stage.

Unit testing comprises the following steps and associated tasks:

- **Planning** *(1) Analyze input artifacts* - before starting the tests, it is necessary to access the repository (if available) in order to identify if there is any unit test available that can be reused; *(2) Define and prepare the test environment*; *(3) Select features to be tested in each cycle*; *(4) Define the schedule, testing techniques, strategy to test the integration of classes, coverage criteria, test input domain, acceptance (Pass/Fail rate) criteria*; *(5) Summarize the information in a test plan*; and *(6) Review and approve the test plan.*

- **Design** *(1) Implement tests to evaluate the components methods. For mandatory features, the test case implementation is trivial, i.e., same technique from single system can be employed. However, whether a feature is either optional or alternative, the test case should implement the variable behavior, so as to enable its forthcoming reuse, when instantiating products with different feature selection*; *(2) Create tests to evaluate the classes integration*; *(3) Group test cases in test suites.*

- **Execution** *(1) Perform tests for each method*; *(2) Perform tests of the integration of methods contained in a class*; *(3) Perform tests of the integration of classes contained in a component*; *(4) Associate a Change Request (CR) to a defect found.*

- **Reporting** *(1) Assemble information on the cycle execution*; *(2) Record the issues found*; *(3) Check for test completion and provide feedback.*

Later in application engineering, *integration*, *system* and *acceptance* testing are performed, based on the assets previously developed and tested. *Unit* testing is performed in application engineering only in special cases, as explained next. It may be required whenever a new requirement or feature that does not belong to the core asset base yet is to be included in a specific product. That is, it involves the analysis, design and implementation of a new unit. Hence, as this new artifact is built, unit tests have to be performed. After that, the new *unit* might be propagated to the core asset base, to be reused in other product instances.

Whether there is no new unit to implement, but instead only existing features are to be assembled to compose a new product instance, *integration testing* is the first level to be performed. The point is why to perform it again. Given the purpose of avoiding repetition and consequently reducing the overall testing effort, in domain engineering only the integration between tightly coupled units is performed, regardless integrating the whole set of components. It considers the behavior of the core asset base, in which there are several components, attending to a diverse range of variations, that not necessarily integrate with each other, but indeed should be ready for instantiation in application engineering. This is when integration testing should take place in this SPL phase. Integration testing on the components of a product instance might ensure the workability of the interconnected modules as well.

Next, *system testing* is carried out. It focuses on validating the product instance as a whole, intended to detect system and/or end-to-end defects. At this point, testers evaluate the system against the requirements specification.

In all activities mentioned for both domain and application testing, each level encompasses four main tasks: *planning, design, execution* and *reporting*, to be iteratively and incrementally performed, with feasible feedback connections enabling refinements, as earlier introduced in section 5.1. Indeed, we can share the same tasks as in single system development. What mainly changes from one to another perspective is the capability of handling with variable assets, and the inherent systematic reuse of test assets. This is the main focus of every strategy for SPL testing.

5.2 Experiment planning

We applied the Goal Question Metric (GQM) method (Basili *et al.*, 1994) to set the measurement goal, define a set of questions, and finally create the corresponding metrics to perform the evaluation.

The **goal** of the controlled experiment was to analyze *the unit testing level* for the purpose of *evaluation* with respect to *its effectiveness* from the point of view *of the potential users (testers)* in the context *of a SPL testing project in an academic environment*.

The following **questions** were defined to evaluate the goal.

- **RQ1.** Does the *quality of the detected defect* improve when the process is followed?
- **RQ2.** Does the *rate of defect detection* increase when the process is followed?
- **RQ3.** Does the *test coverage rate* increase when the process is followed?
- **RQ4.** Which *professional skills* influence results of testing activity?

Three **metrics**, associated with the questions in order to answer them in a measurable way, were defined, as follows.

M1. Test Case Effectiveness (TCE): The more defects test cases find, the more effective they are (Chernak, 2001). It is defined as the ratio of defects found by test cases to the total number of defects reported during a test cycle. We tailored such measure to our context, so that (TCE) is defined as the ratio of the amount of defects (D_{tot}) reported to the total number of test cases (N_{tc}). This value provides insights on the effectiveness of functional test cases. It refers to both *RQ2* and *RQ4*, and is defined as: $TCE = \frac{D_{tot}}{N_{tc}}$.

M2. QDF: It refers to the number of valid defects found, normalized to **difficulty (DD)** and **severity (SV)** values. We characterized **severity** values as follows (Jones, 2010): *high* (software does not operate at all), *medium* (major features disabled or incorrect), and *low* (minor features disabled or incorrect, i.e., cosmetic defects that do not affect operation).

Difficulty values were also tabulated as *high*, *medium* and *low*, based on the expected amount of effort required to find the defects within the code. The experiment team classified difficult values themselves using consensus.

Every defect, in a set of known defects, are valued with a coefficient (k and r) according to its DD and SV. As we did not have baseline values to define the values for such coefficients, we performed Principal Component Analysis (PCA) correlations (Abdi and Williams, 2010), in order to identify their values. A principal component is a linear combination of weighted observed variables. By performing PCA, it is possible to calculate a score for each subject on a given principal component (Koch and Naito, 2010).

The quality of defects is then assumed as the total amount of defects considering each value category and their coefficients. The higher the score of the quality of defects found, the more effective testing was. It refers to both *RQ1* and *RQ4*, and is defined as: $QDF = f(DD) + f(SV)$. These are next detailed in formulas (1) and (2), respectively.

With the coefficients, it was possible to extract the *QDF* score. The formula is presented next (3). E_i corresponds to the amount of defects found by the subject i in a determined class - D_L and S_L for low, D_M and S_M for medium, and D_H and M_H for high, in respectively *difficulty* and *severity* classes.

$$f(DD) = \sum_{E_i \in D_L} E_i.k_l + \sum_{E_i \in D_M} E_i.k_m + \sum_{E_i \in D_H} E_i.k_h \quad (1)$$

$$f(SV) = \sum_{E_i \in S_L} E_i.r_l + \sum_{E_i \in S_M} E_i.r_m + \sum_{E_i \in S_H} E_i.r_h \quad (2)$$

$$S_{DS} = f(DD) + f(SV) \quad (3)$$

These formulas were created for the purpose of this experiment. We have no evidences about their effectiveness in other contexts. They need to be re-calibrated to other situations.

M3. Test Coverage (TCov): It gives the fraction of all features (or requirements/use cases) covered by a selected number of test cases or a complete test suite (Naik and Tripathy, 2008). We assume C_{ov} as the *basic blocks* coverage[2] generated by the Eclemma[3] code coverage tool, used in this study. It refers to *RQ3* and *RQ4*.

5.2.1 Design, variables, materials and participants

Design. This was an experiment with *one factor with two treatments*. We compared the two treatments against each other (Wohlin *et al.*, 2012). Factor in this experiment was the **SPL unit testing process**; the treatments were: (1) *Testing with the process.*; and (2) *Testing without the process*. We used a *completely randomized design* for comparing the two treatment means, i.e., for the design setup, we used the same objects for both treatments and assigns the subjects randomly to each treatment (Wohlin *et al.*, 2012). Hence, the participants were divided in two groups, each addressing a treatment, with respectively, 17 and 15 participants, each. Initially, we had 34 participants, but 2 of them left experiment during its execution, and their data were not considered. As we did not have the same number of subjects per treatment the design was unbalanced.

Variables. The dependent variables are the metrics TCE, QDF, and $TCov$. Independent variables are the background information items that compose the professional skills of the participants.

Experiment materials. The experiment used Consent Form, Background and Feedback Questionnaires, a set of Test Assets, Project Source Code, Defect Reporting Form, and the process documentation - guidelines and usage samples. After being informed about the goals and be given overall information on the experiment, all participants signed a consent form, as a means of agreeing to join the study, and filled out a background questionnaire, providing information about their expertise with software development in industrial projects, SPL, testing, and the tools used in the experiment. These are detailed in Section 5.3.

There were two main kinds of feedback questionnaire: (a) addressed to the group which did not follow the test process; (b) addressed to the group that applied the process. A third kind of questionnaire was designed to gather feedback from the participants which performed the experiment without the process. We aimed at gathering some information regarding the participants' opinion about the likelihood of finding more defects, if they had used the process

[2]`http://emma.sourceforge.net/faq.html#q.blockcoverage`
[3]`http://www.eclemma.org/`

Table 5.1: Hypothesis formulation.

Null Hypothesis	***Alternative Hypothesis***
$H_{01} : \mu_{TCE_{RP}} \leq \mu_{TCE_{AH}}$	$H_{11} : \mu_{TCE_{RP}} > \mu_{TCE_{AH}}$
$H_{02} : \mu_{QDF_{RP}} \leq \mu_{QDF_{AH}}$	$H_{12} : \mu_{QDF_{RP}} > \mu_{QDF_{AH}}$
$H_{03} : \mu_{TCov_{RP}} \leq \mu_{TCov_{AH}}$	$H_{13} : \mu_{TCov_{RP}} > \mu_{TCov_{AH}}$

instead of performing tests in an ad-hoc fashion. This last one was answered after the ad-hoc group was trained in process.

We used as project for this study an SPL project in the conference management systems domain. The project is named *X-Chair product line (XCPL)*, and it is targeted at the full management of papers submission in conferences, journals, etc. XCPL is an academic project, developed by some of the Ph.D. and M.Sc. students from our research group, thus including the author of this thesis. The project was conceived based on the commonality and variability analysis of a series of known conference management systems, such as: JEMS-SBC[4], EasyChair[5] and CyberChair[6]. As an SPL, it enables the derivation of different products from a common asset base. XCPL is composed of 41 features, identified in the domain analysis. The SPL was implemented with the J2EE platform, with variability in 8 core components.

Participants. The subjects were chosen based on convenience. They were students taking the "Systems Verification and Validation" course at Federal University of Bahia, Brazil. They were upper-level Computer Science majors and all were in good academic standing. All these students had previously attended mandatory courses such as OOP, Java, and Software Engineering-related courses as well.

5.2.2 Hypotheses

We set up three hypotheses that formed the basis for the design of this experimental study. Table 5.1 shows both the null (H_{0n}) and alternative (H_{1n}) hypotheses. The null hypothesis states that there is no benefit of using the process (*RP*) if compared to *ad-hoc* testing (*AH*), in terms of effectiveness.

[4]`http://jems.sbc.org.br`
[5]`http://easychair.org`
[6]`http://www.borbala.com/cyberchair/`

5.3 Experiment operation

Table 5.2: Experiment agenda.

Day	Tasks	Length
1	(A) Introduction to the experimental study	0:30
	(B) Characterization / Consent Term	0:30
	(C) Introduction to SPL	3:00
2	(D) JUnit training	4:00
3	(E) Ad-hoc Testing ***[Group 1]***	4:00
	(F) Feedback 1 ***[Group 1]***	0:30
4	(G) Training in the SPL testing process	3:00
5	(H) Testing with the process ***[Group 2]***	4:30
	(I) Feedback 2 ***[Group 2]***	0:30

Procedure. Table 5.2 shows the experiment design with training and execution schedule. Initially the participants became aware of the experiment purpose and associated tasks, as well as enrolled in an introductory session on SPL (Tasks A-C). Next, in order to balance the knowledge on the tools required, training sessions were performed with practical exercises (Task D).

Group 1 performed the tests without following the process (Task E), by using their own expertise. After this first testing session, the training in the SPL testing process was carried out (Task G). Later on, Group 2 performed the tests following the process (Task H). In this case, the participants were requested to perform test design, execution and reporting activities. Due to time constraints, planning artifacts were available beforehand. In the experiment, it was simulated that core assets had to be tested, in which participants should think of the forthcoming reuse of test assets. We assumed testing only the core assets as they hold most variability implementation. Due to time constraints, participants were not encouraged to test product-specific parts, as it might not be relevant for the purpose of our investigation.

Every participant reported their valuable feedback on the experiment by filling out a questionnaire (Tasks F and I).

Operation. All participants were asked to fill out a background questionnaire, as the initial task in the experiment. By gathering data about their preceding experience, we could perform some correlations between obtained results and experience. The questionnaire comprised the following items.

- *English reading* - Measures the participant's English reading expertise. A value in a three-level-scale, from basic to advanced, is used. As Portuguese is the main language in Brazil, our initial assumption was that difficulties with English language could affect the results, since all artifacts were designed in such a language;
- *Participation in Industrial Development/ Testing* - The answer was either *yes* or *no*, whether they had already participated in software development and/or testing projects in industry or not;
- *Experience in Programming/ Java/ Testing/ SPL/ jUnit* - Measures the level of participant experience with: programming, Java, Testing, SPL, jUnit;
- *Testing Tools* - They could mention unit test tools/frameworks in which they had some expertise, other than JUnit (e.g. NUnit, RSpec, Test::Unit, Selenium, etc). Our assumption is that, if participants were familiar with any unit testing framework, she could easier understand how jUnit works, and therefore achieve better results than ones who do not have any experience.

The participants were said not to implement new features, but rather to analyze the available components, design and implement the test cases. Their assigned tasks were: (1) to analyze the available code and specifications; (2) to build test assets (test cases and suites); (3) to execute them; and (4) the report the findings in the proper form. At the end of the experiment, each of the 32 participants completed a feedback questionnaire.

Deviations from the planning. Data was collected from all the participants. However, data from 2 participants were removed, since they either did not participate in all activities of the experiment, or they did not complete the forms as requested at the beginning of the experiment. Although we were counting on all participants, we believe that the absence of two of them does not invalidate the work, in terms of statistical analysis and interpretation of results.

5.4 Data analysis

The analysis was performed based on descriptive statistics, hypothesis testing using t-test and PCA (Abdi and Williams, 2010). PCA is appropriate when there are obtained measures on a number of observed variables and there is a wish to develop a smaller number of artificial variables that will account for most of the variance in the observed variables. It is basically a variable reduction procedure. PCA enabled us to statistically select valid parameters, in order to identify which variables impact on the results (Yamamoto *et al.*, 2007). The principal components may then be used as predictor or criterion variables in subsequent analyses (Koch and Naito, 2010). As we intended to extract score values from a data set, PCA was helpful.

5.4.1 Descriptive statistics

5.4.1.1 Test case effectiveness (M1)

In terms of **valid defects found**, in G1, the *mean value* was 6.188 with a *standard deviation (sd)* of 3.187, while in G2, the *mean value* was 3.857, with a *sd* of 3.505. False positives identified were considered invalid and were not included in the analysis. Regarding the amount of **designed test cases** by participants per group, in G1, the *mean* was 10.380 with a *sd* of 5.137; and 8.143 of *mean* with a *sd* of 3.670 in G2.

By applying the TCE formula, we obtained the following values. The *mean* in G1 was 0.725 with a *sd* of 0.540. In G2 the *mean* was 0.425 with a *sd* of 0.353. *Median* of G1 is slightly higher than in G2, respectively 0.631 and 0.431.

5.4.1.2 Quality of defects found (M2)

Data from all *defect reporting forms* were tabulated and analyzed. The goal was to extract similarities among the reported defects, since many of these expressed the same problems, but they were rather described in different ways. Like in TCE, false positives were not considered in the analysis.

We used a set of mutation operators to seed errors in the code, which represented five valid errors the participants could find. We initially seeded five errors. During the experiment execution, the participants reported an additional seven errors types. These twelve defects were classified according to *Difficulty* and *Severity*, as can be seen in table 5.6, which denotes ε_i as every valid defect.

Table 5.3: Amount of defects found in terms of difficulty and severity.

	Difficulty			**Severity**		
	Low	*Med*	*High*	*Low*	*Med*	*High*
Group 1	66	12	19	21	7	70
Group 2	40	6	7	7	2	44

We combined the three variables in each class (low, medium and high) according to the values from the data set (Table 5.3). Tables 5.4 and 5.5 shows the final results for both *difficulty* and *severity* calculations, respectively.

Table 5.4: Difficulty ratings. Correlations on the PCA of the high/medium/low variables.

PC	λ	*r*(%)	Difficulty		
			dif_low	*dif_med*	*dif_high*
1	2.193	47.9%	-0.91	-0.20	-0.35
2	1.869	34.8%	0.31	0.20	-0.93
3	1.319	17.3%	0.26	-0.96	-0.12

Table 5.5: Severity ratings. Correlations on the PCA of the high/medium/low variables.

PC	λ	*r*(%)	Severity		
			sev_low	*sev_med*	*sev_high*
1	2.622	75.6%	-0.13	-0.13	-0.98
2	1.361	20.4%	0.98	-0.13	-0.12
3	0.604	4.0%	0.12	0.98	-0.15

Table 5.6: Difficulty and severity of defects found.

Difficulty		
Low	***Med***	***High***
ε_1, ε_2, ε_3, ε_5, ε_6, ε_{11}, ε_{12}	ε_4, ε_7, ε_8	ε_9, ε_{10}
Coefficient (k)		
0.91	0.20	0.35

Severity		
Low	***Med***	***High***
ε_4, ε_7, ε_{10}	ε_{11}	ε_1, ε_2, ε_3, ε_5, ε_6, ε_8, ε_9, ε_{12}
Coefficient (r)		
0.13	0.13	0.98

When we consider the quality of valid defects, as a function of difficulty and severity, we have an indicative that participants from G1 had better results. The *mean* value for G1 was 8.868, with a *sd* of 3.896, whereas in G2 the *mean* was 6.048, with a *sd* of 5.302. *Median* values were 8.555 and 6.295, for G1 and G2, respectively.

The descriptive statistics for the QDF score are detailed in Table 5.7.

5.4.1.3 Test coverage (M3)

The *mean* value for G1 was 0.663, with a *sd* of 0.136. G2 achieved a mean value of 0.630, with a *sd* of 0.364. *Median* values were 0.644 and 0.745, for G1 and G2 respectively. By analyzing the the data, we can notice that participants from G2 had slightly better results. Table 5.8 presents the descriptive statistics for this measure.

Table 5.7: Descriptive statistics for QDF score.

	Group 1	*Group 2*
Min.	1.040	0.000
1st. Quartile	7.288	0.120
Median	8.555	6.295
Mean	8.868	6.048
3rd. Quartile	10.080	10.350
Max.	19.990	15.160
Sd.	3.896	5.302

Table 5.8: Descriptive statistics for TCov score.

	Group 1	*Group 2*
Min.	0.450	0.000
1st. Quartile	0.567	0.583
Median	0.644	0.745
Mean	0.663	0.630
3rd. Quartile	0.753	0.848
Max.	0.889	1.000
Sd.	0.136	0.364

5.4.2 Hypothesis testing

The first hypothesis regarding TCE is evaluated using a t-test (unpaired, two-tailed). Table 5.9 presents the results, in which a p-value higher than 0.05 was found, which indicates that the Null Hypothesis H_{01} cannot be rejected. Hence, according to the values extracted from this experiment, the group who did not follow the process had better results, regarding TCE.

The t-test was also applied to QDF. From the analysis, we can conclude that it was not possible to reject the *Null Hypothesis* H_{02}, since p-value was higher than 0.05, out of the confidence interval of 95%. Thus, there was no gain using the process, regarding the QDF metric.

T-test applied to Test Coverage ($TCov$) resulted in a very high p-value, showing that means are extremely different. As a consequence, we can not draw conclusions on such measure.

Table 5.9: Results from the t-test applied to TCE, QDF and $TCov$ measures.

Measures	df	$p-value$	$t-value$
TCE	28	0.08685	1.7746
QDF	28	0.1053	1.674
$TCov$	16.163	0.7483	0.3264

5.4.3 Exploring relationships among variables

With the expertise variables, we performed a set of correlations in order to understand how they would behave when inserting and/or removing a new variable.

The set of **independent variables**, extracted from the background questionnaire is as follows: *group, English skill (Eng1-advanced; Eng2-intermediate; Eng3-basic), length experience in programming, Java, jUnit, testing, participation in industrial testing projects (part_proj_ind_test), participation in industrial development projects (part_proj_ind_dev), coverage*;

Next, the **dependent variables**: *total amount of valid and invalid defects found, score of defects detection - both considering severity and difficulty measures.*

Variables handling **length experience** were correlated and found no significant results. We applied stepwise regression, in order to identify which elements could make influence on the experimental study. It would be extracted from a model by AIC in a stepwise algorithm.

In terms of **difficulty**, Table 5.10 shows the resultant model. According to the results, there is a strictly relationship among the results and the group the participant was. The results showed that participants from G1 had positive influence on the results, meanwhile participants from G2 had worse results *(p-value: 0.01352)*. Moreover, participants who results were better were directly influenced by their prior participation in industrial development projects. This is inferred by a high confidence level *(p-value: 0.03157)*. Another variable that does not influence the results is the English reading. Based on the data, it is inferred that basic English knowledge suffices to this kind of activity, although this evidence was not totally reliable *(p-value: 0.10907)*.

Table 5.10: Significance of the regression estimation parameter values.

Parameters	$Estimate$	$std.error$	$t-value$	$p-value$	
Intercept	0.6136	1.0088	0.608	0.54850	
Group	−1.6670	0.6273	−2.657	0.01352	$*$
coverage	3.7423	1.2318	3.038	0.00551	$**$
Part_proj_ind_dev	1.6706	0.7335	2.278	0.03157	$*$
Eng3	1.8701	1.1254	1.662	0.10907	

Signif. codes: 0 '$***$' 0.001 '$**$' 0.01 '$*$' 0.05 '.' 0.1 ' ' 1

R^2: 0.5028, Adjusted-R^2: 0.4233, F-statistic: 6.321, Sig F: 0.001172

Table 5.11 shows the most significant parameter values of the model extracted form the correlation among variables, in terms of **severity**. In this model, the inference aforementioned, regarding the English reading level required to perform test activities, herein is ensured by a high confidence level *(p-value: 0.0365)*.

Table 5.11: Significance of the regression estimation parameter values.

Parameters	$Estimate$	$std.error$	$t-value$	$p-value$	
Intercept	0.3795	1.2596	0.301	0.7657	
Group	−1.7230	0.7832	−2.200	0.0373	$*$
coverage	3.5370	1.5380	2.300	0.0301	$*$
Part_proj_ind_dev	2.3586	0.9158	2.575	0.0163	$*$
Eng3	3.1055	1.4052	2.210	0.0365	$*$

Signif. codes: 0 '$***$' 0.001 '$**$' 0.01 '$*$' 0.05 '.' 0.1 ' ' 1
R^2: 0.4578, Adjusted-R^2: 0.3711, F-statistic: 5.278, Sig F: 0.003193

5.4.4 Threats to validity

Threats to *internal validity* come from how experiments were carried out. We used a process for testing SPL projects, responsible for indicating which resources should be developed regarding each test level, and the extent to which unit testing should be organized, as an important task to improve the quality of the artifacts of an SPL project. Although the process was systematically followed, there might be some internal flaws that compromised the validity of the results. Furthermore, we could also observe a maturation effect, namely as the experiment was performed in a continued 4-hour period, it is possible that participants were affected negatively (boring/tiring) or positively (learning) during the experiment. Indeed, the scope was tied to the course schedule. It might have influenced the overall results.

A possible threat to the *construct validity* is the under representation of the construct used in the experiment. That is, although the chosen domain contains a large set of features, only a sample scenario was selected to the experiment. On the one hand, it could not be possible to have everything finished within the timetable. On the other hand, a larger SPL project could yield a more reliable data set to support the analysis.

Regarding the *conclusion validity*, a likely threat refers to the reliability of measures. As measures for SPL testing are not usually generalized, and when the study was performed, suitable measures to employ in this experiment were not found in the literature (Neto *et al.*, 2011a), we applied some measures from traditional development, and also counted on statistics to define coefficient values. We found PCA to be a suitable statistical technique aimed at analyzing the variables that might have affected the results.

In light of what appears to be required for accurate measurements in the SPL testing field, our recent investigation has pointed out some concerns about the use of commonly used measures (Machado *et al.*, 2014b). It is still hard to identify the measures the community use to show value and focus improvements to the field. It might be explained for the observed heterogeneity in studies' design, interventions, and outcome measures.

External validity concerns generalisation of the experiment result to other enviroments (Wohlin *et al.*, 2012). The largest threat to the external validity is the use of students as subjects. This might be not representative of the population. However, this threat was reduced by using fourth-year students which are mostly familiar with the topics under investigation. In addition, some training sessions on the topic were held, involving subjects in practical sessions, in which they could become familiar with the tools used as well as the purpose of product lines, and so on. Indeed, if subjects succeeded in using the proposed approach, it was not convincing that we could generalize its use to SPL testing practitioners at all. Another threat to the external validity is the effect of not having the experimental setting or material representative of, for example, industrial practice. The experiment was conducted on a defined time according to the schedule of the undergraduate course, which may have affected the overall results. The scope was tied to the course schedule in order to make its completion feasible. Thus, although a big domain involves the project in question, only a sample scenario was selected to this experiment.

5.5 Evaluation of results and implications

The results presented insights that enable us to consider that unit testing in SPL does not have impressive differences if compared to traditional development, since practitioners which did not have experience in SPL projects had slightly better results than others who followed a formalized process. The model extracted from the multivariate regression analysis, which correlated all dependent and independent variables used in the experiment, did not return satisfactory results, in a sense that, much was collected but a very small amount of variables impacted the results.

Regarding the better results of the group which did not run the process, the amount of formalism the process comprises might have influenced the results, since the participants had to deal with a step-by-step process to be followed *verbatim*, in a very short period of time, which might have influenced the learning effect. Instead of just conducting exploratory tests, which could return better results within a same period, they had to follow the guidelines. Perhaps, as participants gain confidence on the use of the process (but it obviously depend on time to try different situations), the results might become better. But it is solely an assumption that should be tested more and more, maybe applying in a larger context.

After concluding the experimental study, we have gathered useful information that can serve as a guide to future replications of the experiment following the structure herein presented in other SPL Testing projects. However, some important aspects should be considered, especially the ones seen as limitations in this initial experiment. The general impressions gathered from the experiment are following listed:

Training. Subjects reported the lack of expertise in the tools used in the experiment, especially JUnit Framework. It is very interesting to either have a sample of subjects who have a certain knowledge on the tools or conduct more training sessions before conducting the experiment. JUnit is indeed a complex framework for beginners acting as subjects in an experiment which directly involves unit testing in the Java platform.

Questionnaires. After concluding the experiment, we noticed that useful information was not collected, such as the subject's impression of using the process, or even the points missed by the approach, and so on. On the other hand, we have collected information that we did not analyze, such as subjects' satisfaction with the training sessions.

Project. We would select a project with more specification available in advance. Subjects with large experience in industry complained about the lack of documentation in order to help them to create the test assets. They asked for more test scenarios. Moreover, as we are dealing with a SPL project, we really need a project containing many variation points and variants, in order to analyze the impact of the process in such topic. In the project we used in this experimental study, although the portion of code we chose contained variabilities, just a few subjects reused the assets. Most subjects created everything from scratch than reused.

Measurement. We did not report on the reused artifacts since we did not establish a metric to assess that. There were no evidences in the literature that would help us in defining such metric. It is really necessary for next experiments. Furthermore, metrics such as DRE (Craig and Jaskiel, 2002), defect removal efficiency, a very applied measure, but that can only be applied in a whole project, to check its results in a valuable way, should be included when the whole test levels were considered. Moreover, metrics to evaluate the ease of use should also be collected.

5.6 Concluding Remarks

The empirical evaluation presented in this Chapter was an initial effort towards gathering evidence to support the ideas behind our fault model approach. Although employing a naive approach to simulate the presence of problems in the source code implementation, as a means to measure the effectiveness of the testing approach, the design applied in the experiment enabled the comparison of effectiveness results, and identify, by employing some statistical models, what experience profile could influence on the testing activity.

As a matter of fact, achieving better test results could not depend solely on the presence of a systematic methods to guide test assets management. However, the results obtained so far cannot be fully conclusive. Further studies are thus needed, either through replication of our experiment or similar studies in other environments, involving a larger set of participants, with different expertise, so that it will be possible to gather more empirical evidence to confirm or refute the stated hypotheses. Indeed any difference may only become significant with more participants.

Much of the experimental material and the study design can be reused in future studies. The experiment can be replicated and other experimental variations can be built on top of our experimental package. As discussed earlier this type of material is much needed to gather more empirical evidence in SPL research. The experiment defined measures and sketched models using multivariate regression analysis for SPL. The results will also help defining baseline values to support further investigations.

Next Chapter addresses the construction of the fault classification scheme for variability testing. It discusses the common distribution of faults in variability mechanisms, as a means to support the design of the fault modeling method.

6 Defining a Fault Classification Scheme Towards Variability Testing

Documenting and analyzing faults is a common practice in most software development organizations (Mellegard *et al.*, 2012). The use of historical data about commonly occurring fault types might allow testers to choose testing methods that reveal a greater proportion of the faults present in the software (Basili and Selby, 1987; Juzgado and Vegas, 2003; Nath *et al.*, 2012). In addition to choosing a suitable testing method, understanding the nature of faults may aid testers in focusing their effort in the most fault-prone elements in the software under test. It helps project management to decide which defects to correct first or at all.

There have been several approaches proposed on how to perform structured collection and analysis of fault information, encompassing *defect taxonomies*, *root cause analysis*, and *defect/fault classification schemes*, either restricted to a particular programming language, e.g., (Hayes, 1994), or general to any software development process, e.g., (Basili and Perricone, 1984; Chillarege *et al.*, 1992). Another set of studies have carried out empirical assessments on the basis of such schemes, and others as well, to identify and classify domain specific faults, e.g., (Barbey and Strohmeier, 1994; Chou *et al.*, 2001; Guo and Sampath, 2008; Palix *et al.*, 2011; Paul and Lau, 2012; Strecker and Memon, 2012).

Establishing a means to identify and classify the faults in a software is a key input for a fault modeling approach. Hence, along this Chapter we investigate the existing support for fault classifications, towards building general fault classification schemes to support variability testing.

We investigated existing classification schemes, to identify their characteristics, such as fault classes and associated faults. Next, to aid the collection of empirical knowledge, we gathered data from three open source projects, as a means to associate errors to the use of variability implementation mechanisms. The results of such investigations serves as the basis for our proposal.

The remainder of this Chapter is organized as follows. Section 6.1 presents and discusses the most commonly used fault classification schemes, available in the literature. Next, in Section 6.2 we discuss an empirical investigation of understanding the faults in variability mechanisms. This conclusion of such an investigation leads to the proposal. Section 6.3 concludes the Chapter.

6.1 Fault classification schemes

Efforts to analyze and categorize software faults have a long history, and a number of different general approaches have been tried. There are several fault categorization schemes available in the literature, either by providing generic/language-independent classifications, or attending a very specific demand, such as a specific technology or a given application domain. Alongside, there is a strong knowledge basis that aims at generalizing those to a large domain application, like faults in the C or Java language constructs.

Initial reports on classifying errors in software projects dates back to the 1980's, where Glass (1981) investigated what he coined as *persistent errors*, i.e., those which are not discovered until late in developement, and proposed a classification of them based on the analysis of two significant and mature software projects.

Later on, Basili and Perricone (1984) proposed some abstract, general-like, fault classifications, by analyzing the relationships between the frequency and distribution of errors uncovered during a software development project, under a variety of environmental factors. Table 6.1 summarizes the proposed classification.

As a general-like errors classification, it encompasses those that might be originated from several of the SDLC phases, ranging from the requirements specification up to maintenance. Therefore, while analyzing a software project in the light of such a classification, it is possible to draw inferences about, e.g., the most fault-prone sources of errors, or the most likely errors. Roughly speaking, fault data can be used to predict later fault and failure data.

From a similar but subtly different perspective, Chillarege *et al.* (1992) proposed the Orthogonal Defect Classification (ODC), a very famous and widely used process in both industry and research studies. Being both language and domain-independent, ODC aims at identifying the root cause of faults of a particular error type, in terms of the process life cycle phase where the fault was introduced, so that the development process can be improved to address such an identified problem.

Table 6.1: Classes of errors proposed by Basili and Perricone (1984).

- Requirements incorrect or misinterpreted.
- Functional specification incorrect or misinterpreted.
- Design error involving several components
  - Mistaken assumption about value or structure of data.
  - Mistake in control logic or computation of an expression.
- Error in design or implementation of single component.
  - Mistaken assumption about value or structure of data.
  - Mistake in control logic or computation of an expression.
- Misunderstanding of external environment.
- Error in the use of programming language/compiler.
- Clerical error*.
- Error due to previous miscorrection of an error.

* A clerical error stands for an usually minor, inadvertent negligence in computing a figure, or recording or copying a fact or statement.

Faults of a specific type may be due to some cause in the process, as such the fault types can be associated with the activities of different life cycle phases. This is the basic concept of orthogonality applied in the ODC process. In this effect, ODC basically categorizes a fault into classes that collectively point to the phase that needs attention. Table 6.2 summarizes the relationship between faults types and the life cycle phases. In the ODC process, the reason for such a small number of classes is that, having a small set to choose from makes classification easier and less error-prone (Chillarege *et al.*, 1992).

It is worth mentioning that, while a common software development process will include phases like specification, design, code, and testing, in practice each organization can tailor such a general process to fit their needs, thus including some variations. This explains why Chillarege *et al.* (1992) considered a larger number of phases.

The main three categories of research employed towards understanding the differences of defects and their nature, namely *defect taxonomies*, *root cause analysis*, and *fault classification schemes* are usually based on one such a classification, each providing an in-depth view on the

Table 6.2: Defect types and their description based on ODC.

Fault type	Process Associations (where fault should be searched for)	Description
Function	Design	Error that affects significant capability, interfaces (end-user, product, hardware), demanding a formal design change.
Interface	Low-Level Design (LLD)	Errors interacting with other components, modules, drivers etc., via macros, call statements, control blocks, or parameter lists.
Checking	LLD or code	Addressed program logic that has failed to properly validate data and values before using them.
Assignment	Code	Errors in the initialization of control blocks or datastructure.
Timing/serialisation	LLD	Errors concerning the management of shared and real-time resources.
Build/package/merge	Library code	Errors occurring due to mistakes in library systems, management of changes, or version control.
Documentation	Publications	Errors affecting both publications and maintenance notes.
Algorithm	LLD	Concerns to efficiency or correctness problems that may affect the task, and requires (re)implementing either an algorithm or data structure without requesting a design change.

elements of interest.

From the perspective of source code implementation interest, Basili and Selby (1987) later on adopted an abstract classification of errors classifying these in five categories, as Table 6.3 shows (Basili and Perricone, 1984). These categories basically represent all activities present in any module of code. In addition, the categories are partitioned in errors of *comission* and *omission*. The former are the result of including some incorrect executable statement or fact (*something incorrect*), whereas the latter refers to the result of neglecting to include some entity within a module (*something missing*).

Another general-like fault classification encompassing faults in source code is presented in Burnstein (2003). Table 6.4 lists the defect classes, and give some examples, as the author provides.

Table 6.3: Classification of errors from a source code perspective.

Class	Description
Initialization	Failure to (re)initialize a data structure properly upon a module's input or output.
Control structure	Errors causing an "incorrect path" in a module to be taken
Interface	Errors associated with external structures the module depends on.
Data	Errors that are a reulst of the incorrect use of a data structure.
Computation	Errors that cause a computation to wrongly evaluate a variable's value.

Table 6.4: Coding defect classes defined by Burnstein (2003).

Class	Description
Algorithmic/Processing	Adding levels of programming detail to design, code-related algorithmic and processing defects.
Control, logic, sequence	Defect in decision logic, branching, sequencing, or computational algorithm, as found in natural language specifications or in implementation language.
Data	Defect in data definition, initialization, mapping, access, or use.
Data flow	Certain reasonable operational sequences that data should flow through.
Interface	Defect in specification or implementation of an interface.
Typographical	These are principally syntax errors.

More recently, Seaman *et al.* (2008) reported a defect categorization for source code, that might also serve as the initial input for our investigation. They investigated a decade of varying historical datasets containing, among others, design and source code inspection defect data. Table 6.5 presents the categorization.

All these fault categorizations converge on intention as the determinant key of identifying non-conformities in software behavior. By analyzing each of them we may notice a kind of similar thought different authors have had, in different moments in time, and based on data from different projects. However, despite the observed similarity, none of which is accepted as a basic tool in software projects.

Two important aspects should be considered in order to make any of the classifications above discussed useful nowadays. Firstly, it is necessary to have in mind that, due to modern highly typed programming languages, with powerful IDEs to support source coding, it is likely that a reasonable amount of errors found previously might not be evidenced today any longer.

Table 6.5: Source code defect types defined by Seaman *et al.* (2008).

Defect Type	**Description**
Algorithm/method	An error in the sequence or set of steps used to solve a particular problem or computation, including mistakes in computations, incorrect implementation of algorithms, or calls to an inappropriate function for the algorithm being implemented.
Assignment/initialization	A variable or data item that is assigned a value incorrectly or is not initialized properly or where the initialization scenario is mishandled (e.g., incorrect publish or subscribe, incorrect opening of file, etc.)
Checking	Inadequate checking for potential error conditions, or an inappropriate response is specified for error conditions.
Data	error in specifying or manipulating data items, incorrectly defined data structure, pointer or memory allocation errors, or incorrect type conversions.
External interface	Errors in the user interface (including usability problems) or the interfaces with other systems.
Internal interface	Errors in the interfaces between system components, including mismatched calling sequences and incorrect opening, reading, writing or closing of files and databases.
Logic	Incorrect logical conditions on if, case, or loop blocks, including incorrect boundary conditions ("off by one" errors are an example) being applied, or incorrect expression (e.g., incorrect use of parentheses in a mathematical expression).
Non-functional defects	Includes non-compliance with standards, failure to meet non-functional requirements such as portability and performance constraints, and lack of clarity of the design or code to the reader both in the comments and the code itself.
Timing/optimization	Errors thal will cause timing (e.g., potential race conditions) or performance problems (e.g., unnecessarily slow implementation of an algorithm).
Other	Anything that does not fit any of the above categories that is logged during an inspection of a design artifact or source code.

For example, misplaced and unmatched brackets are considered one of the most common syntax errors, however, IDEs usually provide developers with several mechanisms for finding the matching brace for an `if / for / do-while / while / try-catch` code block, what may prevent her to make such a mistake.

Thinking of implementing a feature in an ordinary programming language, it is all about coding. To a certain extent, this is a correct statement. There is a piece of specification which needs to be implemented, so that a program can be generated as an output. Hence, all these fault categorizations, either process-oriented or source code-focused may be strongly relevant to SPL engineering, as all of these problems are also inherent when implementing both a feature and a combination of these. Nevertheless, this statement is partly true. In SPL engineering, features interact with each other, not only in terms of models (e.g., feature models) but also, and more importantly, in source code entities. Hence, a number of problems can either emerge or not when implementing variable features in an SPL project.

Hence, establishing a general understanding of the common faults in variablity implementation serves as the main input to our proposed fault modeling support. Next section reports on an empirical investigation aimed at gathering some evidence of defects found in variability implementation. It is worth mentioning that we were by no means intended to provide an exhaustive list of exactly what are the common defects found in variability implementation. Instead, we limit our observations to the scope under analysis. However, it might provide some insights for further investigations.

6.2 Faults in variability mechanisms

There have been some initiatives concerned about understanding the nature of errors/faults in variability implementation, mainly covering C preprocessor issues (Nie *et al.*, 2012; Medeiros *et al.*, 2013). Most research in the SPL engineering has been focused on the analysis of data from projects implemented in the C language, where preprocessors are inherently good at enabling product configuration in a straighforward manner.

However, by considering that SPL may also be implemented in other languages such as Java, we devoted some effort to exploring the variability mechanisms available to this programming language. Hence, our analysis encompasses Java-based variability implementation mechanisms. From the commonly used variability mechanisms, as listed in (Svahnberg *et al.*, 2005; Schnieders and Puhlmann, 2006), we selected five of them, as Table 6.6 lists, that are usually implemented as Java variability mechanisms. These are mechanisms we could, to a certain extent, observe when analyzing the source code of open source software systems, next addressed in this Section.

Table 6.6 lists the origin of some faults found in each selected mechanism, and indicates the binding time for each. We should have in mind that some of the listed faults can not be caught at compile-time, but at runtime instead. In this effect, when designing test suites it is important not to overlook the time to which faults should be searched for.

Table 6.6: Variability mechanisms in Java and likely source of errors

Variability mechanism	Source of errors	Binding time
Configuration files (parameterization)	• Setting of variant specific configuration file entry • Entries are not well-formed • Detecting dependencies between configuration parameters	Runtime
Dynamic class loading	• Variant-specific class loading • Variant-specific typecast statement • Variant-specific method call	Runtime
Interface implementation	• Variant-specific method body into method header	Compile-time
Polymorphism with subclasses	• Variant-specific subclass invocation • Global variables statement	Compile-time/ Runtime
Static libraries	• Integration of variant-specific libraries during compilation	Link-time

Such a synthesized data was gathered from both the literature and informal talks with experts in the field. Indeed, there is quite a few reports addressing typical variability implementation issues other than those related to feature modeling.

By analyzing the widely used fault classification schemes, discussed in the preceding Section, we employed the general categorization of faults defined by Burnstein (2003), to drive the construction of the fault models for variability. Table 6.7 shows the classification scheme we used in this investigation. The changes form the original scheme is that we merged the classes *data*, *data flow*, and *control flow*, as in practice it is too hard to verify whether a problem is more significant from either one of them (Thung *et al.*, 2012). Besides, we removed from the analysis the class *typographical*, given that the capability of the currently used IDEs prevent programmers from making simple syntax errors. In most IDEs, when a syntax error is identified, a warning message is usually displayed to the programmer. Examples of common syntax errors include, but are not limited to: [1]: *missing semicolon; undeclared variable name; undefined class name; unmatched parentheses; unterminated string constants; left-hand side of assignment does not contain a variable; value-returning method has no return statement; mistyping the name of a method when overriding; lines of code outside of methods; calling a method with the wrong arguments; local variable not initialized*. Special attention goes to error such as *comparison assignment*, e.g., using = when == is intended. If the value is not a Boolean, this will cause a syntax error. However, if it is not a Boolean, the compiler will not display any error message, for the value of the assignment is exactly what the compiler expects. This case is clearly a semantic error, that should be handled accordingly.

Such a classification scheme was designed based on the guidelines presented in the IEEE Standard 1044:2009 (IEEE, 2009), which deals with the classification of software anomalies. In the standard, an *anomaly* refers to both any condition that departs from the expected, and also may indicate an enhancement. The term was employed for reasons of semantics, as a general word that might represent the related words *error, fault, failure, incident, flaw, problem, gripe, glitch, defect* or *bug*, that, in essence, conveys a more neutral conotation. Some of these were defined earlier in Section 2.1.

Indeed, without clinging to semantic details, but the overall meaning of the word, and its applicability in such a standard, we believe the underlying concept match our context and needs, in terms of using a structured defect classification scheme. The standard offers a significant process (step-by-step) to recognize, investigate, establish an action plan to handle such, up to its disposition, in a form of a report.

6.2.1 Empirical study - Analysis of open source software systems

The SPL research community has far counted on the source code of open source projects, to carry out investigation handling variability and feature issues. A recent large-scale investigation (Liebig *et al.*, 2010) consisted of analyzing the variability in forty preprocessor-based SPL, around 30 million lines of C code. Authors analyzed the role of variability implementation in `cpp`[2], trying to understand the influence of variability and program size, as well as how `cpp` handle extensions, that enable variability-awareness in software systems development.

[1]Information on common Java syntax errors are easily found in community websites, such as `stackoverflow.com`. The literature is also another great source of such kind of issues.

[2]`cpp` is the macro preprocessor for the C and C++ programming languages.

Table 6.7: Coding defect classes

Coding Defect Classes	Description
Algorithmic and processing	Adding levels of programming detail to design, code-related algorithmic and processing defects.
Control, logic, sequence	Defect in decision logic, branching, sequencing, or computational algorithm, as found in natural language specifications or in implementation language.
Data	Defect in data definition, initialization, mapping, access, or use.
Interface	Defect in specification or implementation of an interface.

We could take the set of systems reported in such a paper (Liebig *et al.*, 2010). However, as the authors only considered a single variability mechanism (conditional compilation), in a single language (C), and the selected systems do not comprise access to their bug tracking systems, so that we could track the reported bugs, we could not count on those software systems to the purpose of our investigation.

Thus, the challenge was mainly about selecting some candidate projects, from which we could identify the employed variability mechanisms. The task was to identify and classify the faults which originate from the variability they implement.

6.2.1.1 Procedure

This activity consisted of the following steps:

1. *Verify the reported issues.* By analyzing the bug tracking systems of the selected open source software, we could sketch a relationship between the reported entry (whether a improvement, a defect, or a patch), and the source file. We only consider the *defects*-related entries, and discarded the remainder. Besides, we analyzed the defects, attempting to associate them with the existing classification.

2. *Map defects/source files with variation points.* As we leveraged the defects, we could analyze their source files, in order to check whether they represented either a variation point concern or not. Indeed, this work is not concerned about locating features in source code (Dit *et al.*, 2011), but instead, from the reported bugs, we attempted to identify whether the associated source files represent a variation point.

3. *Select and categorize bugs.* As we identified the association variation point *vs.* bugs, we

Table 6.8: List of open source software systems.

Project	Version	Domain	LOC	Classes	Packages
`Apache Ant`	1.8.4	Development toolkit	178,674	1,633	80
`Apache JMeter`	2.7	Measurement tool	110,031	1,159	124
`Apache Log4J`	1.2.17	Logging library for Java	37,891	413	37

could leverage the mechanism employed in that implementation. Therefore, we could have an indication that the identified problem was associated to a given variability mechanism.

6.2.1.2 Datasets and empirical study settings

We aimed at identifying and classifying the faults which originate from the variability they implement. We hypothesized that it was possible to define a fault dictionary by transposing the effects of faults affecting the variability implementation.

We analyzed defects from three software systems:

- *Apache Ant*[3] is a Java library and command-line tool aimed at driving processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications.
- *Apache JMeter*[4] is an application designed to load test functional behavior and measure performance. It may be used to test performance both on static and dynamic resources. It can also be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types.
- *Apache log4j*[5] is a logging library for Java.

These are open source systems, mostly implemented in Java[6]. Table 6.8 provides some measures for each software system[7].

[3]http://ant.apache.org/

[4]http://jmeter.apache.org/

[5]http://logging.apache.org/log4j/docs/index.html

[6]The source code of most open source software systems contains source code in more than one language, e.g., Apache Ant project contains 11 languages, from which about 70% was implemented in Java. Source: `http://www.ohloh.net/p/ant/analyses/latest/languages_summary`.

[7]All metrics showed in Table 6.8 were collected with the *Eclipse Metrics plugin*. Available at `http://eclipse-metrics.sourceforge.net`.

All three projects use bugzilla as their bug tracking system. Every issue reported in bugzilla is associated to a severity level (*blocker, critical, regression, major, normal, minor, trivial*, and *enhancement*). Table 6.9 summarizes the number of valid issues from each repository per severity level. The values include all entries within the status set: *NEW, ASSIGNED, REOPENED, NEEDINFO, RESOLVED, VERIFIED, CLOSED*. Issues with *UNCONFIRMED* status were discarded.

Table 6.9: Number of issues reported per severity.

Project	**Severity**							
	blocker	critical	regression	major	normal	minor	trivial	enhancement
`Apache Ant`	188	203	66	521	2646	455	49	1706
`Apache JMeter`	53	60	30	247	1156	216	41	850
`Apache Log4J`	54	77	8	135	754	98	20	219

6.2.1.3 Study operation

We collected random bugs from the bugzilla repositories of the respective software systems: 500 from Apache Ant (from a total of 5834), 200 from Apache JMeter (from a total of 2653), and 100 from Apache Log4J (from a total of 1365). Thus, in total we have 800 randomly selected defects. As earlier mentioned, as we discarded any enhancement labeled entry, all 800 entries we selected are actual defects reported in bugzilla.

We manually carried out the categorization of issues, by associating them to the coding defect classes presented in Table 6.7. There are some initiatives to propose automated defect categorizations, mainly using machine learning models, such as the Suport Vector Machine (SVM) (Thung *et al.*, 2012), neural networks (Glazer and Sipper, 2008), naive bayes, decision trees, etc. In order to increase the confidence on the results, we carried out a manual classification. Table 6.10 shows our findings. The last column (N/C) shows the number of issues we could not classify into any category, as those could not fit into any of the coding defect classes.

Next step was to identify, among the defects, the classes they were referring to. That is, if it had been possible to make any association with the classes, then we could associate the defect to a class. In case it was possible, we verified whether the class was subject to implementation of one (or more) of the selected variability mechanisms. Then we could make an association between the defect class and the mechanism implemented.

In practice, it might be hard to retrieve information whether a variability mechanisms found when walking through the source code were used in the software development project with

Table 6.10: Defect statistics from the open source software systems

Project	**Algorithmic**	**Control & Data**	**Interface**	N/C
`Apache Ant`	105 (62)	280 (98)	30 (18)	85
`Apache JMeter`	43 (28)	113 (52)	12 (10)	32
`Apache Log4J`	18 (10)	59 (35)	5 (5)	18

N/C: Not classified.

the intent of expressing variability, or simply to benefit from the capabilities of the construct, in a non-variability-related implementation. This is especially due to the common lack of documentation to clarify such kind of specifications.

However, taking the perspective that, independently on the purpose, the faults introduced with the use of a given mechanism might be experienced in both scenarios. Therefore, understanding the common problems might be helpful both to use the mechanisms constructs with the intent of either implementing variable entities or not.

There are some bugs in which it is easy to verify the class associated, e.g., Bug ID #509 from Log4J[8], as the reported provides a stack trace of the problem. Hence, all involved files can be traced back. See Figure 6.1 for a snapshot of the issue. However, this was not the average case. In most reported issues it is impossible to trace with the file from which the problem was identified. Whenever it was the case, we discarded the issue. In this particular case, one of the associated class implements polymorphic methods. Hence, the issue could be associated to the *control & data* class of defects.

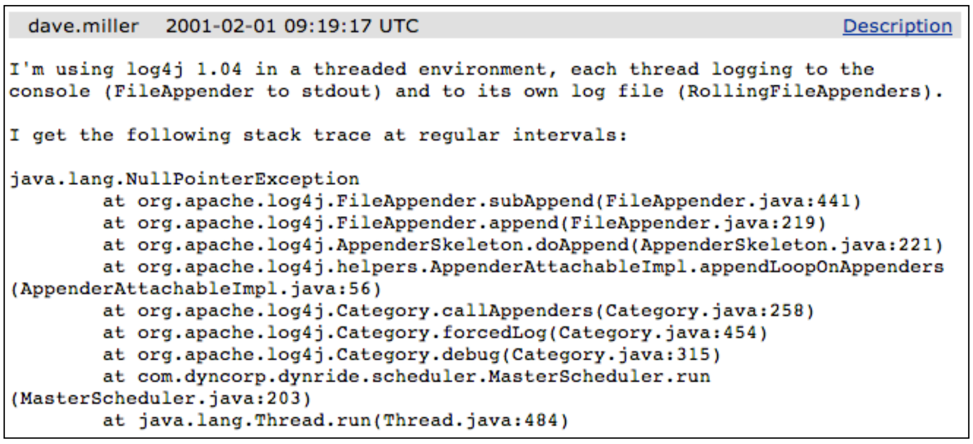
In Table 6.10, the values between parentheses represent the among of defects we could associate with the source files. Next Section summarizes the results.

6.2.1.4 Results

Table 6.11 shows the results of the analysis of the source code. We associated every variability mechanism - *dynamic class loading (M1)*, *interface implementation (M2)*, *polymorphism with sub-classes (M3)* - to the coding defect classes - algorithmic, control, data, interface. The Table shows data for every system analyzed, as follows: *Apache Ant* (A), *Apache JMeter* (B), and *Apache Log4J* (C), and the total amount of defects found per mechanism with respect to each defect class.

From Table 6.6, the first and fifth selected mechanisms, respectively *configuration files*

[8]`https://issues.apache.org/bugzilla/show_bug.cgi?id=509`


dave.miller 2001-02-01 09:19:17 UTC Description

```
I'm using log4j 1.04 in a threaded environment, each thread logging to the
console (FileAppender to stdout) and to its own log file (RollingFileAppenders).

I get the following stack trace at regular intervals:

java.lang.NullPointerException
        at org.apache.log4j.FileAppender.subAppend(FileAppender.java:441)
        at org.apache.log4j.FileAppender.append(FileAppender.java:219)
        at org.apache.log4j.AppenderSkeleton.doAppend(AppenderSkeleton.java:221)
        at org.apache.log4j.helpers.AppenderAttachableImpl.appendLoopOnAppenders
(AppenderAttachableImpl.java:56)
        at org.apache.log4j.Category.callAppenders(Category.java:258)
        at org.apache.log4j.Category.forcedLog(Category.java:454)
        at org.apache.log4j.Category.debug(Category.java:315)
        at com.dyncorp.dynride.scheduler.MasterScheduler.run
(MasterScheduler.java:203)
        at java.lang.Thread.run(Thread.java:484)
```

Figure 6.1: Snapshot of issue # 509 from Apache Log4J project.

Table 6.11: Results from the analysis of open source systems

	M1				**M2**				**M3**			
	A	B	C	*Total*	A	B	C	*Total*	A	B	C	*Total*
Algorithmic	28	12	4	**44**	10	3	2	**15**	24	13	4	**41**
Control & Data	39	25	18	**82**	8	6	5	**19**	51	21	12	**84**
Interface	0	0	0	**0**	14	8	3	**25**	4	2	2	**8**

and *static libraries* were left out from the analysis. We did not find problems associated to configuration files. In the latter, we observed that problems associated to such a mechanism are usually lack of calls to an external libraries, i.e., it is usually managed by the IDE, in project settings (jars to be added to the project classpath). We only find a few issues reporting those kind of problems, e.g., issue #38513[9] from Log4J project.

Table 6.12 provides detailed information on the kinds of erros found. Besides such kinds of errors, there is another set of problems that may occur when implementing variable features, as (Al-Hajjaji *et al.*, 2014) states: independently on the mechanism chosen to implement variability, a defect may occur whenever a specific feature is not selected. For example, if a feature initializes a variable, when the feature is removed, a defect occurs. A defect may also occur whenever two specific features are selected. For instance, one feature calls a method in another feature and the retrieved value is wrong. Next, whenever a specific feature is selected while the another feature is not selected. For instance, one feature calls a method from a feature that is not selected.

We should observe that each of these *new* listed problems, together with those listed in Table

[9]https://issues.apache.org/bugzilla/show_bug.cgi?id=38513

6.6, may be associated to any of the four defect classes analyzed.

Data from Table 6.11 enables deciding which parts test should be focused on, i.e., which kind of defects a test strategy should be directed. For example, the mechanism M1 is more associated to control & data issues, rather than interface issues. This is where the use of fault models is encouraged. Once data about the most occurring kinds of defects is available, a test strategy should prioritize the use of fault models intended to find those most recurring problems. Although we have not considered the severity level of each reported defect, such an information would be valuable to define a test prioritization strategy.

6.2.1.5 Limitations

By carrying out this empirical study, we understand that there are some limitations that prevent us from generalizing the results. First, we understand that not all defects would be related to a variation point, and also that this was a labor intensive and error-prone work. We attempted to mitigate such a problem by involving another interested researcher, that could use such outcomes to their investigations. It may, to a certain extent, reduce the bias, and strenghten the evidence base.

The analysis only included three software projects, and only a few variability mechanisms. Also, we did not validate the classification with external developers. It would be a good strategy to strengthen the evidence base. Besides, for the small amount of data, we decided not to carry out any statistical analysis based on the gathered data.

However, we have some indications on how defects are distributed in three important variability mechanisms, that are widely used in practice, not only in the development of SPL projects, but also variant-rich systems. The results may serve as an insight for those intended to prioritize test design based on the most occurring problems faced when any of the given variability mechanisms are employed in the construction of her variable software projects.

Table 6.12: Fault models for implementation issues

Fault type	**Fault list**
Algorithmic and processing	1. Unchecked overflow 2. Underflow conditions

Continued on next page...

Fault type	**Fault list**
Algorithmic and processing	3. Comparing inappropriate data types 4. Converting one data type to another 5. Incorrect ordering of arithmetic operators 6. Miunderstanding of operator precedence 7. Not using a block when one is required 8. Code that is inside a loop, but that does not belong there 9. Loops that iterates one more or one fewer time than it is supposed to
Data	10. Incorrect data initialization (a) A variable should be initialized, before it is used. (b) A variable should not be initialized twice before there is an intermediate use. (c) A variable should not be disregarded before it is used. 11. NULL object references 12. Incorrect data type or column size 13. Incorrect variable name 14. Valid range undefined 15. Incorrect relationship cardinality in data model 16. Missing or incorrect value in pick list.

Continued on next page…

Fault type	**Fault list**
Control, logic, sequence	17. Dangling else clause 18. Incorrect sequencing of operations 19. Incorrect operator or operand in expression 20. Missing logic to test for or respond to an error condition 21. Input value not compared with valid range 22. Missing system response in sequence diagram 23. Ambiguous definition of business rule in specification
Interface	24. Design incapable of supporting stated requirements - the module interface is not coded as designed 25. Incorrect derivation of physical data model from logical data model 26. Incorrect application program interface design

6.3 Chapter summary

The underlying basis to build a fault model is to understand which kind of problems one should meet in a given portion of the software development lifecycle. Those kind of problems will serve as the basis to build fault models to support variability testing. Indeed, depending on the application domain, a subset of a general fault model could be useful.

Hence, along this Chapter we presented the most common and widely used fault classification schemes. From these, we tailored a classification scheme we believe to be useful to support categorizing defects found in variability testing. We applied the scheme in the analysis of three open source software systems, aiming at observing how faults are distributed, in terms of both the fault categories and the variability mechanisms employed in the implementation of those project entities.

The results from the analysis of open source software systems consist of an important input for designing fault models to support variability testing. Those point our the classes of defects that may occur when implementing a feature using a given mechanism.

Next chapter discusses our fault modeling proposal in details, that makes a wide use of the fault classification scheme detailed in this current Chapter.

7 Fault Modeling for Variability Testing

A key goal in SPL testing is to maximize the quality of products delivered to customers while reducing the cost of testing. As earlier discussed in this thesis, the presence of variability in the features poses a special challenge to SPL testing.

The previous chapters were dedicated to understanding how the existing SPL testing strategies might contribute to achieving such a goal, and also as errors constitute an unavoidable aspect of software development, and this holds true for SPL engineering as well, we also investigated the common faults in variability implementation, as a means to define a suitable fault classification scheme.

We hypothesize that understanding the nature and frequence of occurrence of errors works as a good strategy to improve test effectiveness. This knowledge can be used to build suitable fault models. Based on fault dictionaries, fault models guide the test case design and assessment, by pointing out and prioritizing testing on the most fault-prone elements, attempting to demonstrate that the set of prescribed faults are not present in the system. As a result, increased fault detection rates, while avoiding wasting time and effort.

In this Chapter, we present the proposed approach to building fault models for variability testing, that employs the knowledge acquired so far. The remainder of the Chapter is organized as follows. In Section 7.1 we provide an overview of the fault model based support for SPL testing. Sections 7.2 and 7.3 addresses the capability of the proposed approach to cope with test case assessment and design, respectively. Section 7.4 reports on a controlled experiment aimed at evaluating the effectiveness of the proposed approach, and Section 7.5 concludes the Chapter.

7.1 Overview of the approach

Software testing usually focuses on detecting differences between the observable behavior of a system and its specification, by exploiting the concept of errors. The more errors are found, the more effective the testing strategy employed is. Thus, the ultimate goal of a software testing strategy is to find errors before delivering the final product to customers.

Fault modeling is an engineering model to capture the behavior of the system against faults. A fault model relates faults that a programmer typically makes when implementing a set of requirements to the observable errors they produce. Fault models specify, by means of a fault dictionary, which kinds of faults have to be detected by a test. Hence, test case design and assessment can be guided by fault models to improve test effectiveness. Test effectiveness focuses on improving the quality of test inputs and test oracles (Xiao *et al.*, 2012).

Considering an SPL product instance as a composition of features, the quality of each product is strongly dependent on the quality of each individual feature, as well as on the quality of the interaction among features. Recall the impact a single problem within a feature may have on the set of products that make use of it. Feature implementation lies on the use of one or more variability mechanisms, so that the variability modeled in the problem space can be materialized into code artifacts in the solution space.

In this effect, we may weight the importance of keeping features, both individually and in conjunction, at an adequate level of quality. A fault modeling approach may aid in the process of improving the level of quality. Let us briefly discuss the role of fault modeling from a broader perspective, thus taken into account the overall SPL engineering process.

Recall the generic fault model-based process earlier introduced in Section 2.6 (c.f. Figure 2.3). It consists of a common software development life cycle, in which an element, so called *fault model* will be part of each life cycle phase.

We incorporated the two SPL processes, *domain* and *application engineering*, in an overall fault model-based SPL development life cycle, as Figure 7.1 shows. In either process, the V&V activities in each life cycle phase are the first line of defense. For example, employing proper techniques for requirements verification (Corriveau *et al.*, 2011) aid in the identification of non-conformities before proceeding to the SPL design. In addition, an archictetural analysis method, such as the ATAM method (Kazman *et al.*, 2000) might be applied to identify and report some types of inconsistency among design scenarios. Table 7.1 shows a list of strategies (line of defense) for each software life cycle phase.

It is possible to incorporate fault models in each life cycle phase, in order to guide the

Table 7.1: Testing strategies and the life cycle

Phase	Line of defense
Requirements	• Observe completeness, consistency, feasibility, and testability • Attempt to identify missing, wrong, and extra information • Determine testing strategy, generate functional test cases and test specification, perform reviews and the like
Design	• Functional and structural tests can be devised on the basis of the decomposition • The design itself can be tested - against the requirements, the architecture can be evaluated (e.g., using the ATAM method) • Formal verification techniques can be employed
Implementation	• Check consistency between implementation and previous documents • Code-inspection and code-walthrough • All kinds of functional and structural test techniques • Formal verification techniques
Maintenance	• Regression testing - either retest all, or a more selective retest

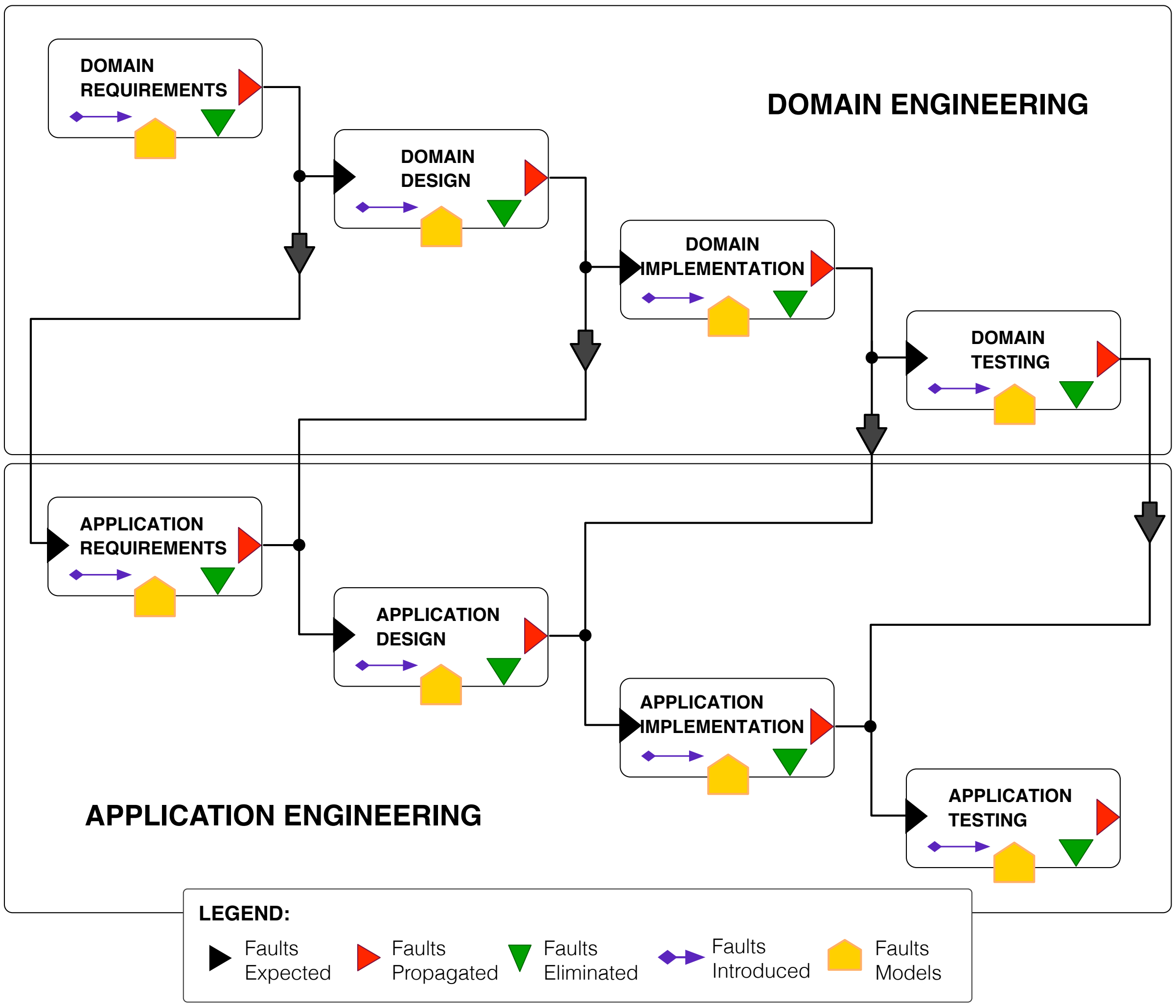


Figure 7.1: Fault Model-based SPL Process.

specification/design of the artifacts, having in mind the likely mistakes anyone can make, when in charge of such a task. The goal is to minimize the amount of faults that are propagated from one phase to another.

Indeed, later phases will also search for those faults that are expected to be propagated from the earlier phases. However, this is where the exponential problem comes into play. Considering that a specification has a problem that had not been identified before leaving to the design phase, and such a problem was then modeled as an architectural design, and, the problem earlier introduced in the specification phase had not been identified again. Proceeding in the life cycle, the implementation is going to materialize such a problem, that much probably will only be identified when testing is carried out. The cost to repair every single artifact, and much probably another set of artifacts that have also been affected by the problem, will be exponentially increased.

As a single artifact built in domain engineering may be reused in a range of products, observe from Figure 7.1 that not only *inter-phases* fault propagation but also *inter-processes* fault propagation may also occur. That is, a single fault in a domain artifact may affect the development life cycle of a range of products.

With fault modeling support, in each phase, a set of fault models can be associated to the V&V activities, and expected fault types and introduced fault types are searched for. The fault models represent the set of instructions to follow to proactively anticipate the problems in the software project. Irrespective of their abstraction level, the whole range of development phases enable to study and leverage its properties, serving as an input to build a fault model for that particular phase.

While a complete fault modeling approach should consider all the software development life cycle phases, the approach proposed in this thesis narrowed down the focus to investigate the problems that are likely to occur when implementing either the source code or the test scripts.

The preceding phases, such as requirements and design, as well as other important SPL-related concerns (product derivation, process and management aspects) are left aside in this thesis. The literature on SPL engineering has provided researchers with a large set of explicit activities and operations to follow in the formulation of both specifications - comprising features and requirements, e.g., (Corriveau *et al.*, 2011; Souza *et al.*, 2013), and design - encompassing activities to assess the quality of the product line architecture, e.g., (Matinlassi *et al.*, 2002; Thiel, 2002; Etxeberria and Sagardui, 2005; Kim *et al.*, 2008; Etxeberria *et al.*, 2008; Nakagawa *et al.*, 2011), just to name a few, that are relevant to build fault model specific to inspecting such artifacts.

Figure 7.2 shows the overall variability testing workflow, enhanced by the fault modeling support. It shows the relationship between the fault models and the test process. The solid arrows (direct link) represents a relationship among test tasks (design, execution, and reporting), and between test phases and other elements (design, source code, fault models, test cases, and knowledge base), emphasizing which ones provide input to others. There are two more kinds of relationships, represented by dashed arrows: (i) subjective link (produces), that particularly indicates that *test cases* are produced by the task *test design*, and (ii) subjective link (uses/analyzes), that indicates that some test phase/element makes use of any other element.

The fault models, by means of their associated fault dictionaries, will be mainly used to support the design of *test cases*. The fault dictionaries pinpoint which faults are more likely to occur, given a particular *variability mechanism*, involved in the *source code implementation*. To this end, fault models should be aware of variability mechanisms used in the project.

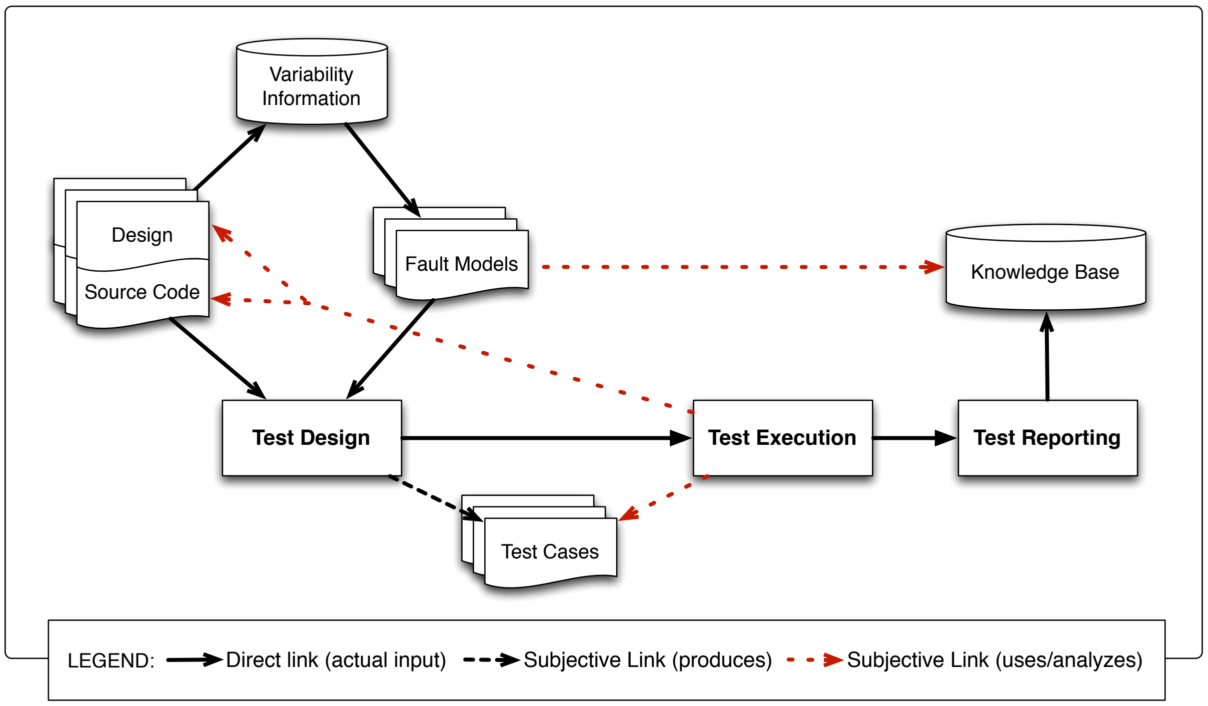


Figure 7.2: Overall variability testing workflow, enhanced by the fault modeling support.

The fault model is a living document so that, as new faults are found that have not been listed in the model, an analysis on the problem should be undertaken, so as to enable its incorporation into the model. For this reason there is a feedback arrow, from the *test reporting* task to the *knowledge base* repository. The idea behind a *knowledge base* comes from the fact that all fault models can evolve to include novel constructs. And test running can be the input of such information. The repository specifications are not detailed herein, but in essence it comprises the list of fault models, and their semantics, namely descriptions on which scenario it should be applied to. The idea behind it might be compared to design patterns, where known problems are modeled to avoid repeating the problem. Likewise the patterns, in which developers are provided with usage scenarios, every fault model is expected to accomplish a real application scenario.

Despite the capability of current technologies (both hardware and software) have reduced to a minimum the effort of automatically generating the test sets, a huge subset of the generated test cases may be useless. Hence, rather than generating huge sets of test cases, it seems to be more important to sketch a strategy to aid in the understanding of how the program should behave, under a set of circsumstances, so that testing can be focused on uncovering the most likely faults.

A fault model contains historical data about commonly occurring errors, with data coming from two main sources: *(i)* data from the same project under evaluation, in case past test runs has provided the knowledge base (c.f. Fig. 7.2) with appropriate feedback, or *(2)* from a historical database, which encompasses knowledge from other projects implemented in similar conditions, in terms of project domain, size, programming language, etc. All such information might influence the fault distribution and occurrence, and should be taken into account accordingly.

Inspired by the theory of fault-based testing by Morell (1990) and highlights from the work of McGregor (2008), who first introduced the notion of fault modeling in SPL engineering, we next describe how variability testing, key in SPL engineering, can benefit from a fault modeling approach. The approach addresses two main perspectives: first, applying the fault model concept towards assessing existing test sets, and second, aiding at designing test cases, by prioritizing test effort according to fault-proneness of the elements in the source code.

It is worth mentioning that we are not concerned about generating test suites for SPL feature models, but instead generating test suites for the actual source code of SPL features, in line with the second known SPL interest, as discussed in (Machado *et al.*, 2012b).

7.2 Fault modeling for test suite evaluation

For the purpose of this perspective, we can assume that an SPL project already comes with a test suite. As a consequence, every product instance may comprise a subset of the SPL test suite. Considering we are strictly working in testing variability mechanisms implemented in Java, we could consider test suites as a set of test scripts, implemented in any ordinary test automation framework such as the JUnit.

This first perspective is aimed at employing fault models to evaluate the effectiveness of the existing test sets. Figure 7.3 illustrates the evaluation workflow.

Let an SPL be a tuple $< F,T >$ where F is a feature set, and T is a test suite. Program $P \in < F,T >$ is a runnable instance of the SPL. P is not an actual product instance, but it is rather a valid subset of features $\Phi \subseteq F$, or even a single feature $f_i \in F$, that can be tested as an isolated instance. Hereinafter, each P will be referred to as a *Program Under Test (PUT)*.

Each feature $f \in F$ may be associated to a set of tests $t \in T$. Let R be a subset of T, then there is a function $X : F \to R$, which represents the set of test cases that are suitable to a feature f_i. Each feature f_i also holds information about the mechanisms $M = \{me_0, me_1, \ldots, me_n\}$ employed to implement variability in the feature. As we intend to take into account variability information, we consider that a variability implementation mechanism $m \in M$ can be associated to a set of

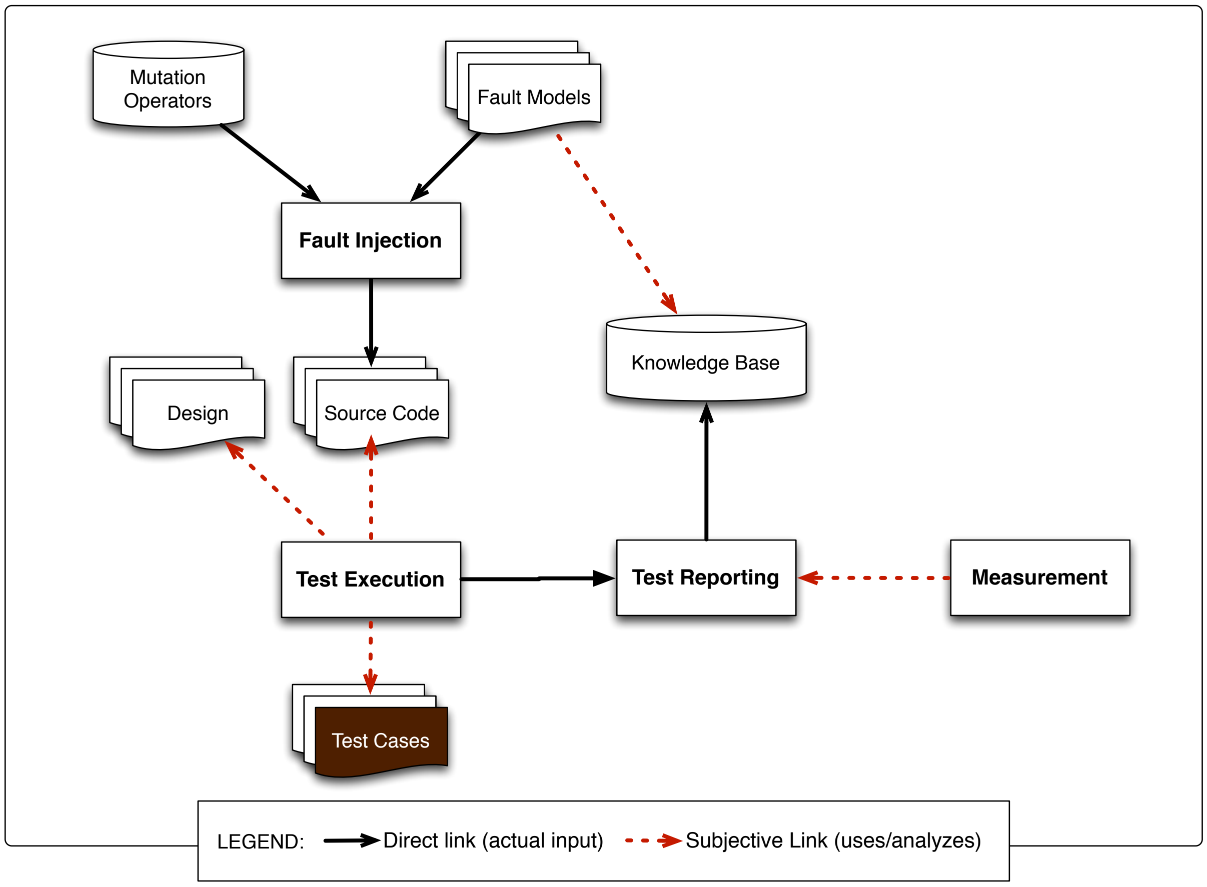


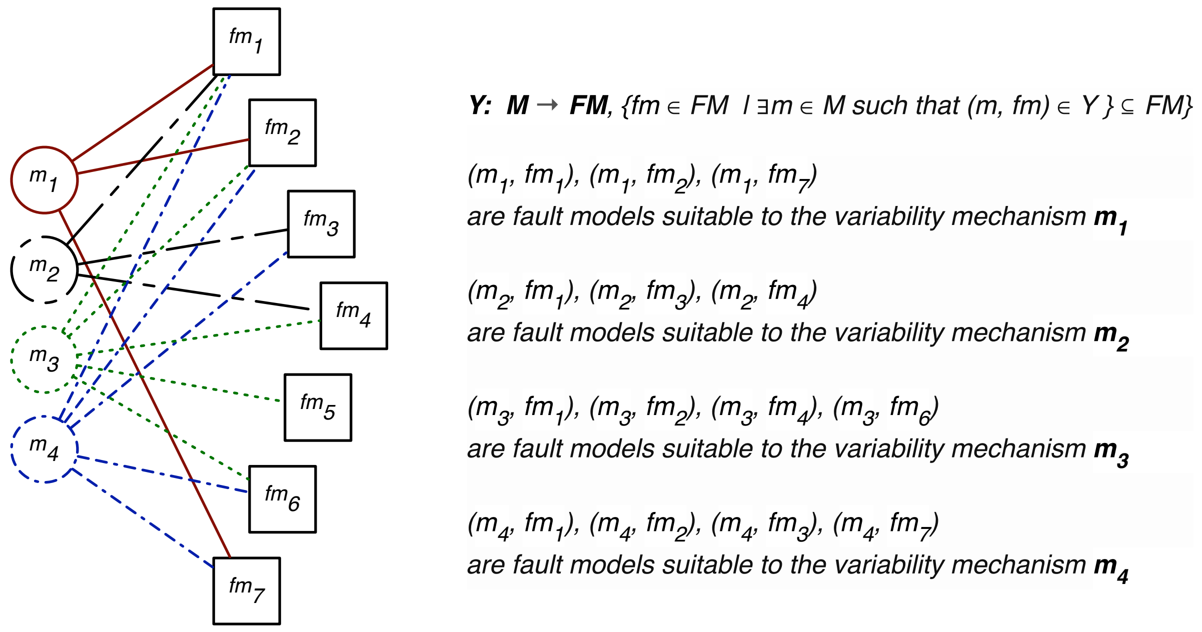
Figure 7.3: Overview of the evaluation workflow.

fault models $FM = \{fm_0, fm_1, \ldots, fm_p\}$, so that $Y : M \rightarrow FM$. Figure 7.4 illustrates how this set relation could be in practice, namely there might be several fault model serving any variability mechanism, and vice-versa.

A fault model fm_i subsumes information about fault types to search for when testing a program P which uses it. Building a fault model consists of analyzing historical data to understand which fault types, and associated faults, are occurring in a given variability mechanism, and in which frequency range.

Now, let S be the program specification represented schematically as $S \vdash \{\forall inputs, \exists output \mid spec(input, output)\}$, where *input* is a vector of arguments, *output* is an expected result and *spec* is a proposition function describing the required relation between them. Hence, a program P under test (PUT) is a 3-tuple $< X, Y, S >$.

Testing P consists of checking that the behavior of an implementation, its *actual output* is conform to its specification, namely its *expected output* (the *output* from function *spec* above), given a set of *inputs*.



Y: M → FM, *{fm ∈ FM / ∃m ∈ M such that (m, fm) ∈ Y } ⊆ FM}*

(m_1, fm_1), (m_1, fm_2), (m_1, fm_7)
are fault models suitable to the variability mechanism **m_1**

(m_2, fm_1), (m_2, fm_3), (m_2, fm_4)
are fault models suitable to the variability mechanism **m_2**

(m_3, fm_1), (m_3, fm_2), (m_3, fm_4), (m_3, fm_6)
are fault models suitable to the variability mechanism **m_3**

(m_4, fm_1), (m_4, fm_2), (m_4, fm_3), (m_4, fm_7)
are fault models suitable to the variability mechanism **m_4**

Figure 7.4: Set relation $F \rightarrow FM$ illustrated.

Hence, to carry out test evaluation, a subset of features Φ will be selected. Each $f_i \subseteq \Phi$ is associated to a set R of test cases. These are *developer tests*, i.e., those developers design to test their code as they write it, as opposed to the tests done by a separate quality assurance organization. Developer testing, often in the form of unit testing, helps developers to both gain high confidence in the program unit (e.g., a class) under test while they are writing it, and reduce fault-fixing cost by detecting faults early when they are freshly introduced in the program unit (Xiao *et al.*, 2012). However, as we are dealing with features composed of one or more classes, we also consider the integration test level. While a unit test is usually a sequence of method calls on an object instance, therefore the main components are method and constructor calls, an integration test involves interfaces between components, and, from a broader perspective, the likely interaction between features, whenever a feature depends upon calling an external object.

Furthermore, as each $f_i \subseteq \Phi$ holds information about the variability implementation mechanism, then let $faultModels(m)$ be a function that returns a list containing the classes of errors of the fault models that are appropriate to the mechanism $m \in f_i$.

Besides, let $\tau(P,R)$ be a function that executes test cases $t_n \subset R$ on program P against the specifications S and returns the outcome of the test execution. The *actual output* can be of one of the following types:

1. *Pass*: The execution of P against t_i succeeds.

2. $Fail_{CE}$: The execution of P against t_i fails because a class or method accessed in t does not exist in P.

3. $Fail_{RE}$: The execution of P against t_i fails due to an uncaught runtime exception.

4. $Fail_{AE}$: The execution of P against t_i fails due to an assertion violation.

Please notice in Figure 7.3 the presence of a task called *fault injection*, and a repository called *mutation operators*. In mutation analysis terminology, a mutant is a version of a software program which differs from the original by a single potential error (DeMillo *et al.*, 1978). A mutation operator is a function which is applied to the original program to generate a mutant (Martin and Xie, 2007). Hence, we can use a set of mutation operators to describe all expected errors, and therefore defines the behavioral fault model. The proposed approach does not make distinctions, for the time being, between *static* and *dynamic* fault injection. Given that we combine mutation analysis and fault injection, we could be inclined to narrow down the focus to the former. However, it is important to mention that a wide variety of faults listed in the fault models can be dynamically emulated as well.

Given a set of test cases T, the fault models may indicate some mutation operators to program P to produce a modified version, a mutant P'. A set of representative faults, suggested by the fault models, are injected into the code of P. Hence, let function $\tau(P,T)$ be executed.

Then, it will be possible to measure the adequacy of test cases, i.e., a test case is *adequate* if it is effective at detecting faults in the program (Offutt *et al.*, 2001).

The mutants are run with an input data *input* from a given test set T. If a test set can distinguish a mutant P' from the original program P, i.e., it produces a different *output*, the mutant P' is said to be killed. Otherwise, the mutant is called as a live mutant. That is, if after modifying the source code, with the set of mutatns, the same output is observed, it means that the test cases are not *adequate* enough. Conversely, a test set which can kill all non-equivalent mutants is said to be *adequate*. That can be explained by the mutant score calculation, as follows.

The mutation score $ms(P,T)$ is defined as the ratio between the number of mutants detected and the total number of mutants minus the equivalent ones (Jia and Harman, 2011). A mutant is said to be equivalent if it syntacticaly differs from the original program, but semantically the mutation can not be detected. A test set T is *mutation adequate* if its mutation score is 100%. The score *ms* can be calculated as follows:

$$ms(P,T) = 100 * \frac{DM(P,T)}{MT(P)}$$

where:

- $DM(P,T)$ is the number of mutants killed by T;
- $MT(P)$ is the total number of mutants generated from P;

Fault models are expected to increase the probability of finding a given fault as the associated metric. However, this attribute should really reflect the percentage of faults that the technique can detect.

7.3 Fault modeling for test suite design

The second perspective encompasses the fault modeling support for test design purposes. Relying primarily on a prioritization strategy, test design seeks to identify *what to look at prior to testing*.

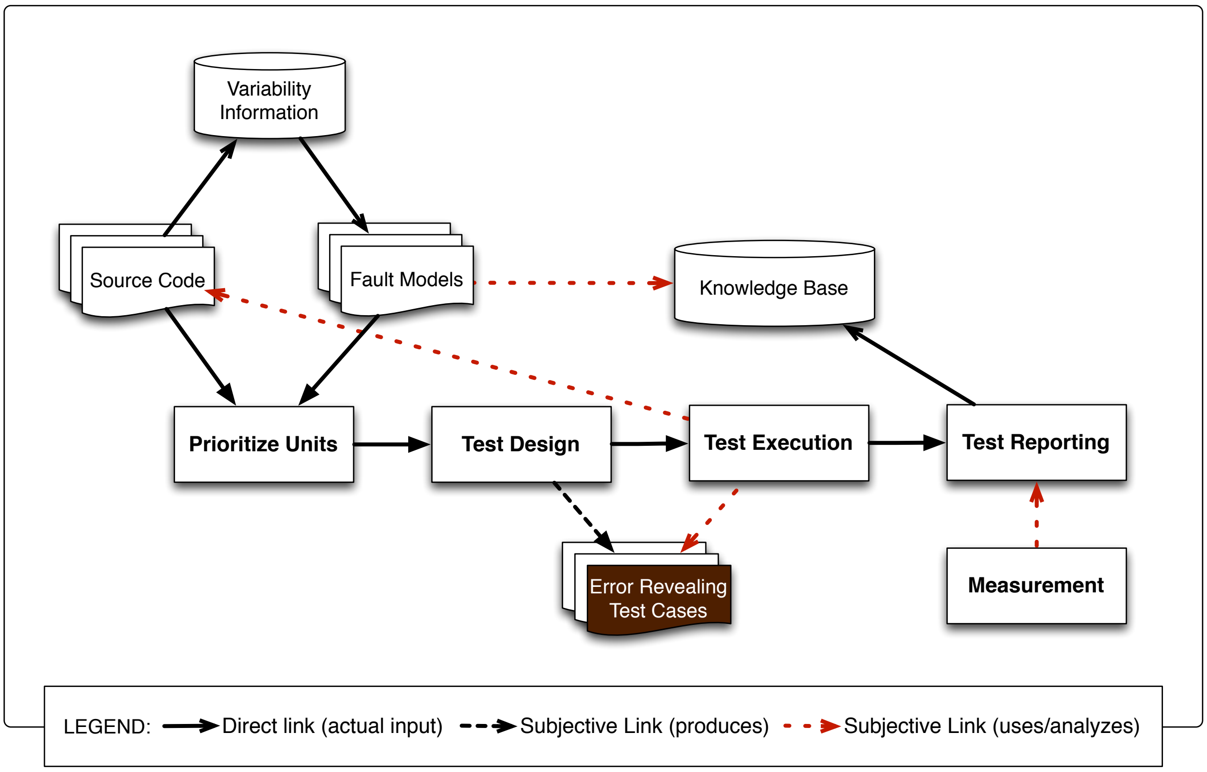


Figure 7.5: Overview of the test generation workflow.

There exists a bunch of formalized test prioritization techniques, as deeply discussed in (Catal and Mishra, 2012). Most of them are concerned about establishing effective means to improve test case selection aimed at regression testing. They are usually concerned about analyzing past historical data about modules structure (e.g., size, coupling, etc), bugs reported, bug fixing modifications, and general maintenance modifications. Based on such data, some heuristics are calculated that enable prioritization.

Our goal is not to propose a novel test case prioritization strategy. Indeed, this perspective aims at prioritizing the units for testing, not the test cases themselves, as they still do not exist.

Figure 7.5 shows an overview of the test design workflow. There is a task called *prioritize units*. We followed the prioritization principles first introduced by (Rothermel *et al.*, 2001). The heuristic defines that each unit in the source code will be attributed to a *weight*, namely an integer value. The prioritization strategy consists of analyzing every attribute and weighting every unit. The principle is that the element with maximum weight is taken first, followed by the element with the second highest weight, and so on.

A feature f_i is composed of a set of units $u_i \in U$. Following the ideas of generating unit tests described in (Fraser and Zeller, 2012), and tailoring it to subsume integration concerns, as it may illustrate how variable entities should interact with each other, let a unit be a 6-tuple $u =< cs, ms, fs, ps, \delta, \varepsilon >$, with all values of type integer, where:

- cs is the number of *constructor statements* in a unit;
- ms is the number of *method statements*;
- ps is the number of *primitive statements*;
- δ is the number of *calls to internal units*, i.e., units from the same feature;
- ε is the number of *calls to external units*, i.e., units from other features.

In order to simplify the understanding, let us assume that a unit is always a Java class, and that every feature is associated to one or more Java classes. Conversely, one single class is considered not to belong to more than one feature. Thus, testing an “unit” must concentrate on the state of the objects and check that the state remains consistent when executing the operations of an object.

For example, let us consider the feature model of the *PL_SimElevator* project (this is further detailed in Section 7.4) Figure 7.7 shows. We analyzed the entities from such an SPL project. Table 7.2 shows metrics of some of the classes, according to the prioritization strategy elements.

Although Table 7.2 includes a last column called *weight.* in which the sum of the values is given, the prioritization depends on which elements the engineer consider as the most important for the project. However, it is also possible to consider the sum of all value. For example, if the

Table 7.2: Data from the PL_SimElevator SPL.

Feature	Class	cs	ms	ps	δ	ε	Weight (Σ)	Mechanisms
1. Buttons	*A:* ArrowButton	1	7	5	0	3	16	m_1, m_2
	B: AuthorizeButton	1	1	0	0	0	2	
	C: DefaultButton	3	8	6	0	6	25	
	D: EmergencyButton	1	4	7	0	0	12	
2. Controller	*E:* SuperController	1	5	0	4	3	13	m_1, m_3, m_4
	F: AbstractController*	1	20	10	0	5	36	
	G: DistanceEController	1	9	2	0	18	30	
	H: DefaultEController	1	7	0	0	10	18	
3. Elevator	*I:* Elevator	0	10	5	0	0	15	m_1
	J: Target	2	6	3	0	0	11	

(*) Abstract class. Only concrete methods were considered.

prioritization strategy considers the total amount, the order of the units for which test cases will be designed can be F-G-C-H-A-I-E-D-J-B.

Special attention should be paid to the presence of abstract classes. Notice from Table 7.2 that the abstract class (`F: AbstractController`) received the highest total weight. In order to design a unit test for such a class, the test engineer should simply write a mock object[1] and use them just for testing (Freeman *et al.*, 2004). They usually are very minimal (inherit from the abstract class) and not more. Then, in the unit test she can call the abstract method she want to test. Unless the abstract class does not contain some logic, it should be tested like all other concrete classes.

Furthermore, the task *prioritize units* has as input the source code and the fault models. The source code contains variability information to be consumed by the prioritization task in order to select the most suitable fault models. From function $Y : M \rightarrow FM$, where M is a set of variability mechanisms, and FM a set of faults models $(fm_i) \mid (fm \in FM)$ that can be suitable to any $m \in M$.

It is expected that each unit has information about the variability mechanism employed in its construction, so that the PUT can be associated to a set of fault models (c.f., Fig. 7.4). Hence, it is important to know which mechanisms each feature (and its associated classes) implement, so

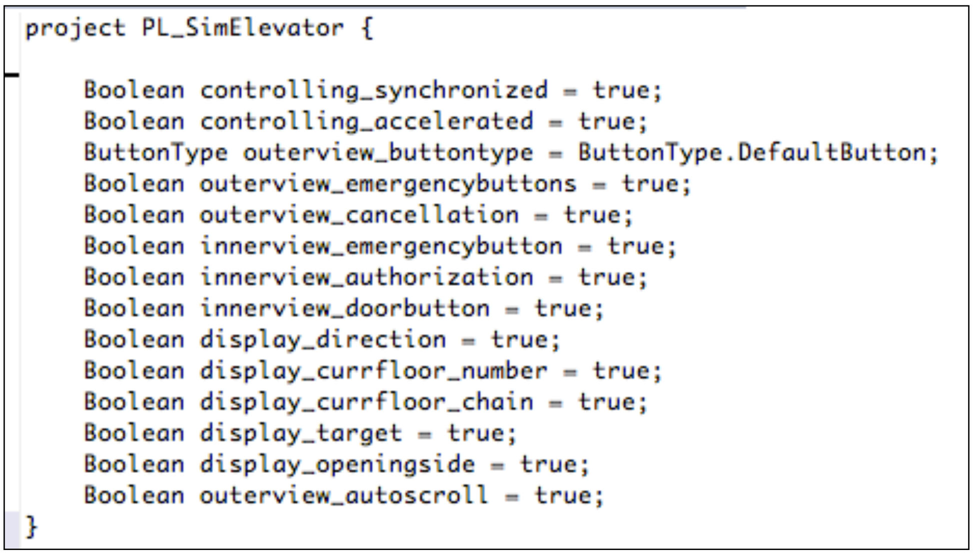
[1]A mock object is simply a debug replacement for a real-world object.

that proper fault models can be associated to it.

In the running example, the variable classes mostly implement the following mechanisms: *configuration files (m_1), dynamic class loading (m_2), interface implementation (m_3)* and *polymorphism (m_4)*.

We should mention some important facts about the variability implementation in this SPL project. The use of configuration files is a capability of the EASy-Producer tool, we used to handle this SPL project[2]. The tool enables the selection of features for a product instance based on the use of configuration files. While we are not dealing with how the tool works, as it is not the scope of our investigation, we are concerned about the way the features are included or not given a selection. Other widely used variability engineering tools might work differently, e.g., Feature IDE (Thüm *et al.*, 2014), BigLever Gears (Krueger and Clements, 2013), or pure::variants (Beuche, 2013). We worked with the EASy-Producer tool both for convenience and easiness of mapping configuration files and features.

The general idea underlying the selection of features is to associate each configuration file entry, disposed sequentially in a *.ivml* file (c.f. Figure 7.6 for an `.ivml` file sample), with a Boolean global constant, and `if-else` blocks of code are called whenever such kind of variability is requestes. Indeed, how the tool parses the code is not explicit.


```
project PL_SimElevator {

    Boolean controlling_synchronized = true;
    Boolean controlling_accelerated = true;
    ButtonType outerview_buttontype = ButtonType.DefaultButton;
    Boolean outerview_emergencybuttons = true;
    Boolean outerview_cancellation = true;
    Boolean innerview_emergencybutton = true;
    Boolean innerview_authorization = true;
    Boolean innerview_doorbutton = true;
    Boolean display_direction = true;
    Boolean display_currfloor_number = true;
    Boolean display_currfloor_chain = true;
    Boolean display_target = true;
    Boolean display_openingside = true;
    Boolean outerview_autoscroll = true;
}
```

Figure 7.6: Example of a `.ivml` file containing the selection of features.

In addition, in this project *dynamic class loading* was implemented using the Java Reflec-

[2]EASy-Producer is a tool for the development of Software Product Lines. It was developed by University of Hildesheim, SSE, Germany. More information can be found at: `http://www.sse.uni-hildesheim.de/en/EASy-Producer`

tion API. The remainder variability mechanisms were implemented like any ordinary Java implementation.

In this particular project, it was not difficult to associate variability mechanisms with the units, as this sample SPL project was designed for academic purposes, so that all documentation is available. However, for legacy systems where there is no clear mapping between mechanisms and software entities, it might be harder to select the fault models based on the variability mechanisms. It is a scenario which demands the presence of both a domain expert and someone with adequate knowledge on the software project architecture.

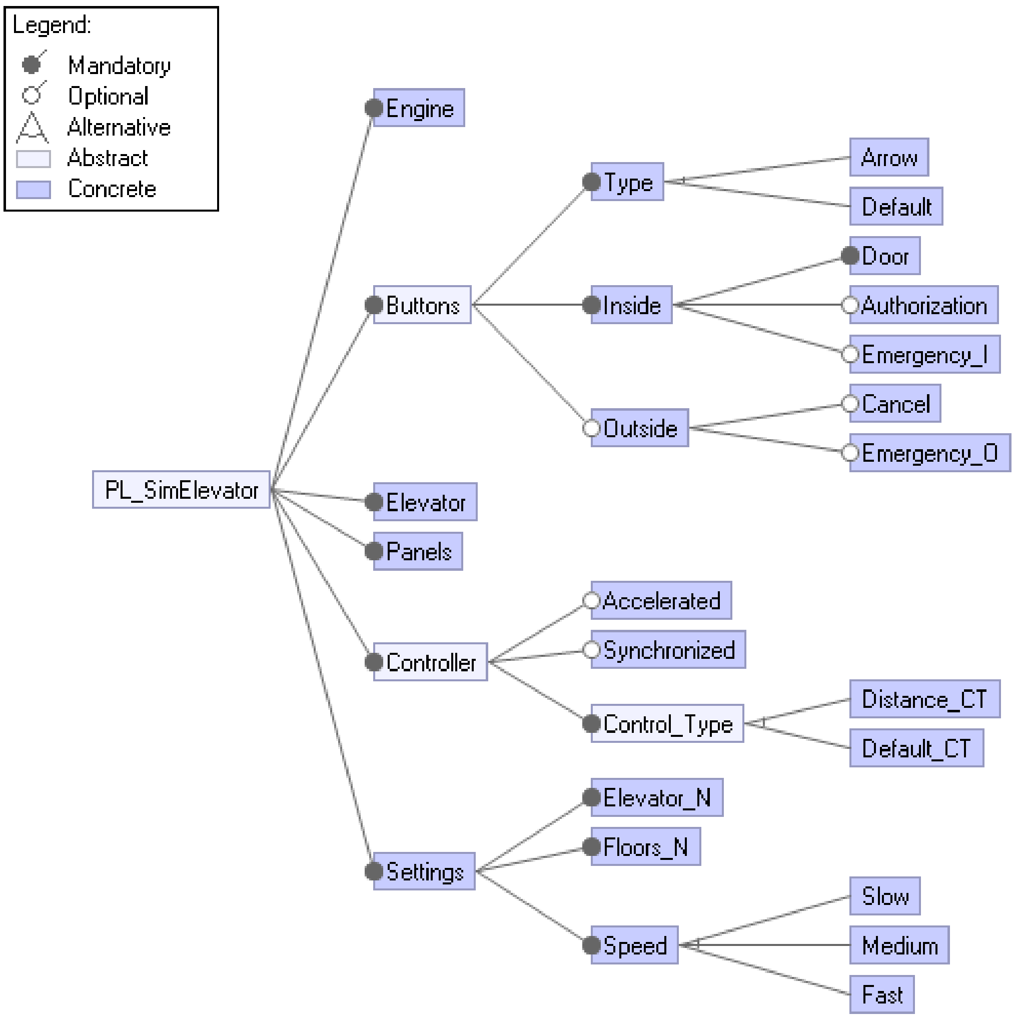


Figure 7.7: Feature model of the elevator simulator SPL.

Let us consider that there is no test suite available to the PUT, i.e., $P \in < F, \emptyset >$. The goal is to find all $r \in R \mid R \subseteq T \wedge F \rightarrow R$. On the other hand, the goal of R is to encompass tests that show that particular classes of faults, defined in the fault model, are in program P when $\tau(P,R)$

is executed.

For the purpose of documentation, for each test case, an entity called *test purpose*, containing some statements indicating what kind of errors the test case tries to detect, is suggested. It comes from the observation that, although the fault models can be implicitly realized by observing how their test cases are generated, the absence of explicitly stating so gives the impression that having fault models beforehand is unnecessary.

It is important to be aware that, although most of the fault types are programming language-independent, the language that is used might affect how the faults manifest.

In this approach, observe that instead of using fixed fault models to generate test cases, the task of defining fault models is left to the users. Leaving the duty of specifying fault models to the users provides more flexibility for generating test cases. That is, a general method is proposed such that test cases can be generated by using any software suitable testing technique.

7.4 Empirical evaluation

This section describes a controlled experiment performed in an industrial setting that aimed at **evaluating the strength and significance of the proposed fault modeling approach**. The experiment involved seven experienced software engineers, all of them familiar with the development of variant-rich software systems, but not familiar with SPL engineering. We analyzed how the fault models could support the testing of variable features in an SPL project.

7.4.1 Experiment planning

This experimental study focused on the following research questions:

- **RQ1. Does the use of fault models lead to best variability testing results?** This question aims at investigating whether the fault modeling solution is worthwhile to be used by practitioners.
- **RQ2. Is the fault modeling approach helpful to uncover the faults that the fault models prescribed?** Not only finding more errors, but finding those that were actually pointed out by the fault models might be another measure of effectivenes. Hence, this question aims to measure the precision, recall, and F-measure of the results, for assessing accuracy.

This study was inspired by the preliminary evaluation reported in Chapter 5. We cannot assume this as a replication study, but in practice they both share some elements. This holds true for the set of **metrics** used in this experiment. From the preliminary evaluation, we applied the following two metrics:

- *Test Case Effectiveness **(TCE)***
- *Test Coverage **(TCov)***

We also calculated the ***ms score***, as earlier defined in Section 7.2, to assess whether the fault models help identifying the expected fault types.

7.4.1.1 Hypotheses

The use of fault models to anticipate hot spots for testing is assumed to yield better fault coverage, improving the software testing activity. Thus, we formalized the definition of the *null* and *alternative* hypotheses that drive this investigation. Table 7.3 shows both the null (H_{0n}) and alternative (H_{1n}) hypotheses.

Table 7.3: Hypothesis formulation - PL_SimElevator SPL.

Null Hypothesis	***Alternative Hypothesis***
$H_{01} : \mu_{TCE_{FM}} \leq \mu_{TCE_{AH}}$	$H_{11} : \mu_{TCE_{FM}} > \mu_{TCE_{AH}}$
$H_{02} : \mu_{TCov_{FM}} \leq \mu_{TCov_{AH}}$	$H_{13} : \mu_{TCov_{FM}} > \mu_{TCov_{AH}}$
$H_{03} : \mu_{MS_{FM}} \leq \mu_{MS_{AH}}$	$H_{13} : \mu_{MS_{FM}} > \mu_{MS_{AH}}$

Legend: AH - adhoc (without the support of fault models) | FM - fault model supported test design

7.4.1.2 Variables

Independent variables. The independent variables are those that we have the control when executing the experiment (Wohlin *et al.*, 2012). In this experimental evaluation, the independent variables are designed test cases and background experience of the participants.

Dependent variables. The dependent variables are those that we observe in order to understand the effects produced in the treatments (Wohlin *et al.*, 2012). In this evaluation, the dependent variables are the uncovered faults.

7.4.1.3 Selection of subjects

The subjects were chosen based on *convenience* (Wohlin *et al.*, 2012). We randomly selected a set of software engineers from a partner company from Salvador, Brazil[3]. The company has had some interest in the software reuse field, and noticed that by introducing SPL engineering practices into their software development, the company could yield better results. Hence, we provided its software engineers with training sessions on the topic, in which they could learn from us how to systematically reuse their developed software artifacts, and next, some of them were selected to serve as subjects in this evaluation.

7.4.1.4 Instrumentation

The instruments of this experiment are the consent form, background and feedback questionnaires, and the source code and documentation of the SPL project under testing.

We used a Java-based SPL, called *PL_SimElevator*. The project consists of an SPL aimed at simulating the operation of an *elevator controller*. Variability in the SPL was implemented using *inheritance, polymorphism* and *encapsulation*. The project comprises 11 optional features. From this SPL, distinct variants of elevator controllers can be generated, to meet different product configurations. Some of the variants are bound at compile-time, while others necessarily at runtime.

Figure 7.7, presented in the preceding section, shows the feature model of the *PL_SimElevator* project. The SPL contains over 3,500 lines of code, 12 packages, and 32 classes. The class diagrams and a package diagram, with the relationship between the entities, can be found in Appendix C.

We generated three product instances with the aid of the Easy-Producer tool. They are: *(i) Simple Elevator, (ii) Enhanced Elevator*, and *Premium Elevator*. Figure 7.8 shows their configurations, generated with the FeatureIDE tool (Thüm *et al.*, 2014).

7.4.1.5 Design

In this experiment, we employed a completely randomized design, with *one factor with two treatments*. Alike the preliminary evaluation, we also compared the two treatments against each other, namely test design with the support of fault models and test design without such a support. The participants were randomly allocated to each group. The control group applied

[3]Recôncavo Institute of Technology. Available at: `http://reconcavotecnologia.org.br/`.

software testing techniques they are familiar with, and the experimental group used the fault lists/dictionaries from the fault models available to them.

It is worth mentioning that for both treatments, the data used in the test design, i.e., the source code and asssociated documents remained the same, as a means to avoid possible biases.

7.4.2 Experiment operation

The experiment was run late in 2013. The participants were all given a 1h introductory lecture where an overview of the study was given. A background questionnaire was also used to elicit information related to the demographic characteristics of the participants, as described in Appendix B. Table B.1 shows raw data for all respondents. The anonymity of the participants was guaranteed. Tha participants also had to take a a 3h introductory lecture on SPL engineering.

A 4h training session was held, where the participants could become familiar with the SPL project. All participants had the opportunity to practising their own test design strategy. As earlier stated in this Section, all software engineers involved in this study had some experience in the development of variant-rich systems, thus involving commonly used variability mechanisms for Java.

The participants were given the source code of every product instance (premium, enhanced, and simple). The source code contained documentation comments so that JavaDoc could be generated seamlessly.

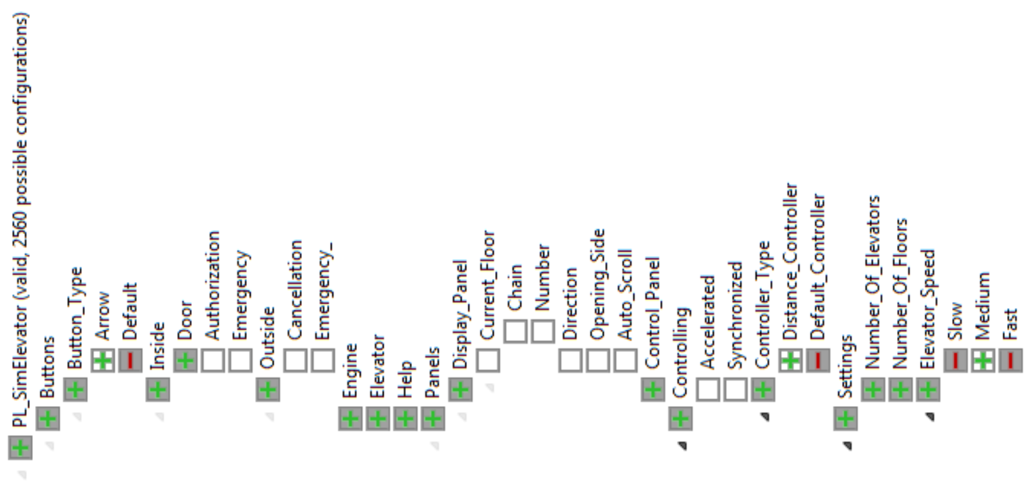
The experiment session, hereinafter referred to as *study section*, was held one day later. The participants had 4h to complete the task. The task consisted of designing and implementing JUnit test cases for entities from the packages `gui.buttons`, `simulator`, `simulator.controllers`, and `simulator.model` (Table 7.2 lists all classes involved in the testing activity, and Appendix C details the project classes considered in this study). The participants had to analyze the source code, and implement unit and integration test cases from the source code, using the JUnit framework.

As our intention was to simulate a real testing environrment, the participants were told to implement both *unit* and *integration* tests. These are usually tests a *developer* implement to either execute a specific functionality in the code (unit) or test the behavior of a component or the integration between a set of components (integration). The participants could implement mocks whenever a method depends on other parts of the system.

Indeed, sometimes it might be hard for an average software developer to only implement a single test level, especially when the code is already developed. Unit testing is a very common strategy to follow in SPL engineering when the source code is to be developed, such as is test-driven development (Ghanam *et al.*, 2008). There are some other studies in the SPL field that consider unit testing, but units are large-scale entities (Ganesan *et al.*, 2012), rather than single source code functionalities.

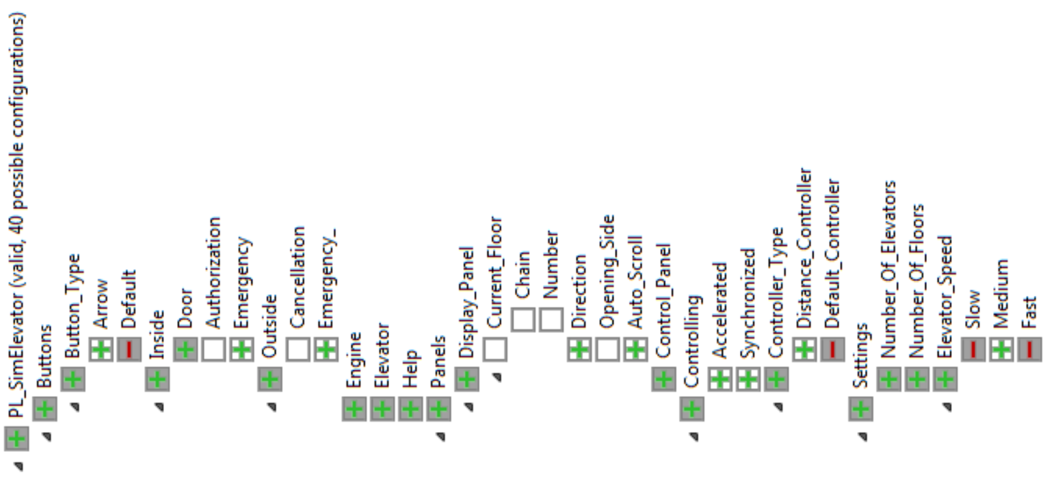
Some further instructions on how to proceed with the creation of packages to accomodate the tests and other details were given during the study section.

The participants had to design test cases to handle variability testing. That is, their must cover the more variation points they could, from the set of packages previously mentioned. The idea was to design the test cases thinking of their further reusability. The test effectiveness would be measured by considering the capability of a test case to be reused in other product instances, and also its capability of uncovering defects.



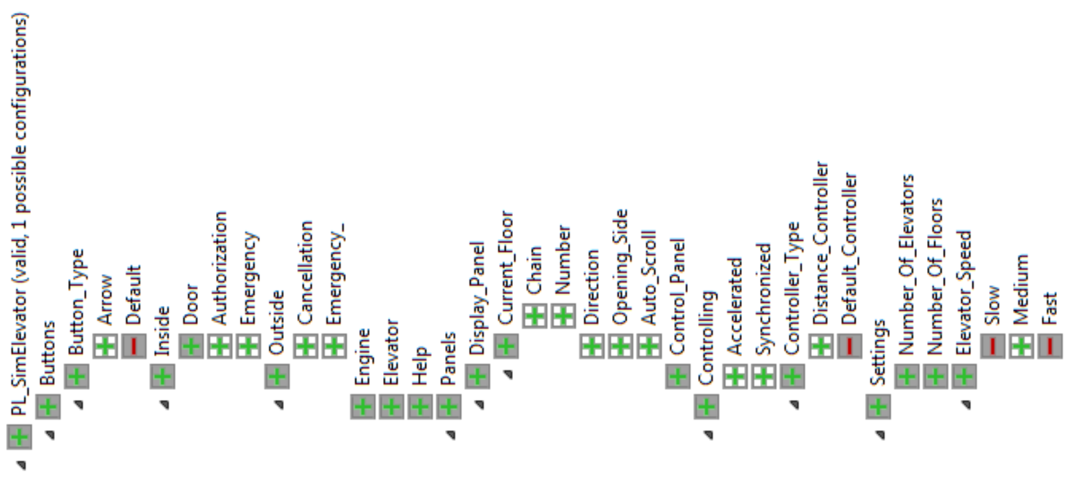
- 🟩 PL_SimElevator (valid, 2560 possible configurations)
  - 🟩 Buttons
    - 🟩 Button_Type
      - 🟩 Arrow
      - 🟥 Default
    - 🟩 Inside
      - 🟩 Door
      - ☐ Authorization
      - ☐ Emergency
    - 🟩 Outside
      - ☐ Cancellation
      - ☐ Emergency_
  - 🟩 Engine
  - 🟩 Elevator
  - 🟩 Help
  - 🟩 Panels
    - 🟩 Display_Panel
      - ☐ Current_Floor
        - ☐ Chain
        - ☐ Number
      - ☐ Direction
      - ☐ Opening_Side
      - ☐ Auto_Scroll
    - 🟩 Control_Panel
  - 🟩 Controlling
    - ☐ Accelerated
    - ☐ Synchronized
    - 🟩 Controller_Type
      - 🟩 Distance_Controller
      - 🟥 Default_Controller
  - 🟩 Settings
    - 🟩 Number_Of_Elevators
    - 🟩 Number_Of_Floors
    - 🟩 Elevator_Speed
      - 🟥 Slow
      - 🟩 Medium
      - 🟥 Fast

(a) Simple elevator.



- 🟩 PL_SimElevator (valid, 40 possible configurations)
  - 🟩 Buttons
    - 🟩 Button_Type
      - 🟩 Arrow
      - 🟥 Default
    - 🟩 Inside
      - 🟩 Door
      - ☐ Authorization
      - 🟩 Emergency
    - 🟩 Outside
      - ☐ Cancellation
      - 🟩 Emergency_
  - 🟩 Engine
  - 🟩 Elevator
  - 🟩 Help
  - 🟩 Panels
    - 🟩 Display_Panel
      - ☐ Current_Floor
        - ☐ Chain
        - ☐ Number
      - 🟩 Direction
      - ☐ Opening_Side
      - 🟩 Auto_Scroll
    - 🟩 Control_Panel
  - 🟩 Controlling
    - 🟩 Accelerated
    - 🟩 Synchronized
    - 🟩 Controller_Type
      - 🟩 Distance_Controller
      - 🟥 Default_Controller
  - 🟩 Settings
    - 🟩 Number_Of_Elevators
    - 🟩 Number_Of_Floors
    - 🟩 Elevator_Speed
      - 🟥 Slow
      - 🟩 Medium
      - 🟥 Fast

(b) Enhanced elevator



- 🟩 PL_SimElevator (valid, 1 possible configurations)
  - 🟩 Buttons
    - 🟩 Button_Type
      - 🟩 Arrow
      - 🟥 Default
    - 🟩 Inside
      - 🟩 Door
      - 🟩 Authorization
      - 🟩 Emergency
    - 🟩 Outside
      - 🟩 Cancellation
      - 🟩 Emergency_
  - 🟩 Engine
  - 🟩 Elevator
  - 🟩 Help
  - 🟩 Panels
    - 🟩 Display_Panel
      - 🟩 Current_Floor
        - 🟩 Chain
        - 🟩 Number
      - 🟩 Direction
      - 🟩 Opening_Side
      - 🟩 Auto_Scroll
    - 🟩 Control_Panel
  - 🟩 Controlling
    - 🟩 Accelerated
    - 🟩 Synchronized
    - 🟩 Controller_Type
      - 🟩 Distance_Controller
      - 🟥 Default_Controller
  - 🟩 Settings
    - 🟩 Number_Of_Elevators
    - 🟩 Number_Of_Floors
    - 🟩 Elevator_Speed
      - 🟥 Slow
      - 🟩 Medium
      - 🟥 Fast

(c) Premium elevator

Figure 7.8: Configurations of the three product instances generated from the PL_SimElevator SPL project.

7.4.2.1 Fault injection

As a means to measure the effectiveness of the test cases designed, we emulated faults in the source code, by considering the types of faults listed in Table 6.6. They represent some of the commonly occurring problems when implementing variability using Java constructs.

The modules selected to this experiment represented suitable locations for emulating some of the desired types of faults. It means a fault was identified and classified according to an operator library of type of faults. Indeed, the faults should only be emulated where they could actually exist in the compiled code, considering the logic of programming structures.

It is worth mentioning that we avoided modifying statements representing syntax errors. With the increasing power of IDEs, syntax errors cannnot be considered as realistic faults any longer, since the compilers easily point these flaws and they would be corrected before the program execution.

Fault emulation was manually performed. Hence, we left out modifications in the Java bytecode files, although we believe such capability makes it possible to emulate some other desired faults.

7.4.3 Analysis and interpretation

This section presents the statistical analysis of the gathered data. Each question stated in the experiment definition is answered, along with some discussion on the presented results.

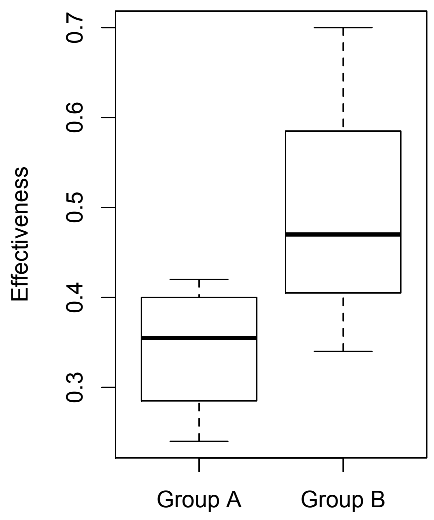
The data were collected from the test cases implemented by the participants, and the defect log report they filled out during the experiment session. Each defect found in each subject's defect log was analyzed to check whether they represented an actual defect. False positives were discarded from the final analysis.

7.4.3.1 Does the use of fault models lead to best variability testing results? (RQ1)

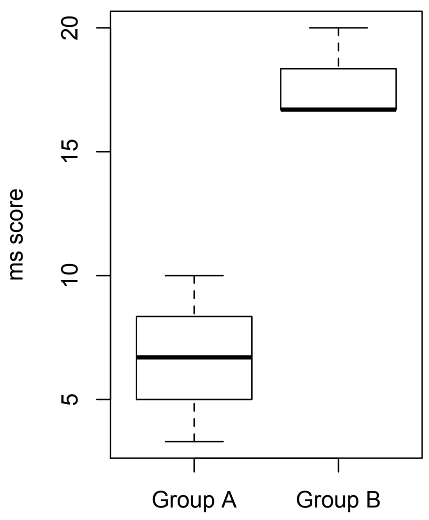
By applying a randomized design, we assigned the participants to two different groups: *Group A* (ad-hoc testing): **P3, P4, P5, P6**; *Group B* (fault model support): **P1, P2, P7**. As we observed an expertise homogeneity, and especially for the small number of participants, we did not make effort to achieve an *ideal* calibration between the groups.

Table 7.4 shows raw results for each participant, in terms of TC (number of designed test cases), and DF (number of defects found). Based on such values, it is possible to observe a higher *mean* value for the **TCE** calculations for the group which employed the notion of fault models for test case design: 0.504 with a *sd* of 0.182, against 0.343, with a *sd* of 0.078. Figure

7.9a shows a boxplot plotting the effectiveness reached by each group.



(a) TCE

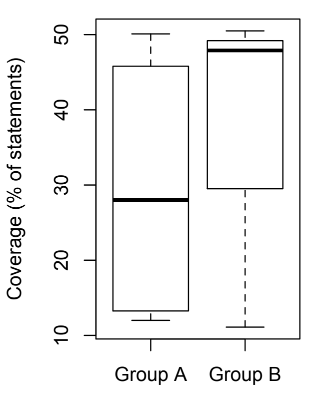


(b) *ms score*

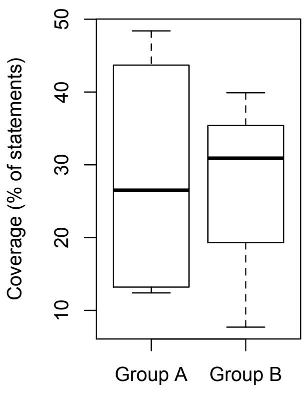
Figure 7.9: Boxplots for TCE measures and the *msscore*.

Furthermore, Table 7.6 shows the *statement* coverage values (%) for each product instance. When the participant designed/implemented a test case, he should have in mind the reuse potential of it. That is, a test case aimed at a class *c* can be reused in any product configuration containing such a class. Hence, the values presented in such a Table serve as an attempt to demonstrate the reuse capability of the set of designed test cases.

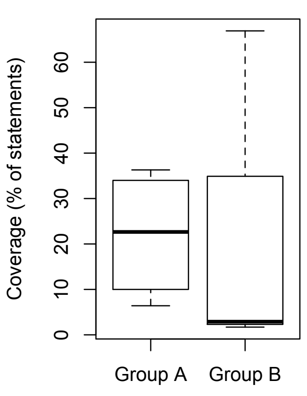
Figure 7.10 shows the results for the TCov measurement. Each boxplot shows the TCov values for each product instance. The descriptive statistics for each product is as follows. **Premium**: Group A (*mean*: 29.525, *sd*: 19.145), Group B (*mean*: 36.500, *sd*: 22.035); **Enhanced**: Group A (*mean*: 28.450, *sd*: 18.034), Group B (*mean*: 26.167, *sd*: 16.614); **Simple**: Group A (*mean*: 22.000, *sd*: 14.288), Group B (*mean*: 23.833, *sd*: 37.302). Group B reached better mean for the first two products, while Group A for the third one.



(a) Premium



(b) Enhanced



(c) Simple

Figure 7.10: Boxplots for TCov measures for each product instance.

7.4.3.2 Is the fault modeling approach helpful to uncover the faults that the fault models prescribed? (RQ2)

The ultimate goal of a fault modeling method is to reduce the number of test cases designed. Besides, the idea is to uncover the more defects with less test cases. In order to investigate the effects of using a fault modeling approach, we calculated the accuracy of the fault models at uncovering the fault types they were expected to. The last two lines from Table 7.4 show, respectively, the number of mutants killed and the ms score calculations. By analyzing the *mean* values of groups A and B, respectively 6.675 (*sd: 2.735*) and 17.800 (*sd: 1.905*), the group of participants using a fault model had better results. Figure 7.9b shows a boxplot plotting the *ms scores*.

7.4.3.3 Hypothesis testing

The hypotheses were tested using a standard paired t-test with a. 95% confidence level. We calculated the t-test to compare the two treatments against all metrics. Table 7.5 shows the results for all three metrics.

By analyzing the results, we observed a significant difference between the *ms score* of the both groups A and B. It is an indication that programmers when using fault models can achieve higher accuracy on finding the faults they are searching for, than when not employing such a strategy. Such an observation enable us to refute the null hypothesis H_{03}.

Table 7.4: Number of designed test cases and defects found

	Subject						
	P1	***P2***	***P7***	*P3*	*P4*	*P5*	*P6*
# of TC	10	15	29	9	12	8	25
# of DF	7	7	10	3	5	3	6
TCE	0.70	0.47	0.34	0.33	0.42	0.38	0.24
DM(P,R)	5	6	5	3	2	1	2
ms(P,R)	16.7	20.0	16.7	10.0	6.7	3.3	6.7

However, the same does not hold true for the remainder hypothesis. Although descriptive statistics showed better results for the group undertaking the task with the fault model support, we have not found evidence of a statistically significant difference for the hypotheses H_{01} and H_{02}, what prevent us to make any conclusions about such findings.

Table 7.5: t-test results.

metric	**t**	**df**	**p-value**	**n**
TCE	1.434	2.549	0.262	
TCOV-Premium	0.438	4.042	0.684	
TCOV-Enhanced	0.173	4.667	0.870	7
TCOV-Simple	0.081	2.445	0.942	
ms	6.338	4.999	0.001	

7.4.3.4 Threats to validity

In order to discuss the threats to the validity of this experimental study, we followed the advices on validity analysis and threats given by Wohlin *et al.* (2012).

Conclusion validity focuses on how sure we can be that the treatment we used in an experiment really is realted to the actual outcome we observed. Typically this concerns if there is statistically significant effect on the outcome. A general threat to conclusion validity in this experiment is the low number of samples, which may reduce the ability to reveal real patterns in the data. The analysis and interpretation of the results of this experiment was described using descriptive statistics, which are appropriate to the data type collected during the experiment.

Table 7.6: Test coverage (% of statements) per product instance

		Participants						
Tested classes		P1	P2	P7	P3	P4	P5	P6
gui.buttons	**PREMIUM**	33.0	50.2	15.9	35.6	33.4	20.6	21.8
gui.buttons.Default		22.5	59.0	0.0	24.4	22.5	0.0	28.3
gui.buttons.Authorize		58.7	58.7	0.0	58.7	58.7	0.0	0.0
gui.buttons.Emergency		53.6	63.6	0.0	53.6	56.4	67.3	67.3
gui.buttons.Arrow		32.4	33.5	42.8	37.4	32.4	28.8	0.0
simulator		71.8	78.0	0.0	24.9	21.5	0.0	0.0
simulator.controllers		74.1	33.0	0.0	80.2	57.6	0.0	0.0
simulator.model		86.1	65.3	47.5	86.1	80.2	0.0	47.5
Total Coverage		***50.5***	***47.9***	***11.1***	***50.1***	***41.5***	***12.0***	***14.5***
gui.buttons	**ENHANCED**	29.4	23.3	13.0	35.6	29.8	20.6	20.5
gui.buttons.Default		22.5	18.7	0.0	24.4	22.5	0.0	28.3
gui.buttons.Authorize		0.0	0.0	0.0	0.0	0.0	0.0	0.0
gui.buttons.Emergency		53.6	37.4	0.0	53.6	56.4	67.3	56.5
gui.buttons.Arrow		32.4	27.7	37.1	37.4	32.4	28.8	0.0
simulator		15.6	78.0	0.0	24.9	21.5	0.0	0.0
simulator.controllers		64.0	30.1	0.0	80.2	57.6	0.0	0.0
simulator.model		73.7	58.4	0.0	86.1	80.2	0.0	47.5
Total Coverage		***39.9***	***30.9***	***7.7***	***48.4***	***39.0***	***12.4***	***14.0***
gui.buttons	**SIMPLE**	0.0	62.7	0.0	7.4	35.3	16.5	36.4
gui.buttons.Default		0.0	55.4	0.0	0.0	0.0	0.0	27.7
gui.buttons.Authorize		0.0	74.3	0.0	58.7	0.0	0.0	0.0
gui.buttons.Emergency		0.0	75.4	0.0	0.0	51.5	73.3	73.3
gui.buttons.Arrow		0.0	0.0	0.0	0.0	0.0	0.0	0.0
simulator		0.0	0.0	0.0	0.0	0.0	0.0	0.0
simulator.controllers		0.0	32.0	0.0	10.2	0.0	0.0	0.0
simulator.model		27.7	42.6	47.5	0.0	64.4	0.0	47.5
Total Coverage		***1.7***	***66.9***	***2.9***	***6.4***	***36.3***	***13.6***	***31.7***

Internal validity concerns matters that may affect the independent variable with respect to causality, without the researcher's knowledge. There is a main threat to internal validity in this experiment, instrumentation. The SPL project used in this experiment might have affected the results. The source code might contain an additional set of issues, other than those injected (expected to be uncovered by the tests), so that those were also considered as valid defects. There are other likely threats we consider as small, such as the maturation effect. As each participant was allocated to a single treatment, hence there is not threat of maturation in this experiment. Regarding selection, as we carried out the experiment inside a software company, with software engineers we were not aware of, prior to the study, we mitigated such a likely threat.

Construct validity is concerned with the relation between theory and observation. It concerns generalisation of the exepriment result to concept or theory behind the experiment. The construct validity include two main threats. The first threat is that the measurements as defined may not be appropriate. Besides, the pre-chosen fault models may not be representative or good enough for the scenario under testing. These may limit the scope for the conclusions made to the use of the fault modeling approach.

External validity is concerned with whether we can generalize the results outside the scope of our study. Despite we counted on experienced software engineers as participants in this experimental study, the small number of subjects may limit generalisations of findings. However, the setting employed in the study resembled a real testing situation, what might strengthen the inferences about the use of fault models for variability testing.

7.5 Chapter summary

As far as we know, the initial attempt to model the faults of an SPL was presented by McGregor (2008). In such a study, the author proposed an initial version of a fault model for SPL engineering. The model describes the set of known defects that can result from the application of a SPL strategy, which leads to the test points, in the whole life cycle. The author argued that a whole life cycle must comprise several test models, each matching a specific phase. Based on a motivating example, a small SPL, he leveraged a set of common problems that can be faced when employing an SPL strategy.

As earlier discussed in this document, the role of a fault model is to establish a prioritization on test case design, i.e., the main idea behind a fault model is to design test cases that may anticipate the likely faults a source code should contains. Besides, it is possible to use fault models to assess existing test suites.

Roughly speaking, fault modeling entails an increase in fault detection rates. In order to build such a model, it is necessary to have prior knowledge about the commonly occurring faults that are likely to be present in the implementation. Therefore, the initial step towards building a fault model for variability testing is to know which faults are common in variability implementation, considering the problems emerging during implementation, as well as the set of problems propagated from preceding phases.

This Chapter presented the approach we believe SPL engineers can achieve better results in terms of fault detection. It describes how a fault modeling approach can be used in both perspectives, namely designing more effective test cases, and assessing existing ones. While even this approach can not guarantee the absence of a specific type of fault, it might be used in assurance arguments that specific procedures have been used to search for specific faults.

8 Conclusions

SPL engineering has been applied in software industry to improve the quality of delivered products, and reduce the overall development effort. Convincing results have been achieved in a range of domains. Variability management is a key element in SPL engineering, as it controls a myriad of products that share a set of functionalities and differ in another, enabling meeting particular customer and/or market demands.

Software testing, as the most widely used approach for improving software quality in practice, plays an important role for SPL engineering. The research in SPL testing has been growing increasingly, addressing the inherent complex issues surrounding the field.

In this work, we proposed a contribution to SPL testing. We carried out in-depth analyses of existing literature in this research field, and noticed a lack of evidence about how to handle variability testing from a source code perspective. Hence, apart from most existing SPL testing techniques, we are focused on handling variability issues emerging at source code level. That is, instead of handling variability at a higher level of abstraction, such as in feature models, we cope with variability testing at the lowest level of abstraction, where the implementation mechanisms are used consistently.

The contribution consists of improving SPL testing by employing the concept of fault models. Fault models, widely used in the development of physical devices and systems, such as the design of integrated circuits and chips, is said to be an effective means of reducing effort in software testing while improving, or at least not harming, the fault detection capabilities of the test sets. Fault models can be merged into a SPL process to improve test effectiveness.

We designed an approach for using fault models to support variability testing, from two main perspectives, namely *test assessment*, which focuses on the evaluation of the effectiveness of existing test suites, and test design, which aims to aid the construction of test sets, by focusing on fault-prone elements. We think of fault models as a strategy that provide software developers with assistance, for anticipating fault-prone scenarios, and thus fostering improvements in test effectiveness.

As a means to evaluate the capability of the proposed fault modeling approach, we carried out an experimental study, in which experienced software engineers could make use a set of fault models to design developer tests (unit and integration tests) for an SPL project. The evaluation showed promising results. However, we understand that the approach can be improved in many ways. In Section 8.1 we discuss future research directions. Next, in Section 8.2 we discuss the main related work. Finally, we sketch the main contributions achieved so far with this investigation in Section 8.3.

8.1 Future work

We have suggested the merge of fault models in the SPL process, an as improvement of testing in such a field. However, there remain many improvements to explore. As future work, we plan to investigate aspects of automation, addressing challenges of generating desirable test inputs and checking the behavior of the features under test. In addition, we plan to carry out empirical studies to better understand the role of fault models in other domains and scenarios. We next list the main further research directions that arise from the work presented here:

- *Automated generation of fault models.* Building a set of faults models would encompass the automated analysis of historical data, considering a range of sources, such as previous versions of the project under analysis, previous projects from the same organizations, project from similiar domains. Automating the construction of fault models is encouraged. However, we understand this might not be a straightforward task, and might demand knowledge from several fields. Hence, our next steps include the investigation of *data mining* techniques, as a means to establish affordable and systematic means to retrieve and analyze past historical data. Besides, we plan to explore *machine learning* techniques, to support automated defect classifications, and increase precision as well. In this branch of research, it is possible to explore the capabilities of a range of techniques, to analyze which could fit together, to achieve better results. In a recent partnership, we have explored the use of *support vector machines* to aid bug triage (Cavalcanti *et al.*, 2014). Future research direction includes exploring such a technique in the context of our proposed fault modeling approach.

- *Automated selection of fault models.* In addition to the preceding research direction stated, establishing a means to automatically select fault models, by taking into account data about the application under test might lead to reductions in effort. Taking a large-scale SPL as an example, it is possible that, along the development of such a project, several fault models can be built. Hence, it is important to investigate how to better associate a given fault model to a given scenario. Besides, the choice of a fault model could influence the outcome of the assessments. For instance, whether one fault model can be shown to detect more errors with less effort, it is advisable to choose this model over less efficient models for systematic errors detection. If, on the other hand, the evaluations of two distinct models are highly similar, the less costly model (in terms of implementation, setup and run time effort) can be justifiably substituted for the more expensive one. Furthermore, a comparative evaluation of fault models could provide guidance on the selection of fault models.

- *Systematic analysis of past historical data.* The goal should be a complete list of faults, though we do not make this claim, due to the NP-hardness nature of fault analysis (Hu *et al.*, 2013). A big issue is that fault categorization task often involves much manual effort, and time-consuming. Achieving good results with repository mining, with data from, e.g., open source projects, would be very hard as many test-related activities do not leave traces in the repositories. Besides, it is not easy to find large-scale SPL projects available, so that we could make use open data. We are not aware of any effective and general strategy to automatically categorize software faults. There are some initiatives, such as (Huang *et al.*, 2011; Thung *et al.*, 2012), which yields good and reliable results, but comprising a rather constrained scenario, with some threats that limits some inferences concerning generalizability potential. In this context, we plan to investigate how such initiatives could work in other scenarios, i.e., count on empirical evaluations of real-world and large-scale ones. Besides, we intend to apply algorithms other than those used in such investigations, to analyze their effectiveness.

- *Fault models in the SDLC.* In the context of the SDLC, it is possible to build fault models for every SPL phase. A fault modeling approach to be considered useful and complete should encompass all phases. Hence, a number of fault models can be built that anticipate the likely problems. Although there is a number of studies investigating how to better manage variability at the initial SPL phases, not every problem, especially regarding different application domains, has been tackled yet. Hence, such a gap could be taken as a starting point towards establishing a complete fault modeling approach for SPL engineering.

- *Empirical evaluations.* This thesis presented the definition, planning, operation, analysis, interpretation, presentation and packaging of two experimental studies. However, new studies in different contexts, including more subjects and other domains are still necessary in order to carry out more evaluations and calibrate the experimental plan.

- *Measurement.* We employed mutation testing as a technique to measure the effectiveness of existing test sets. Although some of the mistakes a developer usually makes can be simulated in a mutation model, we have to understand that mutation testing is not widely used in industry, and as such many of the results obtained may not reflect the reality, what hinders generalizations. Hence, we believe the approach should consider empirical evalutions in real-world scenarios, so as to obtain more accurate and reliable measures.

8.2 Related work

In literature, there are several proposals to improve testing in SPL engineering. We systematically analyzed the literature on the topic (Neto *et al.*, 2011a; Machado *et al.*, 2014b), as reported in Chapter 4. The most important publications in the field, given their acknowledgement by the research community, has been discussed in this thesis, as well as in both papers. However, the key difference between this work and other is the attempt to take past historical data on variability implementation issues, as a key driver for test prioritization. Most research in the field is concerned about either proposing means to validate the combination of features to drive product generation, or to design tests based on specifications, either formal or not.

8.3 Main contributions

We earlier described in the Section 1.4 the main contributions expected from this investigation. Some of the results have been already published. Next, we list the set of papers resulting from this investigation:

- Machado, I. C., Neto, P. A. M. S., Almeida, E. S., and Meira, S. R. L. RiPLE-TE: A Process for Testing Software Product Lines. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering*, SEKE, pages 711–716, Miami Beach, FL, USA. KSI. (Machado *et al.*, 2011)

- Machado, I. C. Towards a reasoning framework for software product line testing. In *16th International Software Product Line Conference*, SPLC, *Doctoral Symposium*, pages

229–232. ACM. (Machado, 2012)

- Machado, I. C., Almeida, E. S., Gomes, G. S. S., Neto, P. A. M. S., Novais, R. L., and Neto, M. G. M. A preliminary study on the effects of working with a testing process in software product line projects. In *IX Experimental Software Engineering Latin American Workshop*, ESELAW, Buenos Aires, Argentina. (Machado *et al.*, 2012a)

- Machado, I. C., McGregor, J. D., and Almeida, E. S. (2012). Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes*, **37**(6), 1–8. (Machado *et al.*, 2012b)

- Machado, I. C., Neto, P. A. M. S., and Almeida, E. S. Towards an integration testing approach for software product lines. In *13th IEEE International Conference on Information Reuse and Integration (IRI), 2012* , pages 616–623. (Machado *et al.*, 2012c)

- Machado, I. C., Santos, A. R., Cavalcanti, Y. C., Trzan, E. G., Souza, M. M., and Almeida, E. S. Low-level variability support for web-based software product lines. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, pages 15:1–15:8, New York, NY, USA. ACM. (Machado *et al.*, 2014a)

- Machado, I. C., McGregor, J. D., Cavalcanti, Y. C., and Almeida, E. S. (2014b). On strategies for testing software product lines: A systematic literature review. *Information & Software Technology*, **56**(10), 1183–1199. (Machado *et al.*, 2014b)

Other important publications from this thesis, in which there are other main(first) authors, but still hold true importance for this work:

- Neto, P. A. M. S., Machado, I. C., McGregor, J. D., Almeida, E. S., and Meira, S. R. L. (2011a). A systematic mapping study of software product lines testing. *Information and Software Technology*, **53**(5), 407–423. (Neto *et al.*, 2011a)

- Neto, P. A. M. S., Runeson, P., Machado, I. C., Almeida, E. S., Meira, S. R. L., and Engström, E. (2011b). Testing software product lines. *IEEE Software*, **28**(5), 16–20. (Neto *et al.*, 2011b)

- Wohlin, C., Runeson, P., Neto, P. A. M. S., Engström, E., Machado, I. C., and Almeida, E. S. (2013). On the reliability of mapping studies in software engineering. *Journal of Systems and Software*, **86**(10), 2594 – 2610. (Wohlin *et al.*, 2013)

- Cavalcanti, Y. C., Machado, I. C., Neto, P. A. M. S., Almeida, E. S., and Meira, S. R. L. (2014). Combining rule-based and information retrieval techniques to assign software change requests. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Västeras, Sweden. ACM. (Cavalcanti *et al.*, 2014)

References

Abdi, H. and Williams, L. J. (2010). Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, **2**(4), 433–459.

Aguillo, I. F. (2012). Is google scholar useful for bibliometrics? a webometric analysis. *Scientometrics*, **91**(2), 343–351.

Al-Hajjaji, M., Thüm, T., Meinicke, J., Lochau, M., and Saake, G. (2014). Similarity-based prioritization in software product-line testing. In *FOSD Meeting 2014, Schloss Dagstuhl*.

Alford, D., Sackett, P., and Nelder, G. (2000). Mass customisation - an automotive perspective. *International Journal of Production Economics*, **65**(1), 99 – 110.

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., and Lucena, C. (2006). Refactoring product lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE, pages 201–210, New York, NY, USA. ACM.

Alves, V., Calheiros, F., Nepomuceno, V., Menezes, A., Soares, S., and Borba, P. (2008). Flip: Managing software product line extraction and reaction with aspects. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC, Washington, DC, USA. IEEE Computer Society.

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

Ammann, P. and Offutt, J. (2008). *Introduction to software testing*. Cambridge University Press.

Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.-C., Rummler, A., and Sousa, A. (2010). A model-driven traceability framework for software product lines. *Software and System Modeling*, **9**(4), 427–451.

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, **28**(10), 970–983.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated.

Arisholm, E., Briand, L., and Fuglerud, M. (2007). Data mining techniques for building fault-proneness models in telecom java software. In *The 18th IEEE International Symposium on Software Reliability*, ISSRE, pages 215–224.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, **1**(1), 11–33.

Babar, M., Chen, L., and Shull, F. (2010). Managing variability in software product lines. *Software, IEEE*, **27**(3), 89 –91, 94.

Babu, C. and Krishnan, H. R. (2009). Fault model and test-case generation for the composition of aspects. *SIGSOFT Software Engineering Notes*, **34**(1), 1–6.

Barbey, S. and Strohmeier, A. (1994). The problematics of testing object-oriented software. In *Proceedings of the Second Conference on Software Quality Management*, pages 411–426. Comp. Mech. Publications.

Basili, V. R. and Hutchens, D. H. (1983). An empirical study of a syntactic complexity family. *IEEE Transactions on Software Engineering*, **9**(6), 664–672.

Basili, V. R. and Perricone, B. T. (1984). Software errors and complexity: An empirical investigation. *Communications of the ACM*, **27**(1), 42–52.

Basili, V. R. and Selby, R. W. (1987). Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, **SE-13**(12), 1278–1296.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). Goal question metric paradigm. In *Encyclopedia of Software Engineering*, volume 2, pages 528–532. Wiley.

Bastos, J. F., Neto, P. A. M. S., Almeida, E. S., and Meira, S. R. L. (2011). Adopting software product lines: A systematic mapping study. In *15th International Conference on Evaluation and Assessment in Software Engineering*, EASE, pages 11–20. IET.

Batory, D. (2003). A tutorial on feature oriented programming and product-lines. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE, pages 753–754.

Batory, D., Höfner, P., and Kim, J. (2011). Feature interactions, products, and composition. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component engineering*, GPCE, pages 13–22, New York, NY, USA. ACM.

Batory, D. S. (2006). Feature modularity in software product lines. In *10th International Conference on Software Product Lines*, SPLC, page 230. IEEE Computer Society.

Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, **35**(6), 615–636.

Bengtsson, T. and Kumar, S. (2005). A survey of high level test generation methodologies and fault models. Technical Report Research Report 04:5, School of Engineering, Jönköping University, Jönköping, Sweden.

Beuche, D. (2012). Modeling and building software product lines with pure::variants. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC, pages 255–255, New York, NY, USA. ACM.

Beuche, D. (2013). Modeling and building product lines with pure::variants. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC Workshops, pages 147–149, Tokyo, Japan. ACM.

Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Boehm, B. and Basili, V. R. (2001). Software defect reduction top 10 list. *IEEE Computer*, **34**(1), 135–137.

Boehm, B. W. (1979). Guidelines for verifying and validating software requirements and design specifications. In P. A. Samet, editor, *Euro IFIP 79*, pages 711–719. North Holland.

Boehm, B. W. and Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, **14**(10), 1462–1477.

Bosch, J. and Capilla, R. (2013). *Systems and Software Variability Management*, chapter Variability Implementation, pages 75–86. Springer-Verlag Berlin Heidelberg.

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J. H., and Pohl, K. (2002). Variability issues in software product lines. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 13–21, London, UK, UK. Springer-Verlag.

Botaschanjan, J. and Hummel, B. (2009). Specifying the worst case: orthogonal modeling of hardware errors. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ISSTA, pages 273–284, New York, NY, USA. ACM.

Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., and Khalil, M. (2007). Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, **80**(4), 571–583.

Briand, L. C. and Wüst, J. (2002). Empirical studies of quality models in object-oriented systems. volume 56 of *Advances in Computers*, pages 97 – 166. Elsevier.

Briand, L. C., Basili, V. R., and Thomas, W. M. (1992). A pattern recognition approach for software engineering data analysis. *IEEE Transactions on Software Engineering*, **18**(11), 931–942.

Burger, S., Hummel, O., and Heinisch, M. (2013). Airbus cabin software. *IEEE Software*, **30**(1), 21–25.

Burnstein, I. (2003). *Practical Software Testing: A Process-Oriented Approach*. Springer-Verlag New York, Inc.

Catal, C. and Mishra, D. (2012). Test case prioritization: a systematic mapping study. *Software Quality Journal*, pages 1–34.

Cavalcanti, R. O., Almeida, E. S., and Meira, S. R. L. (2011a). Extending the riple-de process with quality attribute variability realization. In *7th International Conference on the Quality of Software Architectures, QoSA 2011 and 2nd International Symposium on Architecting Critical Systems, ISARCS 2011*, pages 159–164. ACM.

Cavalcanti, Y. C., Machado, I. C., Neto, P. A. M. S., Lobato, L. L., Almeida, E. S., and Meira, S. R. L. (2011b). Towards metamodel support for variability and traceability in software product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS, pages 49–57, Namur, Belgium. ACM.

Cavalcanti, Y. C., Machado, I. C., Neto, P. A. M. S., Almeida, E. S., and Meira, S. R. L. (2014). Combining rule-based and information retrieval techniques to assign software change requests. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Västeras, Sweden. ACM.

Chen, L. and Babar, M. A. (2011). A systematic review of evaluation of variability management approaches in software product lines. *Information & Software Technology*, **53**(4), 344–362.

Chernak, Y. (2001). Validating and improving test-case effectiveness. *IEEE Software*, **18**(1), 81–86.

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., and Wong, M.-Y. (1992). Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, **18**(11), 943–956.

Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. (2001). An empirical study of operating systems errors. *SIGOPS Operating Systems Review*, **35**(5), 73–88.

Clements, P. and McGregor, J. D. (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.

Corriveau, J.-P., Bashardoust, S., and Radonjic, V. (2011). Requirements verification in the presence of variability. In *Model-Driven Requirements Engineering Workshop*, MoDRE, pages 74–78. IEEE Computer Society.

Craig, R. D. and Jaskiel, S. P. (2002). *Systematic Software Testing*. Artech House, Inc., Norwood, MA, USA.

Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, **34**(2), 56–78.

Cruzes, D. S. and Dybä, T. (2011). Research synthesis in software engineering: A tertiary study. *Information & Software Technology*, **53**(5), 440 – 455.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Czarnecki, K., Helsen, S., and Eisenecker, U. W. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, **10**(1), 7–29.

Dashofy, E. M., van der Hoek, A., and Taylor, R. N. (2002). An infrastructure for the rapid development of xml-based architecture description languages. In *Proceedings of the 24th*

International Conference on Software Engineering, ICSE ’02, pages 266–276, New York, NY, USA. ACM.

Deelstra, S., Sinnema, M., and Bosch, J. (2009). Variability assessment in software product families. *Information & Software Technology*, **51**(1), 195–218.

Delamaro, M. E., Maldonado, J. C., and Jino, M. (2007). *Introdução ao teste de software*. Elsevier, Rio de Janeiro.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, **11**(4), 34–41.

Denaro, G., Morasca, S., and Pezzè, M. (2002). Deriving models of software fault-proneness. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE, pages 361–368, New York, NY, USA. ACM.

Dhungana, D., Grünbacher, P., and Rabiser, R. (2011). The dopler meta-tool for decision-oriented variability modeling: A multiple case study. *Automated Software Engineering*, **18**(1), 77–114.

Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2011). Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice*.

Dordowsky, F., Bridges, R., and Tschope, H. (2011). Implementing a software product line for a complex avionics system. In *Proceedings of the 15th International Conference on Software Product Lines*, SPLC, pages 241–250, Munich, Germany. ACM.

Dreyfus, S. E. and Dreyfus, H. L. (1980). A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition. Technical report, University of California, Berkeley.

Dybå, T. and Dingsøyr, T. (2008a). Empirical studies of agile software development: A systematic review. *Information & Software Technology*, **50**(9–10), 833–859.

Dybå, T. and Dingsøyr, T. (2008b). Strength of evidence in systematic reviews in software engineering. In *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 178–187. ACM.

Dybå, T., Kitchenham, B. A., and Jorgensen, M. (2005). Evidence-based software engineering for practitioners. *IEEE Software*, **22**(1), 58–65.

Elish, K. O. and Elish, M. O. (2008). Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, **81**(5), 649–660.

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

Engström, E., Runeson, P., and Skoglund, M. (2010). A systematic review on regression test selection techniques. *Information & Software Technology*, **52**(1), 14–30.

Etxeberria, L. and Sagardui, G. (2005). Product-line architecture: New issues for evaluation. In H. Obbink and K. Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 174–185. Springer Berlin Heidelberg.

Etxeberria, L., Sagardui, G., and Belategi, L. (2008). Quality aware software product line engineering. *Journal of the Brazilian Computer Society*, **14**, 57 – 69.

Fazal-Amin, Mahmood, A. K., and Oxley, A. (2011). An analysis of object oriented variability implementation mechanisms. *SIGSOFT Software Engineering Notes*, **36**(1), 1–4.

Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., and Saake, G. (2013). Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, **18**(4), 699–745.

Fenton, N. E. and Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, **25**(5), 675–689.

Frankl, P. G. and Iakounenko, O. (1998). Further empirical studies of test effectiveness. *SIGSOFT Software Engineering Notes*, **23**(6), 153–162.

Fraser, G. and Zeller, A. (2012). Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, **38**(2), 278 –292.

Freeman, S., Mackinnon, T., Pryce, N., and Walnes, J. (2004). Mock roles, objects. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA, pages 236–246, Vancouver, BC, CANADA. ACM.

Fritsch, C., Lehn, A., and Strohm, T. (2002). Evaluating variability implementation mechanisms. In *International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing*, PLEES, pages 59–64.

Gacek, C. and Anastasopoules, M. (2001). Implementing product line variabilities. *SIGSOFT Software Engineering Notes*, **26**(3), 109–117.

Ganesan, D., Lindvall, M., McComas, D., Bartholomew, M., Slegel, S., Medina, B., Krikhaar, R., and Verhoef, C. (2012). An analysis of unit tests of a flight software product line. *Science of Computer Programming*.

Ghanam, Y., Park, S., and Maurer, F. (2008). A test-driven approach to establishing & managing agile product lines. In *Proceedings of the 5th International Workshop on Software Product Line Testing*, SPLiT, pages 46–51, Limerick, Ireland. Hochschule Manheim CS Reports.

Gill, G. K. and Kemerer, C. F. (1991). Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, **17**(12), 1284–1288.

Gittens, M., Kim, Y., and Godwin, D. (July). The vital few versus the trivial many: examining the pareto principle for software. In *29th Annual International Computer Software and Applications Conference, 2005*, COMPSAC, pages 179–185.

Glass, R. L. (1981). Persistent software errors. *IEEE Transactions on Software Engineering*, **SE-7**(2), 162–168.

Glazer, A. and Sipper, M. (2008). Evolving an automatic defect classification tool. In *Proceedings of the 2008 Conference on Applications of Evolutionary Computing*, Evo, pages 194–203. Springer-Verlag, Naples, Italy.

Gomaa, H. (2005). *Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley.

Gomaa, H. and Eonsuk Shin, M. (2002). Multiple-view meta-modeling of software product lines. In *Eighth IEEE International Conference on Engineering of Complex Computer Systems*, pages 238–246.

Gonzalez-Sanchez, A., Piel, E., Gross, H.-G., and van Gemund, A. (2010). Prioritizing tests for software fault localization. In *10th International Conference on Quality Software*, QSIC, pages 42 –51.

Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. (2001). An empirical study of regression test selection techniques. *ACM Transaction on Software Engineering Methodology*, **10**(2), 184–208.

Greenfield, J. and Short, K. (2003). Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA, pages 16–27, New York, NY, USA. ACM.

Greiler, M., Deursen, A. v., and Storey, M.-A. (2012). Test confessions: a study of testing practices for plug-in systems. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE, pages 244–254, Piscataway, NJ, USA. IEEE Press.

Griss, M. L. (2000). Implementing product-line features by composing aspects. In *Proceedings of the First International Conference on Software Product Lines: Experiences and Research Directions*, SPLC, pages 271–289. Kluwer.

Guo, Y. and Sampath, S. (2008). Web application fault classification - an exploratory study. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 303–305, New York, NY, USA. ACM.

Haidry, S. and Miller, T. (2013). Using dependency structures for prioritization of functional test suites. *IEEE Transactions on Software Engineering*, **39**(2), 258–275.

Harrold, M. J. (1998). Architecture-based regression testing of evolving systems. In *International Worshop on Role of Architecture in Testing and Analysis*, ROSATEA 1998, pages 73–77, Marsala, Sicily, Italy.

Harrold, M. J. (2000). Testing: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE, pages 61–72, Limerick, Ireland. ACM.

Harzing, A.-W. (2013). A preliminary test of google scholar as a source for citation data: A longitudinal study of nobel prize winners. *Scientometrics*, **94**(3), 1057–1075.

Hayes, J. (1994). Testing of object-oriented programming systems (oops): A fault-based approach. In E. Bertino and S. Urban, editors, *Object-Oriented Methodologies and Systems*, volume 858 of *Lecture Notes in Computer Science*, pages 205–220. Springer Berlin Heidelberg.

Hervieu, A., Baudry, B., and Gotlieb, A. (2011). PACOGEN: Automatic generation of pairwise test configurations from feature models. In *22nd IEEE International Symposium on Software Reliability Engineering*, ISSRE, pages 120 –129, Hiroshima, Japan. IEEE Computer Society.

Hesse-Biber, S. N. (2010). *Mixed methods research: merging theory with practice*. The Guilford Press, New York, NY, USA.

Howden, W. E. (2011). Error-based software testing and analysis. In *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, COMPSACW, pages 161–167, Washington, DC, USA. IEEE Computer Society.

Hu, Y.-P., Zhang, F.-R., and Zhang, W.-Z. (2013). Hard fault analysis of trivium. *Information Sciences*, **229**, 142–158.

Huang, L., Ng, V., Persing, I., Geng, R., Bai, X., and Tian, J. (2011). Autoodc: Automated generation of orthogonal defect classifications. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 412–415, Washington, DC, USA. IEEE Computer Society.

IEEE (1998). IEEE Standard for Software Test Documentation.

IEEE (2009). IEEE Standard Classification for Software Anomalies.

Jansen, D. (2010). *The Electronic Design Automation Handbook*. Springer Publishing Company, Incorporated, 1st edition.

Jaring, M. and Bosch, J. (2004). Expressing product diversification – categorizing and classifying variability in software product family engineering. *International Journal of Software Engineering and Knowledge Engineering*, **14**(5), 449–470.

Jedlitschka, A., Ciolkowski, M., and Pfahl, D. (2008). Reporting experiments in software engineering. In F. Shull, J. Singer, and D. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer London.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, **37**(5), 649 –678.

Johansen, M., Haugen, O., and Fleurey, F. (2011). A survey of empirics of strategies for software product line testing. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW, pages 266–269, Berlin, Germany. IEEE Computer Society Press.

Jones, C. (2010). *Software Engineering Best Practices*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

Juran, J. M., Gryna, F. M., and Bingham, R. S. (1979). *Quality Control Handbook*. McGraw-Hill, New York, 3rd edition.

Juristo, N., Moreno, A. M., and Vegas, S. (2004). Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, **9**(1-2), 7–44.

Juzgado, N. J. and Vegas, S. (2003). Functional testing, structural testing, and code reading: What fault type do they each detect? In *ESERNET*, volume 2765 of *Lecture Notes in Computer Science*, pages 208–232. Springer.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute - Carnegie Mellon University, Pittsburgh, PA, USA.

Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE, pages 311–320, New York, NY, USA. ACM.

Kazman, R., Klein, M., and Clements, P. (2000). Atam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute.

Khoshgoftaar, T. M., Lanning, D. L., and Pandya, A. S. (2006). A comparative study of pattern recognition techniques for quality evaluation of telecommunications software. *IEEE Journal on Selected Areas in Communications*, **12**(2), 279–291.

Kim, S. D., Her, J. S., and Chang, S. H. (2005). A theoretical foundation of variability in component-based development. *Information & Software Technology*, **47**(10), 663–673.

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

Kitchenham, B. and Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report.

Kitchenham, B., Brereton, P., Budgen, D., Turner, M., Bailey, J., and Linkman, S. G. (2009). Systematic literature reviews in software engineering - a systematic literature review. *Information & Software Technology*, **51**(1), 7–15.

Kitchenham, B. A., Brereton, P., Turner, M., Niazi, M., Linkman, S. G., Pretorius, R., and Budgen, D. (2010). Refining the systematic literature review process - two participant-observer case studies. *Empirical Software Engineering*, **15**(6), 618–653.

Koch, I. and Naito, K. (2010). Prediction of multivariate responses with a selected number of principal components. *Computational Statistics & Data Analysis*, **54**(7), 1791 – 1807.

Kolb, R. and Muthig, D. (2003). Challenges in testing software product lines. In *Proceedings of the 7th Conference on Quality Engineering in Software Technology*, CONQUEST, pages 81–95. Fraunhofer Publica.

Krueger, C. (2004). Product line binding times: What you don't know can hurt you. In R. Nord, editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 305–306. Springer Berlin Heidelberg.

Krueger, C. and Clements, P. (2013). Systems and software product line engineering with biglever software gears. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC Workshops, pages 136–140, Tokyo, Japan. ACM.

Krueger, C. W. (2001). Easing the transition to software mass customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE, pages 282–293, London, UK. Springer-Verlag.

Kumaresh, S. and Baskaran, R. (2012). Experimental design on defect analysis in software process improvement. In *Recent Advances in Computing and Software Systems (RACSS), 2012 International Conference on*, pages 293–298.

Lamancha, B. P., Usaola, M. P., and Velthius, M. P. (2009). Software product line testing - a systematic review. In *Proceedings of the 4th International Conference on Software and Data Technologies*, ICSOFT, pages 23–30, Sofia, Bulgaria. INSTICC Press.

Le, D., Walkingshaw, E., and Erwig, M. (2011). #ifdef confirmed harmful: Promoting understandable software variation. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150.

Lewis, W. E. (2008). *Software Testing and Continuous Quality Improvement, Third Edition*. Auerbach Publications, Boston, MA, USA, 2nd edition.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE, pages 105–114, Cape Town, South Africa. ACM.

Linsbauer, L., Lopez-Herrejon, E. R., and Egyed, A. (2013). Recovering traceability between features and code in product variants. In *Proceedings of the 17th International Software Product Line Conference*, SPLC, pages 131–140, New York, NY, USA. ACM.

Lisboa, L. B., Garcia, V. C., Lucrédio, D., Almeida, E. S., Meira, S. R. L., and Fortes, R. P. M. (2010). A systematic review of domain analysis tools. *Information & Software Technology*, **52**(1), 1–13.

Lobato, L. L., Neto, P. A. M. S., Machado, I. C., Almeida, E. S., and Meira, S. R. L. (2012). Risk management in software product lines: An industrial case study. In *Software and System Process (ICSSP), 2012 International Conference on*, pages 180–189.

Machado, I. C. (2012). Towards a reasoning framework for software product line testing. In *16th International Software Product Line Conference*, SPLC, pages 229–232. ACM.

Machado, I. C., Neto, P. A. M. S., Almeida, E. S., and Meira, S. R. L. (2011). RiPLE-TE: A Process for Testing Software Product Lines. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering*, SEKE, pages 711–716, Miami Beach, FL, USA. KSI.

Machado, I. C., Almeida, E. S., Gomes, G. S. S., Neto, P. A. M. S., Novais, R. L., and Neto, M. G. M. (2012a). A preliminary study on the effects of working with a testing process in software product line projects. In *IX Experimental Software Engineering Latin American Workshop*, ESELAW, Buenos Aires, Argentina.

Machado, I. C., McGregor, J. D., and Almeida, E. S. (2012b). Strategies for testing products in software product lines. *ACM SIGSOFT Software Engineering Notes*, **37**(6), 1–8.

Machado, I. C., Neto, P. A. M. S., and Almeida, E. S. (2012c). Towards an integration testing approach for software product lines. In *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on*, pages 616–623.

Machado, I. C., Santos, A. R., Cavalcanti, Y. C., Trzan, E. G., Souza, M. M., and Almeida, E. S. (2014a). Low-level variability support for web-based software product lines. In *Proceedings* *of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, pages 15:1–15:8, New York, NY, USA. ACM.

Machado, I. C., McGregor, J. D., Cavalcanti, Y. C., and Almeida, E. S. (2014b). On strategies for testing software product lines: A systematic literature review. *Information & Software Technology*, **56**(10), 1183–1199.

Madeyski, L. (2010). *Test-Driven Development - An Empirical Evaluation of Agile Practice*. Springer Berlin Heidelberg.

Malaiya, Y. K. and Su, S. Y. H. (1982). A New Fault Model and Testing Technique for CMOS Devices. In *Proceedings International Test Conference*, ITC, pages 25–34, Philadelphia, PA, USA. IEEE Computer Society.

Marick, B. (1999). New models for test development. *Testing Foundations*.

Martin, E. and Xie, T. (2007). A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*, WWW, pages 667–676, Banff, Alberta, Canada. ACM.

Mathur, A. P. (2002). *Mutation Testing*. John Wiley Sons, Inc.

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. Technical Report VTT-PUBLICATIONS-456, VTT - Technical Research Centre of Ireland.

McGregor, J. D. (2001). Testing a software product line. Technical Report TR-022, CMU Software Engineering Institute.

McGregor, J. D. (2008). Toward a fault model for software product lines. In *Proceedings of the 12th International Conference on Software Product Lines*, SPLC, pages 157–162, Limerick, Ireland. IEEE Computer Society.

McGregor, J. D., Sodhani, P., and Madhavapeddi, S. (2004). Testing variability in a software product line. In *Proceedings of the International Workshop on Software Product Line Testing*, SPLiT, pages 45–50, Boston, MA, USA. Avaya Labs.

Medeiros, F., Ribeiro, M., and Gheyi, R. (2013). Investigating preprocessor-based syntax errors. In *Proceedings of the 12th international conference on Generative programming: concepts experiences*, GPCE, pages 75–84, Indianapolis, Indiana, USA. ACM.

Meek, B. and Siu, K. K. (1989). The effectiveness of error seeding. *ACM SIGPLAN Notices*, **24**(6), 81–89.

Mellegard, N., Staron, M., and Torner, F. (2012). A light-weight defect classification scheme for embedded automotive software and its initial evaluation. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 261–270.

Mendonca, M., Branco, M., and Cowan, D. (2009). S.p.l.o.t.: Software product lines online tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA, pages 761–762, New York, NY, USA. ACM.

Mohan, K. and Ramesh, B. (2007). Tracing variations in software product families. *Communications of the ACM*, **50**(12), 68–73.

Moraes, M. B. S., Almeida, E. S., and Meira, S. R. L. (2011). An agile scoping process for software product lines. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering*, SEKE, pages 717–722. Knowledge Systems Institute Graduate School.

Morell, L. (1990). A theory of fault-based testing. *IEEE Transactions on Software Engineering*, **16**(8), 844–857.

Myers, G. J., Badgett, T., and Sandler, C. (2011). *The Art of Software Testing*. John Wiley & Sons, 3rd edition.

Naik, K. and Tripathy, P. (2008). *Software Testing and Quality Assurance Theory and Practice*. John Wiley & Sons, Inc.

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.

Nath, S. K., Merkel, R., and Lau, M. F. (2012). On the improvement of a fault classification scheme with implications for white-box testing. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC, pages 1123–1130, New York, NY, USA. ACM.

Neiva, D. F. S., Almeida, F. C., Almeida, E. S., and Meira, S. R. L. (2010). A requirements engineering process for software product lines. In *11th IEEE International Conference on*

Information Reuse and Integration, IRI, pages 266–269, Las Vegas, USA. IEEE Systems, Man, and Cybernetics Society.

Neto, P. A. M. S., Machado, I. C., McGregor, J. D., Almeida, E. S., and Meira, S. R. L. (2011a). A systematic mapping study of software product lines testing. *Information & Software Technology*, **53**(5), 407–423.

Neto, P. A. M. S., Runeson, P., Machado, I. C., Almeida, E. S., Meira, S. R. L., and Engström, E. (2011b). Testing software product lines. *IEEE Software*, **28**(5), 16–20.

Neto, P. A. M. S., Machado, I. C., Cavalcanti, Y. C., Almeida, E. S., Garcia, V. C., and Meira, S. R. L. (2012). An experimental study to evaluate a SPL architecture regression testing approach. In *13th IEEE International Conference on Information Reuse and Integration*, IRI, pages 608–615.

Nie, K., Wang, G., and Zhang, L. (2012). On the relationship between preprocessor-based software variability and software defects. In *Proceedings of the 12th International Conference on Quality Software*, QSIC, Xi'an, Shaanxi, China. IEEE.

Offutt, J., Alexander, R., Wu, Y., Xiao, Q., and Hutchinson, C. (2001). A fault model for subtype inheritance and polymorphism. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, ISSRE, Washington, DC, USA. IEEE Computer Society.

Oliveira, T. H. B. (2009). RiPLE-EM: A Process to Manage Evolution in Software Product Lines.

Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., and Muller, G. (2011). Faults in linux: Ten years later. *SIGPLAN Not.*, **47**(4), 305–318.

Passos, L., Novakovic, M., Xiong, Y., Berger, T., Czarnecki, K., and Wąsowski, A. (2011). A study of non-boolean constraints in variability models of an embedded operating system. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC, pages 2:1–2:8, New York, NY, USA. ACM.

Paul, T. K. and Lau, M. F. (2012). Redefinition of fault classes in logic expressions. In *Proceedings of the 2012 12th International Conference on Quality Software*, QSIC, pages 144–153, Washington, DC, USA. IEEE Computer Society.

Perrouin, G., Sen, S., Klein, J., Baudry, B., and le Traon, Y. (2010). Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, ICST, pages 459–468, Paris, France. IEEE Computer Society.

Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., and le Traon, Y. (2011). Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, pages 1–39.

Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). Systematic mapping studies in software engineering. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE, Bari, Italy. University of Bari.

Petticrew, M. and Roberts, H. (2006). *Systematic Reviews in the Social Sciences: A practical guide*. Oxford: Blackwell Publishing.

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

Pohl, R., Lauenroth, K., and Pohl, K. (2011). A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 313–322, Washington, DC, USA. IEEE Computer Society.

Pomeranz, I. and Reddy, S. M. (2009). Selection of a fault model for fault diagnosis based on unique responses. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE, pages 994–999, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

Porter, A. A. and Selby, R. W. (1990). Empirically guided software development using metric-based classification trees. *IEEE Software*, **7**(2), 46–54.

Rapps, S. and Weyuker, E. (1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, **SE-11**(4), 367–375.

Rashid, A., Royer, J.-C., and Rummler, A. (2011). *Aspect-Oriented, Model-Driven Software Product Lines - The AMPLE Way*, chapter 1. Cambridge University Press.

Reinecke, P., Wolter, K., and Malek, M. (2010). A survey on fault-models for qos studies of service-oriented systems. Technical Report B-2010-02, Freie Universität Berlin, Berlin, Germany.

Rincón, L., Giraldo, G., Mazo, R., and Salinesi, C. (2014). An ontological rule-based approach for analyzing dead and false optional features in feature models. *Electronic Notes in Theoretical Computer Science*, **302**, 111 – 132. Proceedings of the {XXXIX} Latin American Computing Conference (CLEI 2013).

Roos-Frantz, F. (2009). A preliminary comparison of formal properties on orthogonal variability model and feature models. In *Third International Workshop on Variability Modelling of Software-Intensive Systems,*, VaMoS, pages 121–126. Universität Duisburg-Essen, ICB Research Report.

Rothermel, G. and Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, **22**(8), 529–551.

Rothermel, G. and Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, **6**(2), 173–210.

Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, **27**(10), 929–948.

Salinesi, C. and Mazo, R. (2012). *Software Product Line - Advanced Topic*, chapter Defects in Product Line Models and How to Identify Them, pages 97–122. InTech.

Santos, W., Almeida, E., and de L Meira, S. (2012). Tirt: A traceability information retrieval tool for software product lines projects. In *38th EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA, pages 93–100.

Schaefer, I., Bettini, L., Damiani, F., and Tanzarella, N. (2010). Delta-oriented programming of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines*, SPLC, pages 77–91, Berlin, Heidelberg. Springer-Verlag.

Schmid, K. and John, I. (2004). A customizable approach to full lifecycle variability management. *Science of Computer Programming*, **53**(3), 259–284.

Schnieders, A. and Puhlmann, F. (2006). Variability mechanisms in e-business process families. In *9th International Conference on Business Information Systems*, BIS, pages 583–601, Klagenfurt, Austria. GI.

Seaman, C. B., Shull, F., Regardie, M., Elbert, D., Feldmann, R. L., Guo, Y., and Godfrey, S. (2008). Defect categorization: making use of a decade of widely varying historical data. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM, pages 149–157, New York, NY, USA. ACM.

Selby, R. W. and Porter, A. A. (1988). Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, **14**(12), 1743–1757.

Shaw, M. (2002). What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer*, **4**(1), 1–7.

Sinnema, M. and Deelstra, S. (2007). Classifying variability modeling techniques. *Information & Software Technology*, **49**(7), 717–739.

Sommerville, I. (2011). *Software Engineering*. Pearson Addison-Wesley, 9th edition.

Souza, I. S., Gomes, G. S. S., Neto, P. A. M. S., Machado, I. C., Almeida, E. S., and Meira, S. R. L. (2013). Evidence of software inspection on feature specification for software product lines. *Journal of Systems and Software*, **86**(5), 1172 – 1190.

Strecker, J. and Memon, A. M. (2012). Accounting for defect characteristics in evaluations of testing techniques. *ACM Transactions on Software Engineering Methodology*, **21**(3), 17:1–17:43.

Svahnberg, M., van Gurp, J., and Bosch, J. (2005). A taxonomy of variability realization techniques: Research articles. *Software Practice & Experience*, **35**(8), 705–754.

Tevanlinna, A., Taina, J., and Kauppinen, R. (2004). Product family testing: a survey. *ACM SIGSOFT Software Engineering Notes*, **29**(2), 12.

Thiel, S. (2002). On the definition of a framework for an architecting process supporting product family development. In F. Linden, editor, *Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 125–142. Springer Berlin Heidelberg.

Thörn, C. (2007). A quality model for evaluating feature models. In *11th International Conference on Software Product Lines, SPLC, Second Volume (Workshops)*, pages 184–190, Kyoto, Japan.

Thüm, T., Batory, D., and Kastner, C. (2009). Reasoning about edits to feature models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE, pages 254–264, Washington, DC, USA. IEEE Computer Society.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014). Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, **79**, 70–85.

Thummalapenta, S., Sinha, S., Singhania, N., and Chandra, S. (2012). Automating test automation. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE, pages 881–891, Piscataway, NJ, USA. IEEE Press.

Thung, F., Lo, D., and Jiang, L. (2012). Automatic defect categorization. *2013 20th Working Conference on Reverse Engineering (WCRE)*, **0**, 205–214.

Tischer, C., Muller, A., Mandl, T., and Krause, R. (2011). Experiences from a large scale software product line merger in the automotive domain. In *Proceedings of the 15th International Conference on Software Product Lines*, SPLC, pages 267–276, Munich, Germany. ACM.

Trujillo, S., Batory, D., and Diaz, O. (2007). Feature oriented model driven development: A case study for portlets. In *29th International Conference on Software Engineering*, ICSE, pages 44–53.

Tseng, M. M. and Jiao, J. (2001). Mass customization. In G. Salvendy, editor, *Handbook of industrial engineering: technology and operations management*, chapter 25, pages 684–709. John Wiley & Sons, Inc., 3rd edition.

Tsuchiya, R., Kato, T., Washizaki, H., Kawakami, M., Fukazawa, Y., and Yoshimura, K. (2013). Recovering traceability links between requirements and source code in the same series of software products. In *Proceedings of the 17th International Software Product Line Conference*, SPLC, pages 121–130, New York, NY, USA. ACM.

van der Linden, F., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action*. Springer-Verlag Berlin Heidelberg.

van Gurp, J., Bosch, J., and Svahnberg, M. (2001). On the notion of variability in software product lines. In *Working IEEE/IFIP Conference on Software Architecture*, pages 45–54.

Vegas, S. and Basili, V. R. (2005). A characterisation schema for software testing techniques. *Empirical Software Engineering*, **10**(4), 437–466.

Voelter, M. (2009). Variability patterns. In *EuroPLoP 2009: 14th Annual European Conference on Pattern Languages of Programming, Irsee, Germany, July 8-12*, volume 566 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Webster, J. and Watson, R. T. (2002). Analyzing the past to prepare for the future: writing a literature review. *MIS Quarterly*, **26**(2), xiii–xxiii.

Weiss, D. M., Clements, P. C., Kang, K., and Krueger, C. (2006). Software product line hall of fame. In *Proceedings of the 10th International on Software Product Line Conference*, SPLC, pages 237–, Baltimore, Maryland. IEEE Computer Society.

Winter, S., Sârbu, C., Suri, N., and Murphy, B. (2011). The impact of fault models on software robustness evaluations. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE, pages 51–60, New York, NY, USA. ACM.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., and Regnell, B. (2012). *Experimentation in Software Engineering*. Springer.

Wohlin, C., Runeson, P., Neto, P. A. M. S., Engström, E., Machado, I. C., and Almeida, E. S. (2013). On the reliability of mapping studies in software engineering. *Journal of Systems and Software*, **86**(10), 2594 – 2610.

Wood, M., Roper, M., Brooks, A., and Miller, J. (1997). Comparing and combining software defect detection techniques: A replicated empirical study. pages 262–277.

Wu, Y., Peng, X., and Zhao, W. (2011). Architecture evolution in software product line: an industrial case study. In *Proceedings of the 12th international conference on Top productivity through software reuse*, ICSR'11, pages 135–150, Berlin, Heidelberg. Springer-Verlag.

Xiao, X., Thummalapenta, S., and Xie, T. (2012). Advances on improving automation in developer testing. *Advances in Computers*, **85**, 165–212.

Yamamoto, M., Sugiyama, T., Murakami, H., and Sakaori, F. (2007). Correlation analysis of principal components from two populations. *Computational Statistics & Data Analysis*, **51**(9), 4707 – 4716.

Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, **22**(2), 67–120.

Zhang, H., Babar, M. A., and Tell, P. (2011). Identifying relevant studies in software engineering. *Information & Software Technology*, **53**(6), 625–637.

Appendices

A Systematic Literature Review - Primary Studies

This appendix lists the primary studies analyzed in the Systematic Review of Testing Strategies for SPL Engineering, earlier addressed in Chapter 4. We also lists the distribution of the final list of included studies (forty-two) per venue.

A.1 Venues manually searched

Table A.1: Venues subject to manual search

Journals
ACM TOSEM - Transactions on Software Engineering and Methodology
ASE - Automated Software Engineering
IEEE SW - Software
IEEE TSE - Transactions on Software Engineering
IET SW - Software
IST - Information & Software Technology
JSEP - Software: Evolution and Process
JSS - Systems & Software
SPE - Software: Practice and Experience

Continued . . .

SQJ - Software Quality Journal
STVR - Software Testing, Verification & Reliability
Conferences
AOSD - Aspect-Oriented Software Development
ASE - Automated Software Engineering
CSMR - Software Maintenance and Reengineering
ENASE - Evaluation of Novel Approaches to Software Engineering
FASE - Fundamental Approaches to Software Engineering
ICSE - Software Engineering
ICSM - Software Maintenance
ICSR - Software Reuse
ICST - Software Testing
ICTSS - Testing Software and Systems
ISSRE - Software Reliability Engineering
ISSTA - Software Testing and Analysis
MODELS - Model-Driven Engineering and Software Development
SEFM - Software Engineering and Formal Methods
SPLC - Software Product Line Conference
QSIC - Quality Software
Workshops
A-MOST - Advances in Model Based Testing
AST - Automation of Software Test
FOSD - Feature Oriented Software Development
PLEASE - Product Line Approaches in Software Engineering
SPLiT - Software Product Lines Testing
VaMoS - Variability Modelling of Software-Intensive Systems

A.2 Quality assessment results

See Table A.2.

Table A.2: Quality assessment.

Study	QC1	QC2	QC3	QC4	QC5	QC6	QC7	QC8	QC9	QC10	QC11	Score
P01	1	1	1	0	1	0	0	1	1	1	0	**7**
P02	1	1	1	0	0	0	0	0	1	1	1	**6**
P03	1	1	0	0	0	0	0	0	1	1	1	**5**
P04	1	1	1	0	0	0	0	1	1	1	1	**7**
P05	1	1	1	1	0	1	0	0	0	0	0	**5**
P06	1	1	0	0	0	0	0	0	0	0	0	**2**
P07	1	1	0	0	0	0	0	0	0	1	1	**4**
P08	1	1	1	1	0	0	0	0	0	1	1	**6**
P09	1	1	1	0	0	0	0	0	0	1	0	**4**
P10	1	1	0	0	0	0	0	0	0	0	0	**2**
P11	1	1	1	1	0	0	0	0	1	1	1	**7**
P12	1	1	0	0	0	0	0	0	0	0	0	**2**
P13	1	1	1	0	0	0	0	1	0	0	0	**4**
P14	1	1	1	0	0	0	0	0	0	0	0	**4**
P15	1	1	0	0	0	0	0	0	0	0	0	**2**
P16	1	1	1	1	0	0	0	0	0	1	0	**5**
P17	1	1	1	1	0	1	0	1	1	1	1	**9**
P18	1	1	1	0	0	1	0	0	1	1	0	**6**
P19	1	1	1	1	1	1	0	0	1	1	0	**8**
P20	1	1	1	0	0	0	0	0	1	1	0	**5**
P21	1	1	0	0	0	0	0	0	1	1	0	**4**
P22	1	1	0	1	1	1	0	0	1	1	0	**7**
P23	1	1	1	1	1	1	0	0	1	1	0	**8**
P24	1	1	1	1	0	1	0	0	1	1	0	**7**
P25	1	1	1	1	0	0	0	0	1	1	0	**6**
P26	1	1	0	0	0	0	0	0	0	1	0	**3**
P27	1	0	0	0	0	0	0	0	0	0	0	**1**
P28	1	1	1	1	0	1	0	0	1	1	0	**7**
P29	1	1	1	1	0	1	1	0	1	1	0	**8**
P30	1	1	0	0	0	1	0	0	1	1	0	**5**
P31	1	1	1	1	1	1	1	0	1	1	0	**9**
P32	1	1	0	0	0	0	0	0	1	1	0	**4**
P33	1	1	1	1	1	1	0	0	1	1	0	**8**
P34	1	1	1	1	0	1	0	0	1	1	0	**7**
P35	1	1	1	0	0	1	0	1	1	1	1	**8**
P36	1	1	1	1	1	1	1	0	1	1	0	**9**
P37	1	1	1	0	0	0	0	0	1	1	0	**5**
P38	1	1	1	1	1	0	0	0	0	1	0	**6**
P39	1	1	1	1	1	1	1	1	1	1	0	**10**
P40	1	1	1	1	0	1	1	0	1	1	0	**8**
P41	1	1	1	0	1	0	0	0	0	1	0	**5**
P42	1	1	1	1	1	1	1	0	1	1	0	**9**
P43	1	1	1	1	1	1	1	0	1	1	0	**9**
P44	1	1	1	1	0	0	0	0	0	1	0	**5**
P45	1	1	1	0	0	1	0	0	1	1	0	**6**
P46	1	1	1	1	1	1	1	0	1	1	0	**9**
P47	1	1	0	0	1	0	0	0	1	1	0	**5**
P48	1	1	0	0	0	0	0	0	0	1	0	**3**
P49	1	1	1	1	0	0	0	0	1	1	0	**6**

A.3 Primary studies

Table A.3: Selected primary studies.

ID	Title	Author(s)	Venue
P01	Testing software assets of framework-based product families during application engineering stage	J. Al-Dallal, P. G. Sorenson	JSW 3 (5): 11–25, 2008
P02	On extracting tests from a testable model in the context of domain engineering	S. Bashardoust-Tajali, J.-P. Corriveau	ICECCS'08: 98–107
P03	Product line use cases: Scenario-based specification and testing of requirements	A. Bertolino, A. Fantechi, S. Gnesi, G. Lami	Book Chapter: 425–445, 2006
P04	Towards generating acceptance tests for product lines	B. Geppert, J. J. Li, F. Roessler, D. M. Weiss	ICSR'04: 35–48
P05	An approach for selecting software product line instances for testing	T. Gustafsson	SPLC'07: 81–86
P06	Specification-based testing for software product lines	T. Kahsai, M. Roggenbach, B.-H. Schlingloff	SEFM'08: 149–158
P07	Testing variabilities in use case models	E. Kamsties, K. Pohl, S. Reis, A. Reuys	PFE'03: 6–18
P08	Reuse execution traces to reduce testing of product lines	J. J. Li, B. Geppert, F. Roessler, D. Weiss	SPLiT'07: 1–8
P09	A reuse technique for performance testing of software product lines	A. P. K. Reis, S.; Metzger	SPLiT'06: 5–10
P10	Specification based software product line testing: A case study	S. Mishra	CS&P'06: 243–254
P11	System testing of product lines: From requirements to test cases	C. Nebut, Y. Traon, J.-M. Jézéquel	Book Chapter: 447–477, 2006
P12	Model-based testing for applications derived from software product lines	E. M. Olimpiew, H. Gomaa	A-MOST'05: 1–7
P13	Customizable requirements-based test models for software product lines	E. M. Olimpiew, H. Gomaa	SPLiT'06: 17–22
P14	Reusable model-based testing	E. M. Olimpiew, H. Gomaa	ICSR'09: 76–85
P15	Towards software product line testing using story driven modeling	S. Oster, A. Schürr, I. Weisemöller	T.Report: 48–51, 2008
P16	Production-testing of embedded systems with aspects	J. Pesonen, M. Katara, T. Mikkonen	HVC'05: 90–102

Continued on next page...

Table A.3: *Continued.*

ID	Title	Author(s)	Venue
P17	The SCENTED method for testing software product lines	A. Reuys, S. Reis, E. Kamsties, K. Pohl	Book Chapter: 479–520, 2006
P18	Optimizing the selection of representative configurations in verification of evolving product lines of distributed embedded systems	K. Scheidemann	SPLC'06: 75–84
P19	Improving the testing and testability of software product lines	I. Cabral, M. B. Cohen, G. Rothermel	SPLC'10: 241–255
P20	Model-based coverage-driven test suite generation for software product lines	H. Cichos, S. Oster, M. Lochau, A. Schürr	MODELS'11: 425–439
P21	Goal-oriented test case selection and prioritization for product line feature models	A. Ensan et al.	ITNG'11: 291–298
P22	Linking feature models to code artifacts using executable acceptance tests	Y. Ghanam, F. Maurer	SPLC'10: 211–225
P23	PACOGEN: Automatic generation of pairwise test configurations from feature models	A. Hervieu, B. Baudry, A. Gotlieb	ISSRE'11: 120 – 129
P24	Properties of realistic feature models make combinatorial testing of product lines feasible	M. F. Johansen, Ø. Haugen, F. Fleurey	MODELS'11: 638–652
P25	Reducing combinatorics in testing product lines	C. H. P. Kim, D. S. Batory, S. Khurshid	AOSD'11: 57–68
P26	Testing product generation in software product lines using pairwise for features coverage	B. P. Lamancha, M. P. Usaola	ICTSS'10: 111–125
P27	A model based testing approach for model-driven development and software product lines	B. P. Lamancha, M. P. Usaola, M. P. Velthius	ENASE'10: 193–208
P28	Model-based pairwise testing for feature interaction coverage in software product line engineering	M. Lochau, S. Oster, U. Goltz, A. Schürr	SQJ 20(3): 567-604, 2012
P29	Automated incremental pairwise testing of software product lines	S. Oster, F. Markert, P. Ritter	SPLC'10: 196–210
P30	Pairwise feature-interaction testing for SPLs: potentials and limitations	S. Oster, M. Lochau, M. Zink, M. Grechanik	SPLC'11: 1–8
P31	Automated and scalable t-wise test case generation strategies for software product lines	G. Perrouin et al.	ICST'10: 459–468
P32	Modelling requirements to support testing of product lines	C. Robinson-Mallett et al.	A-MOST'10: 11–18
P33	Integration testing of software product lines using compositional symbolic execution	J. Shi, M. Cohen, M. Dwyer	FASE'12: 270–284
P34	A regression testing approach for software product lines architectures	P. A. M. S. Neto et al.	SBCARS'10: 41–50
P35	Avoiding redundant testing in application engineering	V. Stricker, A. Metzger, K. Pohl	SPLC'10: 226–240

Continued on next page...

Table A.3: *Continued.*

ID	Title	Author(s)	Venue
P36	Incremental test generation for software product lines	E. Uzuncaova, S. Khurshid, D. Batory	TSE 36 (3): 309–322, 2010
P37	Model-Driven Software Product Line Testing: An Integrated Approach	A. Schürr, S. Oster, F. Markert	SOFSEM'10: 112–131
P38	Combinatorial Testing for Feature Models Using CitLab	A. Calvagna, A. Gargantini, P. Vavassori	IWCT'13: 338–347
P39	Continuous test suite augmentation in software product lines	Z. Xu, M. B. Cohen, W. Motycka, G. Rothermel	SPLC'13: 52–61
P40	Evolutionary Search-Based Test Generation for Software Product Line Feature Models	F. Ensan, E. Bagheri, and D. Gašević	CAiSE'12: 613–628
P41	Incremental Model-Based Testing of Delta-Oriented Software Product Lines	M. Lochau, I. Schaefer, J. Kamischke, S. Lity	TAP'12: 67–82
P42	Minimizing test suites in software product lines using weight-based genetic algorithms	S. Wang, S. Ali, A. Gotlieb	GECCO'13: 1493–1500
P43	Multi-objective test generation for software product lines	C. Henard et al.	SPLC'13: 62–71
P44	Practical pairwise testing for software product lines	D. Marijan, A. Gotlieb, S. Sen, A. Hervieu	SPLC'13: 227–235
P45	Shared Execution for Efficiently Testing Product Lines	C. H. Kim, S. Khurshid, D. Batory	ISSRE'12: 221–230
P46	SPLat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems	C. H. Kim et al.	ESEC/FSE'13: 257–267
P47	Towards efficient SPL testing by variant reduction	M. Kowal, S. Schulze, I. Schaefer	VariComp'13: 1–6
P48	Requirements-based Delta-oriented SPL Testing	M. Dukaczewski et al.	PLEASE'13: 49–52
P49	Automated Test Case Selection Using Feature Model: An Industrial Case Study	S. Wang, A. Gotlieb, S. Ali, M. Liaaen	MODELS'13: 237–253

B Experimental Study - Materials

This appendix includes the support materials used in the empirical evaluation, as earlier discussed in Section 7.4. In Section B.1 we show the questionnaire, applied to gather background information of every participant in the case study. Next in Section B.2 we show raw data of background information.

B.1 Background questionnaire

A. **What is your overall experience with software development in practice? Check the bottom-most item that applies (by years of experience).**

Less than 1 year	1-3 years	3-6 years	6-9 years	More than 9 years
○	○	○	○	○

B. **What is your previous experience with software development in practice? Check the bottom-most item that applies (by area/years of experience).**

	No exp.	< 1 year	1-3 years	4-6 years	7-9 years	> 10+ years
Technical leader	○	○	○	○	○	○
Requirements engineering	○	○	○	○	○	○
Software design	○	○	○	○	○	○
Software implementation	○	○	○	○	○	○
Software testing	○	○	○	○	○	○
Quality assurance	○	○	○	○	○	○
Software maintenance	○	○	○	○	○	○

C. **How many projects (per application domain) have you been involved in the development team? Check the bottom-most item that applies.**

	No exp.	1 project	2-3 projects	4-5 projects	6-9 projects	10+ projects
Information systems	○	○	○	○	○	○
Embedded systems	○	○	○	○	○	○
Distributed systems	○	○	○	○	○	○
Scientific systems	○	○	○	○	○	○
Expert systems	○	○	○	○	○	○
Components (COTS)	○	○	○	○	○	○

D. **What is your overall experience with software development in Java? Please rate your general skills according to the *novice-expert* scale below (Dreyfus and Dreyfus, 1980).**

Novice are still at a learning stage.
Advanced beginner uses learned procedures and rules to determine what actions are required for the immediate situation.
Competent are task-oriented and deliberately structure their work in terms of plans for goal achievement. Competent can respond to many situations but lack the ability to recognize situations in terms of an overall picture.
Proficient perceive situations as a whole and have more ability to recognize and respond to changing circumstances.
Expert recognize unexpected project responses and can alert others to potential problems before they occur. Experts have an intuitive grasp of whole situations and are able to accurately diagnose and respond without wasteful consideration of ineffective possibilities. Because of their superior performance, expert developers are often consulted by other developers and relied upon to be technical leaders.

Novice	Advanced beginner	Competent	Proficient	Expert
○	○	○	○	○

E. **How many years of experience with Java development?** ______

F. **What is your overall experience with software testing? Check the bottom-most item that applies.**

	No exp.	Novice	Advanced beg.	Competent	Proficient	Expert
Test management	○	○	○	○	○	○
Test specification gathering	○	○	○	○	○	○
Manual test design/execution	○	○	○	○	○	○
Automated test design/execution	○	○	○	○	○	○
Automated testing with JUnit	○	○	○	○	○	○

G. **How many years of experience with software testing?** ______

H. **Please rate your skills with the development of Software Product Lines.**

	No exp.	Novice	Advanced beginner	Competent	Proficient	Expert
Academy	○	○	○	○	○	○
Industry	○	○	○	○	○	○

I. **How many years of experience with software product line engineering (if any)?** ______

J. **Please rate your skills with the development of variable/configurable systems (recall the notions of variability from the training session).**

	No exp.	Novice	Advanced beginner	Competent	Proficient	Expert
Implementation	○	○	○	○	○	○
Testing	○	○	○	○	○	○

K. **How many years of experience with the development of variable/configurable systems (if any)?** ______

L. **How do you consider your skills about identifying feature information from the source code, i.e., given any project source code, are you able to identify its features?**

No exp.	Novice	Advanced beginner	Competent	Proficient	Expert
○	○	○	○	○	○

B.2 Background questionnaire - raw data

See Table B.1. The column item represents the items from the Background Questionnaire. Moreover, each item $B-n$ matches a subitem in the multiple-option grids, e.g., the item *B-1* refers to the *Project Management*. Besides, every option was attributed a value (from 1 to 5, to a 5-item set of options, or from 0 to 5, to a 6-item set of options, especially when there is the option *None/No experience*). For instance, let a subject α has a 2-year experience in Project Management, hence the item *Project Management* in the Table would be valued 2, as it fits into the third column (1-3 years). See the example below.

	No exp.	< 1 year	1-3 years	3-6 years	6-9 years	> 9 years
Project management	○	○	◉	○	○	○
item value	0	1	**2**	3	4	5

Table B.1: Raw data of the background questionnaire applied in the case study.

Item	Subject ID						
	1	2	3	4	5	6	7
A	4	3	2	3	2	3	3
B-1	2	1	0	0	0	1	0
B-2	2	2	1	2	1	1	2
B-3	3	2	1	2	1	3	1
B-4	4	3	2	3	2	2	2
B-5	0	0	1	0	1	0	2
B-6	0	0	0	0	0	1	0
B-7	0	0	0	0	0	0	0
C-1	2	5	2	0	0	1	0
C-2	1	0	0	1	2	1	1
C-6	3	0	0	0	0	1	2
C-3	3	0	2	2	1	0	0
C-4	3	3	0	0	0	2	0
C-5	0	0	0	0	0	0	0
C-6	0	3	0	0	0	0	0
D	3	4	4	3	1	3	2
E	7	6	2	4	1	3	4
F-1	0	0	0	1	0	0	0
F-2	0	0	0	2	0	1	2
F-3	1	2	2	2	1	0	2
F-4	2	2	2	2	0	0	1
F-5	2	2	2	2	0	1	1
G	1	1	1	1	1	0	1
H-1	0	1	0	0	0	0	0
H-2	0	4	2	0	0	0	0
I	0	6	2	0	0	0	0
J-1	0	0	2	0	0	0	0
J-2	0	0	2	0	0	0	0
K	2	2	1	3	2	3	1
L	2	4	0	2	3	3	1

C

Experimental Study - SPL Architecture

This appendix shows the class diagrams and the relationship between classes of the SPL project used to evaluate the proposed fault modeling approach, earlier addressed in Chapter 7.

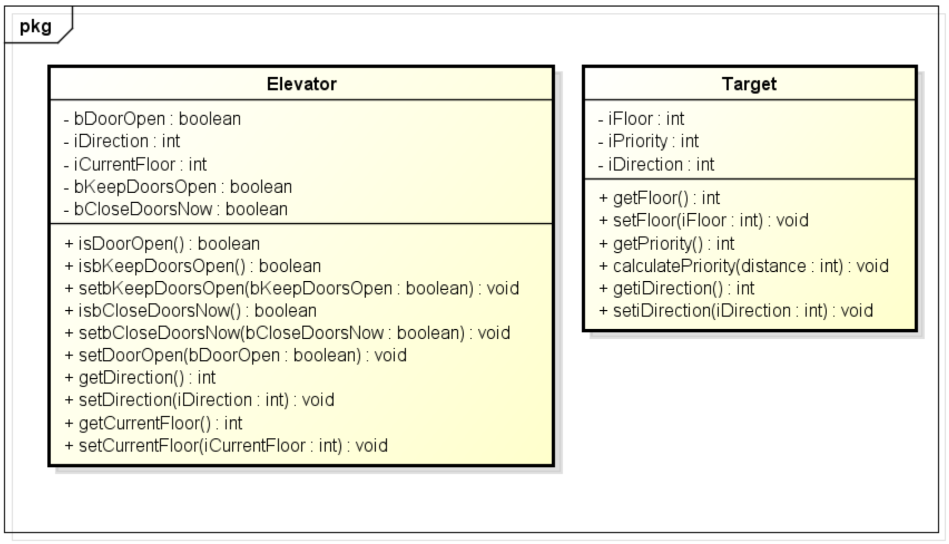

pkg
Elevator
- bDoorOpen : boolean
- iDirection : int
- iCurrentFloor : int
- bKeepDoorsOpen : boolean
- bCloseDoorsNow : boolean
+ isDoorOpen() : boolean
+ isbKeepDoorsOpen() : boolean
+ setbKeepDoorsOpen(bKeepDoorsOpen : boolean) : void
+ isbCloseDoorsNow() : boolean
+ setbCloseDoorsNow(bCloseDoorsNow : boolean) : void
+ setDoorOpen(bDoorOpen : boolean) : void
+ getDirection() : int
+ setDirection(iDirection : int) : void
+ getCurrentFloor() : int
+ setCurrentFloor(iCurrentFloor : int) : void
Target
- iFloor : int
- iPriority : int
- iDirection : int
+ getFloor() : int
+ setFloor(iFloor : int) : void
+ getPriority() : int
+ calculatePriority(distance : int) : void
+ getiDirection() : int
+ setiDirection(iDirection : int) : void

Figure C.1: Classes of the package *simulator model*.

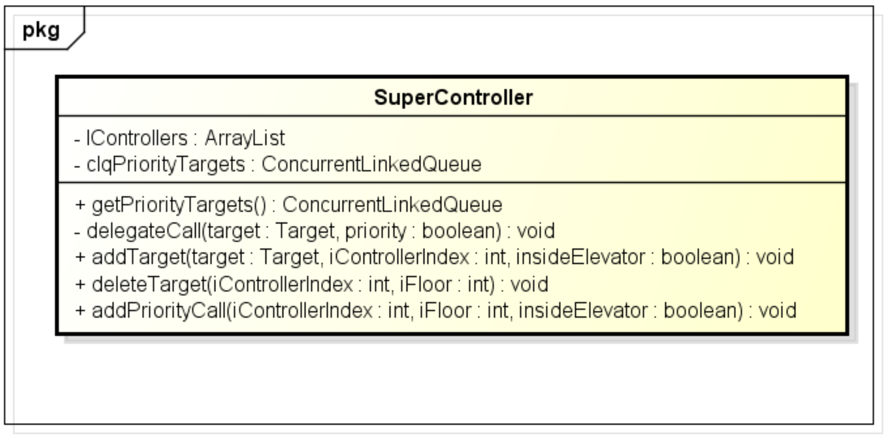

pkg
SuperController
- lControllers : ArrayList
- clqPriorityTargets : ConcurrentLinkedQueue
+ getPriorityTargets() : ConcurrentLinkedQueue
- delegateCall(target : Target, priority : boolean) : void
+ addTarget(target : Target, iControllerIndex : int, insideElevator : boolean) : void
+ deleteTarget(iControllerIndex : int, iFloor : int) : void
+ addPriorityCall(iControllerIndex : int, iFloor : int, insideElevator : boolean) : void

Figure C.2: Classes of the package *simulator*.

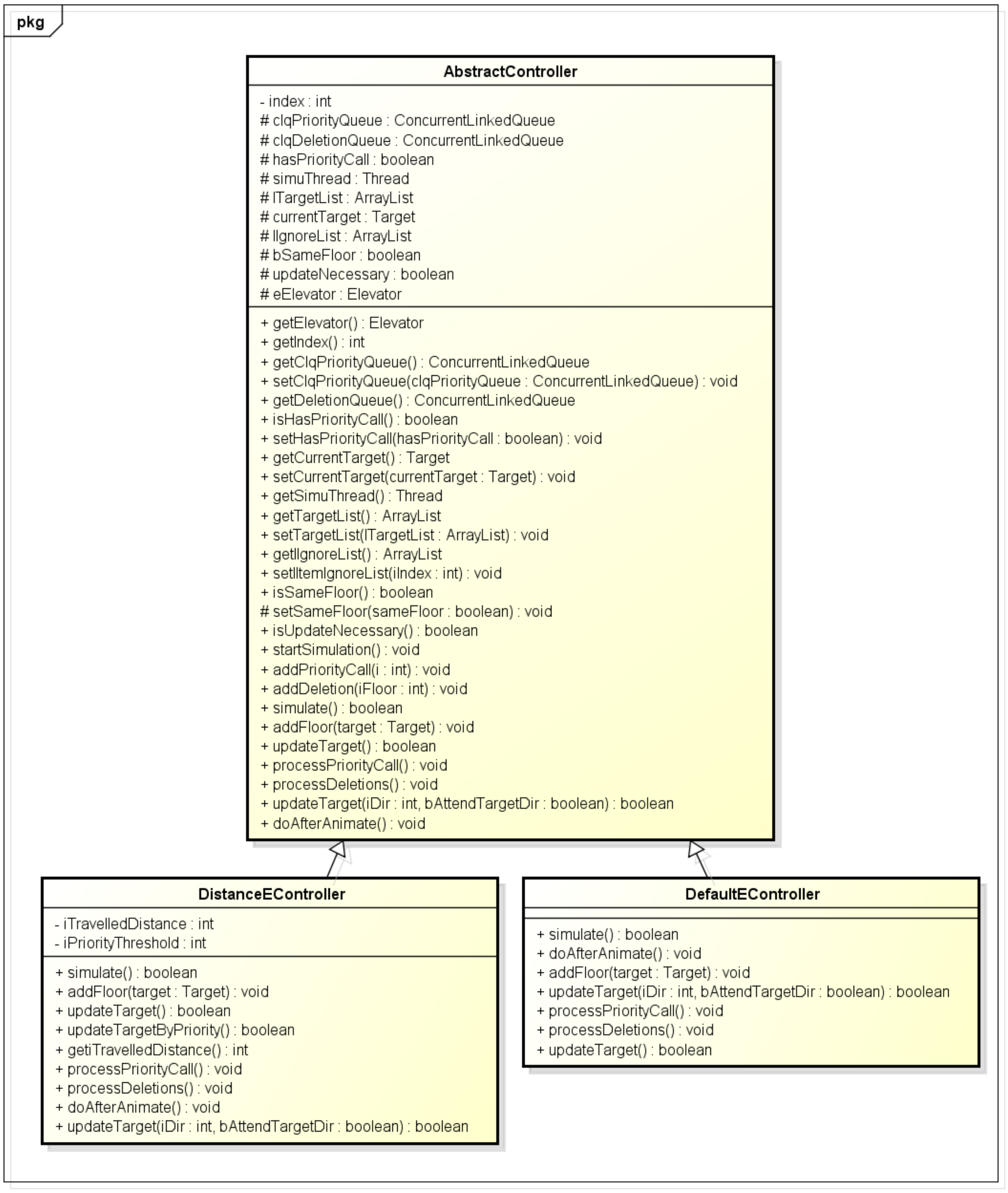

pkg
AbstractController
- index : int
# clqPriorityQueue : ConcurrentLinkedQueue
# clqDeletionQueue : ConcurrentLinkedQueue
# hasPriorityCall : boolean
# simuThread : Thread
# lTargetList : ArrayList
# currentTarget : Target
# lIgnoreList : ArrayList
# bSameFloor : boolean
# updateNecessary : boolean
# eElevator : Elevator
+ getElevator() : Elevator
+ getIndex() : int
+ getClqPriorityQueue() : ConcurrentLinkedQueue
+ setClqPriorityQueue(clqPriorityQueue : ConcurrentLinkedQueue) : void
+ getDeletionQueue() : ConcurrentLinkedQueue
+ isHasPriorityCall() : boolean
+ setHasPriorityCall(hasPriorityCall : boolean) : void
+ getCurrentTarget() : Target
+ setCurrentTarget(currentTarget : Target) : void
+ getSimuThread() : Thread
+ getTargetList() : ArrayList
+ setTargetList(lTargetList : ArrayList) : void
+ getlIgnoreList() : ArrayList
+ setlItemIgnoreList(iIndex : int) : void
+ isSameFloor() : boolean
# setSameFloor(sameFloor : boolean) : void
+ isUpdateNecessary() : boolean
+ startSimulation() : void
+ addPriorityCall(i : int) : void
+ addDeletion(iFloor : int) : void
+ simulate() : boolean
+ addFloor(target : Target) : void
+ updateTarget() : boolean
+ processPriorityCall() : void
+ processDeletions() : void
+ updateTarget(iDir : int, bAttendTargetDir : boolean) : boolean
+ doAfterAnimate() : void
DistanceEController
- iTravelledDistance : int
- iPriorityThreshold : int
+ simulate() : boolean
+ addFloor(target : Target) : void
+ updateTarget() : boolean
+ updateTargetByPriority() : boolean
+ getiTravelledDistance() : int
+ processPriorityCall() : void
+ processDeletions() : void
+ doAfterAnimate() : void
+ updateTarget(iDir : int, bAttendTargetDir : boolean) : boolean
DefaultEController
+ simulate() : boolean
+ doAfterAnimate() : void
+ addFloor(target : Target) : void
+ updateTarget(iDir : int, bAttendTargetDir : boolean) : boolean
+ processPriorityCall() : void
+ processDeletions() : void
+ updateTarget() : boolean

Figure C.3: Classes of the package *simulator controllers*.

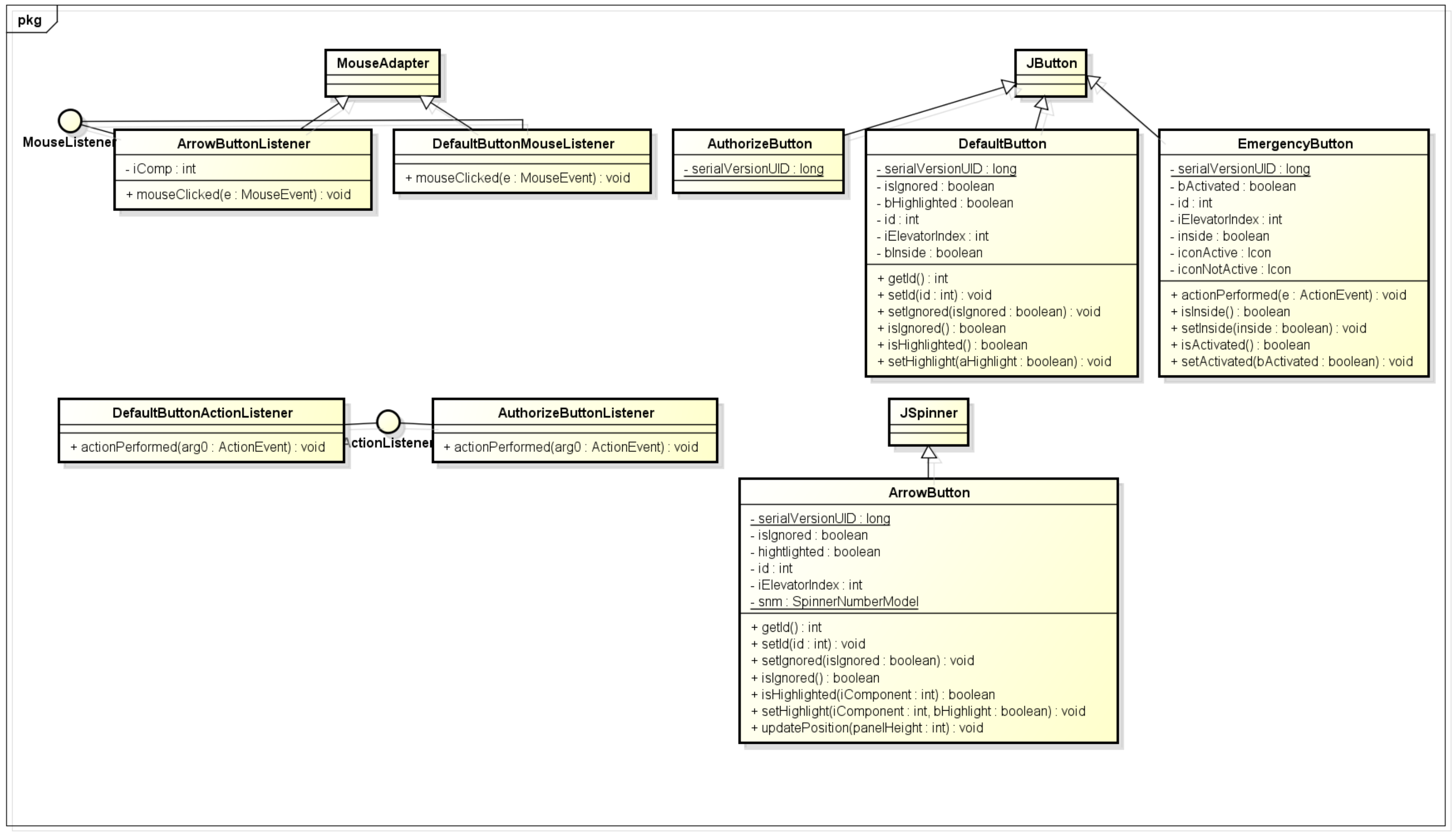

pkg
MouseAdapter
JButton
MouseListener
ArrowButtonListener
- iComp : int
+ mouseClicked(e : MouseEvent) : void
DefaultButtonMouseListener
+ mouseClicked(e : MouseEvent) : void
AuthorizeButton
- serialVersionUID : long
DefaultButton
- serialVersionUID : long
- isIgnored : boolean
- bHighlighted : boolean
- id : int
- iElevatorIndex : int
- bInside : boolean
+ getId() : int
+ setId(id : int) : void
+ setIgnored(isIgnored : boolean) : void
+ isIgnored() : boolean
+ isHighlighted() : boolean
+ setHighlight(aHighlight : boolean) : void
EmergencyButton
- serialVersionUID : long
- bActivated : boolean
- id : int
- iElevatorIndex : int
- inside : boolean
- iconActive : Icon
- iconNotActive : Icon
+ actionPerformed(e : ActionEvent) : void
+ isInside() : boolean
+ setInside(inside : boolean) : void
+ isActivated() : boolean
+ setActivated(bActivated : boolean) : void
DefaultButtonActionListener
+ actionPerformed(arg0 : ActionEvent) : void
ActionListener
AuthorizeButtonListener
+ actionPerformed(arg0 : ActionEvent) : void
JSpinner
ArrowButton
- serialVersionUID : long
- isIgnored : boolean
- hightlighted : boolean
- id : int
- iElevatorIndex : int
- snm : SpinnerNumberModel
+ getId() : int
+ setId(id : int) : void
+ setIgnored(isIgnored : boolean) : void
+ isIgnored() : boolean
+ isHighlighted(iComponent : int) : boolean
+ setHighlight(iComponent : int, bHighlight : boolean) : void
+ updatePosition(panelHeight : int) : void

Figure C.4: Classes of the package *gui buttons*.

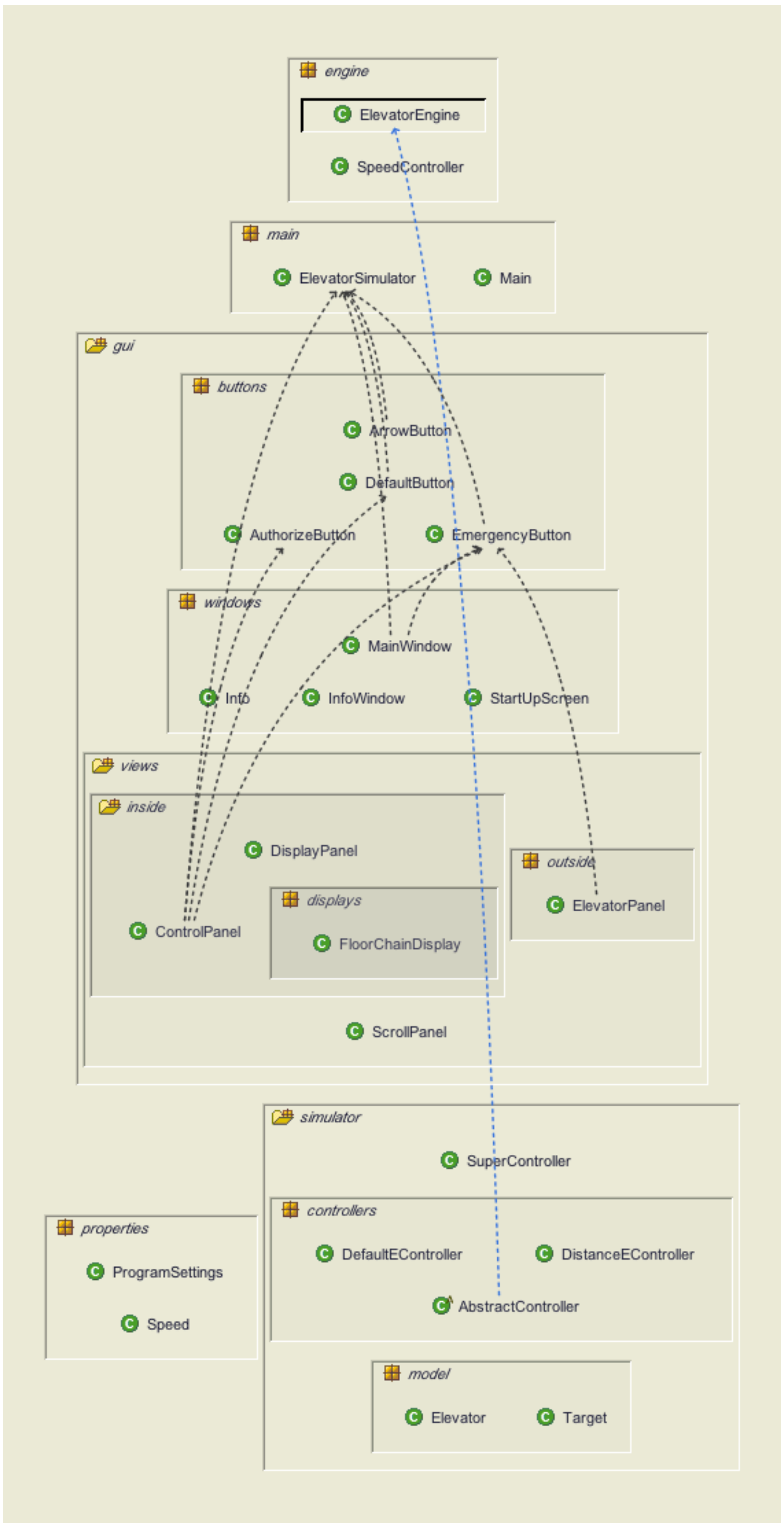


Figure C.5: Relationship between packages and classes.

Programa Multiinstitucional de
Pós-Graduação em Ciência da Computação – PMCC

PMCC-IM-UFBA, Campus de Ondina
Av. Adhemar de Barros S/N, Salvador – Bahia. CEP: 40.170-110
dmcc@ufba.br http://dmcc.dcc.ufba.br/