



“A Regression Testing Approach for Software Product Lines Architectures”

By

Paulo Anselmo da Mota Silveira Neto

M.Sc. Dissertation



Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, June/2010



Universidade Federal de Pernambuco

Centro de Informática

Pós-Graduação em Ciência da Computação

Paulo Anselmo da Mota Silveira Neto

“A Regression Testing Approach for Software Product Lines Architectures”

Trabalho apresentado ao Programa de Pós-Graduação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

A M.Sc. Dissertation presented to the Federal University of Pernambuco in partial fulfillment of the requirements for the degree of M.Sc. in Computer Science.

Advisor: *Silvio Romero de Lemos Meira*
Co-Advisor: *Eduardo Santana de Almeida*

RECIFE, June/2010

Silveira Neto, Paulo Anselmo da Mota

A regression testing approach for software product lines architectures / Paulo Anselmo da Mota Silveira Neto. - Recife: O Autor, 2010.

xvi, 164 folhas : il., fig., tab.,

Dissertação (mestrado) – Universidade Federal de Pernambuco. Cln. Ciência da computação, 2010.

Inclui bibliografia e apêndices.


1. Engenharia de software. 2. Teste de software. 3. Teste de linha de produtos de software. I. Título.

005.1


CDD (22. ed.)

MEI2010 – 095

Dissertação de Mestrado apresentada por **Paulo Anselmo da Mota Silveira Neto** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “**A Regression Testing Approach for Software Product Lines Architectures**”, orientada pelo **Prof. Silvio Romero de Lemos Meira** e aprovada pela Banca Examinadora formada pelos professores:



Prof. Alexandre Cabral Mota
Centro de Informática / UFPE




Profa. Roberta de Souza Coelho
Departamento de Informática e Matemática Aplicada/UFRN



Prof. Silvio Romero de Lemos Meira
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 28 de junho de 2010.



Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

*I dedicate this dissertation to myself and all my family,
friends and professors who gave me all necessary support to
get here.*

Acknowledgements

Initially, I would like to thank my great family and friends. Especially, my parents that always stood by me with everything I needed during my life and my sister for hear me in difficult moments. My uncles, in particular Angelo Silveira which gave me the opportunity to create a horse (the unforgettable Xogum), which provided me a lot of magic moments. A special thank you for my cousins Arthur Silveira, Andrezinho, Leonardo Miranda, João Ricardo, Rodrigo Cavalcanti, Marcela Cavalcanti and João Neto. I would like to thank my grandfather to serve as an example of determination and struggle.

The results of this dissertation could not be achieved without the support of the Reuse in Software Engineering (RiSE) Labs. My gratitude in special to Eduardo Almeida and Vinicius Cardoso for their advises and support in this long journey, and my advisor, Silvio Meira, for accepting me as his student. I would like to thank all my friends from RiSE (Ivan Machado, Thiago Burgos, Leandro Marques, Vanilson Buregio, Yguarata Cavalcanti, Liana Barachisio, Flavio Medeiros, Heberth Braga, Ivonei Freitas, Hernan Munoz, Marcela Balbino, Danuza Neiva, Iuri Santos, Jonatas Bastos), friends from C.E.S.A.R. (Andre Muniz, Mitsuo Takaki, Diego Delgado, Pedro Cunha, Eudes Costa, Ricardo Cheng, Rafael Lima, Rafael Villar, Tereza Novais).

Next, I would like to thank FACEPE for the financial support, which helped me during my master degree. Without this support, I could not spend my time researching and trying to do my best to complete this dissertation on time.

My gratitude to Dr. John D. McGregor his suggestions during our discussions improved the quality of my work.

Finally, I would like to thank God for giving me the wisdom and force to perform this work. In all the moments, You never abandoned me!

*Vou contar a minha história
Do meu cavalo alazão
Era meu melhor amigo
Eu dei-lhe o nome lampião
Por ser um destemido
Cavalo ligeiro, corajoso
Onde ele ia comigo
Gado valente era medroso
Era o cavalo mais cotado
De toda região
Pois em toda a vaquejada
Todo boi ia pro chão
Um vaqueiro respeitado
Era sempre campeão
Tudo isso só por causa
Do meu cavalo lampião*

—RITA DE CASSIA (Meu Cavalo Lampião)

Resumo

Com o objetivo de produzir produtos individualizados, muitas vezes, as empresas se deparam com a necessidade de altos investimentos, elevando assim os preços de produtos individualizados. A partir dessa necessidade, muitas empresas, começaram a introduzir o conceito de plataforma comum, com o objetivo de desenvolver uma grande variedade de produtos, reusando suas partes comuns. No contexto de linha de produto de software, essa plataforma em comum é chamada de arquitetura de referência, que prove uma estrutura comum de alto nível onde os produtos são construídos.

A arquitetura de software, de acordo com alguns pesquisadores, está se tornando o ponto central no desenvolvimento de linha de produtos, sendo o primeiro modelo e base para guiar a implementação dos produtos. No entanto, essa arquitetura sofre modificações com o passar do tempo, com o objetivo de satisfazer as necessidades dos clientes, a mudanças no ambiente, além de melhorias e mudanças corretivas. Desta forma, visando assegurar que essas modificações estão em conformidade com as especificações da arquitetura, não introduziram novos erros e que as novas funcionalidades continuam funcionando como esperado, a realização de testes de regressão é importante.

Neste contexto, este trabalho apresenta uma abordagem de regressão utilizada tanto para reduzir o número de testes que precisam ser reexecutados, da arquitetura de referência e da arquitetura dos produtos, quanto para tentar assegurar que novos erros não foram inseridos, depois que essas arquiteturas passaram por uma evolução ou mudança corretiva. Como regressão é vista como uma técnica que pode ser aplicada em mais de uma fase de teste, neste trabalho regressão é aplicado durante a fase de integração, uma vez que, ao final desta fase teremos as arquiteturas da linha de produto testadas. Desta forma, uma abordagem de integração também foi proposta.

Esta dissertação também apresenta uma validação inicial da abordagem, através de um estudo experimental, mostrando indícios de que será viável a aplicação de testes de regressão nas arquiteturas de uma linha de produto de software.

Palavras-chave: Engenharia de software; Teste de software; Teste de linha de produtos de software.

Abstract

To achieve the ability to produce individualized products, often, companies need high investments which lead sometimes high prices for a individualized product. Thus, many companies, started to introduce the common platform in order to assemble a greater variety of products, by reusing the common parts. In the Software Product Lines (SPL) context, this common platform is called the reference architecture, which provides a common, high-level structure for all product line applications.

Software architectures are becoming the central part during the development of quality systems, being the first model and base to guide the implementation and provide a promising way to deal with large systems. At times, it evolves over time in order to meet customer needs, environment changes, improvements or corrective modifications. Thus, in order to be confident that these modifications are conform with the architecture specification, did not introduce unexpected errors and that the new features work as expected, regression test is performed.

In this context, this work describes a regression testing approach used to reduce the number of tests to be rerun, for both reference architecture and product specific architecture, and to be confident that no new errors were inserted after these architectures suffer a evolution or a corrective modification. Regression is a technique applied in testing different test levels, in this work we are interested in apply it during integration testing level, since the main objective of this level is verify the SPL architectures conformance. Thus, an integration testing approach was also proposed.

This dissertation also presents a validation of the initial approach, through an experimental study, presenting indicators of its use viability in software product line architectures.

Keywords: Software engineering; Software testing; Software product lines testing.

Table of Contents

List of Figures	xiii
List of Tables	xv
List of Acronyms	xvi
1 Introduction	1
1.1 Motivation	2
1.2 Problem Statement	3
1.3 Overview of the Proposed Solution	3
1.3.1 Context	3
1.3.2 Proposal Outline	5
1.4 Out of Scope	5
1.5 Statements of the Contribution	6
1.6 Dissertation Structure	6
2 Software Product Lines: An Overview	8
2.1 Introduction	8
2.2 SPL Essential Activities	10
2.2.1 Core Asset Development	10
2.2.2 Product Development	12
2.2.3 Management	13
2.3 SPL Variability Management	13
2.4 SPL Adoption Strategies	14
2.5 Industrial Experiences with SPL	14
2.6 Chapter Summary	17
3 Overview on Software Testing	18
3.1 Introduction	18
3.2 Fundamental Concepts	19
3.3 The Testing Process	20
3.4 Testing Levels	21
3.4.1 Unit Testing	22
3.4.2 Integration Testing	23
3.4.3 System Testing	23

3.4.4	Acceptance Testing	24
3.5	Regression Testing	25
3.6	Testing Strategies	25
3.6.1	Black-Box Testing Methods	26
3.6.2	White-Box Testing Methods	27
3.7	SPL and Software Testing	27
3.8	Chapter Summary	28
4	A Mapping Study on Software Product Line Testing	29
4.1	Introduction	30
4.2	Literature Review Method	31
4.3	Research Directives	34
4.3.1	Protocol Definition	34
4.3.2	Question Structure	34
4.3.3	Research Questions	35
4.4	Data Collection	36
4.4.1	Search Strategy	36
4.4.2	Data Sources	37
4.4.3	Studies Selection	38
4.4.3.1	Reliability of Inclusion Decisions	41
4.4.4	Quality Evaluation	41
4.4.5	Data Extraction	43
4.5	Outcomes	44
4.5.1	Classification Scheme	44
4.5.2	Results	45
4.5.2.1	Testing Strategy	45
4.5.2.2	Static and Dynamic Analysis	46
4.5.2.3	Testing Levels	47
4.5.2.4	Regression Testing	48
4.5.2.5	Non-functional Testing	49
4.5.2.6	Commonality and Variability Testing	50
4.5.2.7	Variant Binding Time	51
4.5.2.8	Effort Reduction	51
4.5.2.9	Test Measurement	53
4.5.3	Analysis of the Results and Mapping of Studies	53
4.5.3.1	Main findings of the study	57

4.6	Threats to Validity	59
4.7	Related Work	60
4.8	Concluding Remarks	61
4.9	Chapter Summary	61
5	A SPL Integration Testing Approach	63
5.1	Introduction	64
5.2	Integration Testing in SPL	65
5.3	Unit and Integration Testing	66
5.4	Roles and Attributions	68
5.4.1	Method Content and Processes	69
5.5	Integration Testing Strategies	70
5.6	Integration Testing approach for SPL	72
5.6.1	Integration Testing in Core Asset Development (CAD)	72
5.6.2	Integration Testing in Product Development (PD)	78
5.7	Example using the Approach	79
5.8	Chapter Summary	82
6	A Regression Testing Approach for Software Product Lines Architectures	83
6.1	Introduction	83
6.2	Other Directions in SPL Regression Testing	85
6.3	A Regression Testing Overview	86
6.3.1	Maintenance Categories	86
6.3.2	Corrective vs Progressive Regression Testing	87
6.3.3	Test Case Classes	88
6.3.4	Typical Selective Retest Technique	89
6.4	Regression at Integration Level	90
6.5	Regression Testing in SPL Architectures	92
6.6	A Regression Testing Approach for SPL Architectures	94
6.6.1	Approach Steps	95
6.6.1.1	Planning	95
6.6.1.2	Analyzes	96
6.6.1.3	Test Design and Selection	98
6.6.1.4	Execution	101
6.6.1.5	Reporting	101
6.7	Chapter Summary	102

7	The Experimental Study	104
7.1	Introduction	104
7.2	Definition	104
7.2.1	Goal	105
7.2.2	Questions	106
7.2.3	Metrics	106
7.2.4	Definition Summary	107
7.3	Planning	107
7.3.1	Context Selection	108
7.3.2	Hypothesis Formulation	109
7.3.3	Variables Selection	110
7.3.4	Selection of Subjects	110
7.3.5	Experiment Design	111
7.3.6	Instrumentation	111
7.3.7	Validity Evaluation	113
7.4	Operation	115
7.4.1	Preparation	115
7.4.2	Execution	116
7.4.3	Data Validation	118
7.5	Analysis and Interpretation	118
7.5.1	Effort to Apply the Approach	118
7.5.1.1	Corrective Scenario	119
7.5.1.2	Progressive Scenario	123
7.5.2	Approach Understanding and Application Difficulties	125
7.5.2.1	Correlation Analysis	126
7.5.3	Activities, Roles and Artifacts Missing	127
7.5.3.1	Correlation Analysis	127
7.5.4	Number of Defects	127
7.5.4.1	Correlation Analysis	128
7.5.5	Number of Tests Correctly Classified	129
7.5.5.1	Correlation Analysis	130
7.6	Lessons Learned	130
7.7	Chapter Summary	131

8 Conclusion	132
8.1 Research Contributions	133
8.2 Related Work	134
8.3 Future Work	135
8.4 Academic Contributions	136
8.5 Concluding Remarks	136
References	138
Appendices	155
A Experimental Study Questionnaires	156
A.1 Background Questionnaire	156
A.2 Regression Testing Approach Analysis Questionnaire	159
B Mapping Study Sources	161
B.1 List of Conferences	161
B.2 List of Journals	162
C Quality Studies Scores	163

List of Figures

1.1	RiSE Labs Influences.	4
1.2	RiSE Labs Projects.	4
2.1	Essential product line activities (Northrop, 2002).	10
2.2	Core Asset Development (Northrop, 2002).	11
2.3	Product Development (Northrop, 2002).	12
3.1	Difference among Error, Fault and Failure	20
3.2	Testing level activities.	21
3.3	The V-Model.	22
3.4	Types of System Tests (Burnstein, 2003).	24
3.5	Testing Strategies (Burnstein, 2003).	25
4.1	The Systematic Mapping Process (adapted from Petersen <i>et al.</i> (2008)).	32
4.2	Stages of the selection process.	39
4.3	Primary studies filtering categorized by source.	40
4.4	Distribution of primary studies by their publication years.	41
4.5	Amount of Studies vs. sources.	42
4.6	Distribution of papers according to classification scheme.	54
4.7	Distribution of papers according to intervention.	54
4.8	Visualization of a Systematic Map in the Form of a Bubble Plot.	55
5.1	RiPLE Unit Testing level main flow.	67
5.2	Two top-down manners to integrate components and modules.	71
5.3	An overview on the RiPLE-TE Integration approach work flow.	73
5.4	RiPLE Integration Testing level (CAD) main flow.	74
5.5	Variability influence in components interactions.	76
5.6	RiPLE Integration Testing level (PD) main flow.	78
5.7	Feature Dependency Diagram.	80
5.8	ProductMap.	80
5.9	Sequence Diagram.	81
5.10	Architecture modules.	82
6.1	Corrective and Progressive Regression Testing Leung and White (1989).	89
6.2	A Sequence Diagram with two variation points.	91

6.3	Similarities among product architectures.	92
6.4	Two Reference Architecture Versions.	92
6.5	Reference Architecture and Product Specific Architecture.	93
6.6	Similar Product Architectures.	94
6.7	The Regression Testing Approach.	95
6.8	An illustrative example of a bank system class diagram.	97
6.9	Credit method from Special Account class.	97
6.10	Two different versions of a program (Apiwattanapong <i>et al.</i> , 2007) . . .	99
6.11	Two different versions of a method (Apiwattanapong <i>et al.</i> , 2007) . . .	99
6.12	The Overall Regression Testing Approach.	103
7.1	Planning phase overview (Wohlin <i>et al.</i> , 2000).	108
7.2	Experiment Scenarios.	112
7.3	Planning step distribution.	120
7.4	Box plot analysis.	120
7.5	Outliers Analysis.	122
7.6	Outliers Analysis.	124
7.7	Outlier (ID 3) from Reporting step.	125
7.8	Difficulties during approach execution.	126
7.9	Boxplot Analysis.	128
7.10	Number of Subjects vs Faults.	129

List of Tables

2.1	Software Product Line Industrial Cases (Pohl <i>et al.</i> , 2005a; Linden <i>et al.</i> , 2007).	16
4.1	List of Search Strings	37
4.2	Quality Criteria	42
4.3	Research Type Facet	44
4.4	Research Questions (RQ) and primary studies.	56
6.1	Software Maintenance Categories Distribution Hatton (2007).	87
7.1	Subject's Profile.	117
7.2	Approach execution effort (minutes) considering corrective scenario. . .	119
7.3	Effort to apply the approach.	122
7.4	Approach execution effort considering progressive scenario.	123
7.5	Effort to apply the approach.	125
7.6	Difficulties to use the approach.	126
7.7	Defects per subjects.	127
7.8	Number of tests correctly classified.	130

List of Acronyms

C.E.S.A.R. Recife Center for Advanced Studies and Systems (C.E.S.A.R)

GQM Goal-Question Metric

PLA Product Line Architecture

SPL Software Product Lines

PD Product Development

CAD Core Asset Development

RiSE Reuse in Software Engineering Labs

PA Product Architecture

RA Reference Architecture

RiPLE RiSE process for Product Line Engineering

SR Systematic Review

MS Mapping Study

SLR Software Literature Review

CR Change Request

EPF Eclipse Process Framework

'If you think education is expensive, try ignorance'

Derek Bo



Introduction

It has been a challenge for software developers and testers to develop and maintain software system for industry as a result of changes in market and customers requirements (Edwin, 2007). Based on the systematic and planned reuse of previous development efforts among a set of similar products, the Software Product Lines (SPL) approach enables organizations not only to reduce development and maintenance costs, as well as achieving impressive productivity, time-to-market gains and quality improvements.

Testing, still the most effective way for quality assurance, is more critical and complex for product lines than for traditional single software systems (Kolb and Muthig, 2003). According to McGregor *et al.* (2004a), in the context of SPL, software testing for a product can cost from 50% to 200% more than the software development itself. Reducing costs and increasing productivity in the test process is just as important as it is for product creation. Thus, it is important to start testing activities as soon as possible even with static analysis (with no executable code), since a simple fault discovered in a core asset or common platform, may affect the software product line as a whole increasing the cost to correct that problem and impacting the customer satisfaction.

Thus, this dissertation explores the combination of the concepts and characteristics of SPL and testing in a single software engineering approach. In particular, an approach for regression testing product line architectures is defined. In this proposed solution, SPL concepts, such as reference architecture, product specific architecture, commonality and variabilities are considered to support reuse in the SPL testing phase. In addition, regression testing concepts are used to test product line architectures taking advantage of their similarities.

In this dissertation, the focus is on studying the state-of-the-art in software testing for software product lines and providing a systematic approach for regression testing SPL architectures, always searching for maximize the benefits of systematic reuse. In this

way, product line architectures are modified and evolved and can be regression tested considering their commonalities. As the main focus is on SPL architecture testing, an integration testing approach was also defined in order to check if the implementation fulfills (conforms to) its specification. Therefore, the regression testing approach is applied as a technique during integration testing level.

The remainder of this chapter describes the focus and structure of this dissertation. Section 1.1 starts presenting its motivations, and a clear definition of the problem scope is depicted in Section 1.2. An overview of the proposed solution is presented in Section 1.3. Some related aspects that are not directly addressed by this work are shown in Section 1.4. In the Section 1.5, the main contributions of this work are discussed, and finally, Section 1.6 describes how this dissertation is organized.

1.1 Motivation

In the SPL context, a common platform called the reference architecture, provides a common, high-level structure for all product line applications (Pohl *et al.*, 2005a). The architecture is one of the most important assets of a SPL, since all products are derived from it. Considering its importance, this dissertation defined two approaches in order to verify its quality. Firstly, the integration testing approach which aims to verify if the architecture implementation fulfills with its respective specifications, through conformance testing. At last, the regression testing approach, applied after the architecture evolution or modification, in order to be confident that the new version still working properly and did not introduced new faults.

In an important survey in the testing area, (Bertolino, 2007) proposes a roadmap to address some testing challenges, discussing some achievements and pinpoint some dreams. Regarding to SPL, she describes the challenge “Controlling evolution” as a way to achieve the dream “Efficacy-maximized test engineering” highlighting the importance of effective regression testing techniques to reduce the amount of retesting, to prioritize regression testing test cases and reduce the cost of their execution. Briefly, it is important to scale up regression testing in large composition system, defining an approach to regression testing global system properties when some parts are modified and understand how to test a piece of architecture when it evolves.

The regression testing approach can be used during maintenance and development. During maintenance it is used to be confident that some modifications are conform with the architecture specification, did not introduce unexpected errors and that the new

features work as expected. During product development should be performed in the application architecture or product architecture, in order to ensure that it conforms with its specification (Jin-hua *et al.*, 2008) and maintain the conformance with the reference architecture defined during core asset development phase.

1.2 Problem Statement

Encouraged by the motivations depicted in the previous section, the goal of this dissertation can be stated as follows:

This work defines two approaches for testing software product line architectures defining activities, steps, inputs, outputs and roles in order to be confident that modifications (correction or evolution) are conform with the architecture specification, do not introduce unexpected errors and that the new versions work as expected.

1.3 Overview of the Proposed Solution

In order to test software product line architectures, two testing approaches were developed. The remainder of this section presents the context where it was developed and the outlines the proposed solution.

1.3.1 Context

This dissertation is part of Reuse in Software Engineering Labs (RiSE) ¹ (Almeida *et al.*, 2004), formerly called RiSE Project, whose goal is to develop a robust framework for software reuse in order to enable the adoption of a reuse program for companies. RiSE Labs is influenced by a series of areas, such as software measurement, architecture, quality, environments and tools, and so on, in order to achieve its goal. The influence areas are depicted in Figure 1.1. Based on these areas, the RiSE Labs embraces several different projects related to software reuse, as shown in Figure 1.2. They are following described.

- **RiSE Framework:** It involves reuse process (Almeida *et al.*, 2005; Nascimento, 2008), component certification (Alvaro, 2009) and reuse adoption and adaptation processes (Garcia *et al.*, 2008; Garcia, 2010).

¹labs.rise.com.br

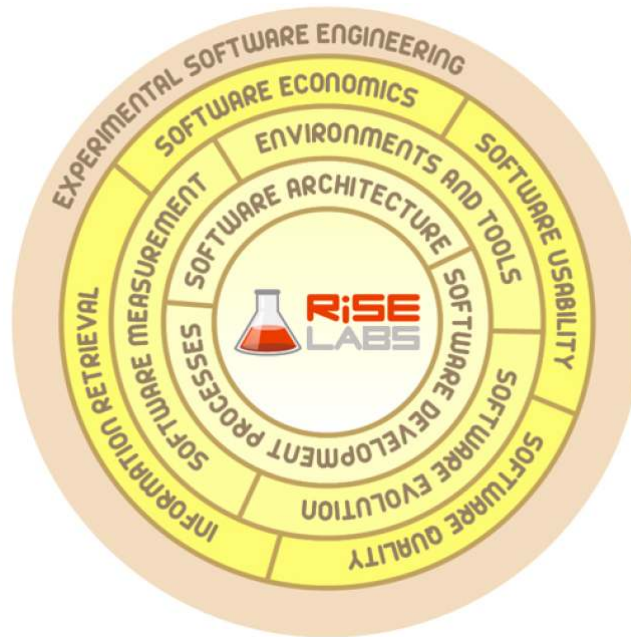


Figure 1.1 RiSE Labs Influences.



Figure 1.2 RiSE Labs Projects.

- **RiSE Tools:** Research focused on software reuse tools, such as the Admire Environment (Mascena *et al.*, 2006), the Basic Asset Retrieval Tool (B.A.R.T) (Eduardo *et al.*, 2006), which was enhanced with folksonomy mechanisms (Vanderlei *et al.*, 2007), semantic layer (Durao, 2008), facets (Mendes, 2008), and data mining (Martins *et al.*, 2008), the Legacy InFormation retrieval Tool (LIFT) (Brito, 2007), the Reuse Repository System (CORE) (Buregio *et al.*, 2007), a tool for Domain Analysis (ToolDay) (Lisboa, 2008) and a Bug Report Analysis and Search Tool (BAST) (Cavalcanti, 2009), (da Cunha, 2009).

- **RiPLE:** Stands for RiSE Product Lines Engineering Process and aims at developing a methodology for Software Product Lines, composed of scoping (Balbino, 2009), requirements engineering (Neiva, 2009), design (Souza Filho *et al.*, 2008), implementation, test, and evolution management.
- **SOPLE:** Development of a methodology for Software Product Lines based on services (Medeiros *et al.*, 2009), following the same structure of RiPLE.
- **MATRIX:** Investigates the area of measurement in reuse and its impact on quality and productivity, based on experimentation.
- **BTT:** Research focused on tools for detection of duplicate bug reports, such as in Cavalcanti *et al.* (2008),(Cunha *et al.*, 2010).
- **Exploratory Research:** Investigates new research directions in software engineering and its impact on reuse.
- **CX-Ray:** Focused on understanding the Recife Center For Advanced Studies and Systems ² (C.E.S.A.R.), and its processes and practices in software development.

This dissertation is part of the RiPLE project and its main goal is to support the architecture regression test in a software product line.

1.3.2 Proposal Outline

The goal of this dissertation is to develop and manage an architecture regression testing approach performed in the integration testing level, by defining a systematic approach composed by four main activities: *(i)* testing planning and analysis, *(ii)* test selection and design, *(iii)* test execution and *(iv)* test reporting, all of them incorporated in an integration approach. These proposed approaches do not exclude existing integration and regression testing techniques, methods and tools, but comes to complement the traditional testing in the software product lines context.

1.4 Out of Scope

As the proposed process is part of a broader context (RiPLE), a set of related aspects will be left out of its scope. Thus, the following issues are not directly addressed by this work:

²www.cesar.org.br

- **Testing Metrics:** Measurement activities are essential in any engineering process. Both measurement activities inside the process and metrics to be used outside the process (to formally evaluate) could be incorporated to the process.
- **Tool Support:** In order to perform some steps in this approach, some tool support may be required. It is out of scope to develop a tool that supports all of the steps. However, other dissertation in our group is investigating this issue.
- **SPL Unit testing:** Considering that unit testing approaches can be perfectly used in the SPL context, we start the architecture testing approach considering only the integration level and regression approach during core asset development and product development. For this reason, unit testing approach was not considered.

1.5 Statements of the Contribution

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

- A mapping study of the state-of-the-art for SPL testing was performed in order to better understand the main trends, gaps and challenges in this area.
- The definition of an integration and regression approach for software product lines.
- The definition, planning, operation and analysis of an experimental study in order to evaluate the proposed approach.

1.6 Dissertation Structure

The remainder of this dissertation is organized as follows:

Chapter 2 discusses the software product lines basic concepts and activities, adoption strategies, as well as successful industry experiences.

Chapter 3 presents software testing fundamental concepts, testing process activities, testing levels, regression testing and testing strategies. The relation between software product lines and software testing are also described.

Chapter 4 presents a mapping study in order to investigate the state-of-the-art testing practices, synthesize available evidence, and identify gaps between needed techniques and existing approaches, available in the literature.

Chapter 5 describes the integration testing approach in the SPL context, presenting the roles associated, activities, inputs and outputs and the key concepts of the approach.

Chapter 6 describes the SPL architecture regression testing approach, activities, steps, inputs and outputs and the main concepts of the approach.

Chapter 7 presents the definition, planning, operation, analysis and interpretation and packaging of the experimental study which evaluates the viability of the proposed approach.

Chapter 8 concludes the dissertation by summarizing the findings and proposing future enhancements to the solution, discussing possible future work and research areas.

“Any customer can have a car painted any color that he wants so long as it is black”

Henry Ford

2

Software Product Lines: An Overview

2.1 Introduction

The concept of software reuse started to be used since 1949, where the first subroutine library was proposed (Tracz, 1988). It gained importance in 1968, during the NATO Software Engineering Conference, considered the birthplace of the field. Its focus was the software crisis - the problem of building large, reliable software systems in a controlled, cost-effective way. Firstly, software reuse was pointed as being the solution of software crisis. McIlroy’s paper entitled “Mass Produced Software Components” (McIlroy, 1968), ended up being the seminal paper in the software reuse area. In his words: “the software industry is weakly founded and one aspect of this weakness is the absence of a software component sub-industry”, it was the basis to consider and investigate mass-customization in software (Almeida, 2007).

On the other hand, the mass-customization idea was born in 1908, in the automobiles domain, when Henry Ford the father of assembly-line automation, built the Model T based on interchangeable parts. It enables the production for mass market more cheaply than individual product creation. However, the production line reduced the products diversification.

Although some customers were satisfied with standardized mass products, not all people want the same kind of car for any purpose. Hence, industry was facing with a growth interest for individualized products. However, mass customization is a “coin” with two distinct faces. In the customer face, mass customization means the ability to have an individualized product, realizing specific needs. For the company, however, it means technological investments, which leads to higher product’s prices and/or lower profit margins for the company (Pohl *et al.*, 2005a).

Considering the software context, two types of software can be observed: (*i*) individual

software products which satisfies specific customers needs and (ii) standard software the mass produced ones. While the first is more expensive to develop, the second suffers a lack of diversification.

In order to avoid higher prices for individualized products and lower profit margins for the companies, some companies introduced the common platform concept for their different types of products, planning beforehand which parts will be further instantiated in different product types. A systematic combination between mass-customization and platform-based development allows us to reuse a common base of technology and, at the same time, to develop products in close accordance with customer needs. Thus, it resulted in “Software Product Line Engineering”, a software development paradigm (Pohl *et al.*, 2005a).

Although Product Lines are no new in manufacturing, Boeing, Ford, Dell and even McDonald’s, the Software Product Lines (SPL) are a relatively new concept, enabling companies exploit their software commonalities to achieve economies of production (Northrop, 2002). It is defined by Clements and Northrop (2001) as being “*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.*” SPL has proven to be the methodology for developing a diversity of software products and software-intensive systems in shorter time, with high quality and at lower costs (Pohl *et al.*, 2005a).

The identification of commonality (common features for the SPL members) and variability (difference among members) is crucial for product diversification. The SPL paradigm was founded in three main activities: (i) Core Asset Development (Domain Engineering), (ii) Product Development (Application Engineering) and (iii) Management. This activities are further exploited in the next section.

Different terms are adopted in the academy and industry to express the same meaning. They might refer to product line as a product family, to core asset set as platform, or to the products of the SPL as customizations or members instead of products. Besides, Core asset development might be referred as domain engineering, and product development as application engineering. In this work the terms adopted are core asset development and product development.

This chapter is organized as follows. Section 2.2 introduces the software product line essential activities. Section 2.3 describes the variability management ideas. Some software product line adoption strategies are described in Section 2.4. Section 2.5 presents software product line successful industrial cases and Section 2.6 summarizes the chapter.

2.2 SPL Essential Activities

Software Product Lines combine three essential and highly iterative activities that blend business practices and technology. Firstly, the *Core Asset Development (CAD)* activity that does not directly aim at developing a product, but rather aims to develop assets to be further reused in other activities. Secondly, *Product Development (PD)* activity which takes advantage of existing, reusable assets. Finally, *Management* activity, which includes technical and organizational management (Linden *et al.*, 2007). Figure 2.1 illustrates this triad of essential activities.

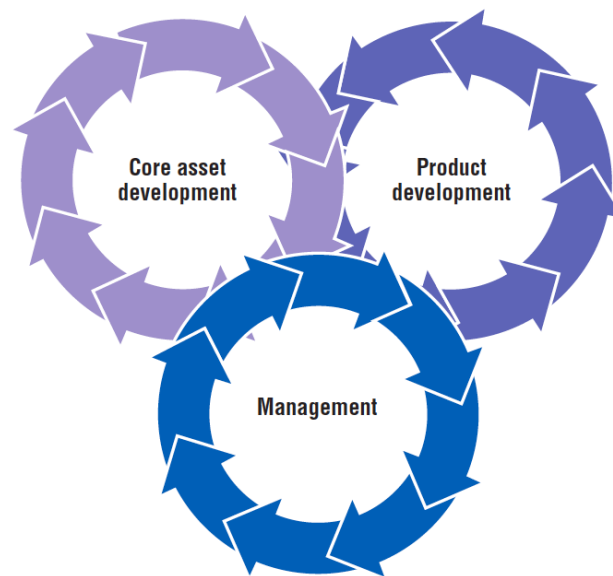


Figure 2.1 Essential product line activities (Northrop, 2002).

2.2.1 Core Asset Development

Core Asset Development is the life-cycle that results in the common assets that in conjunction compose the product line's platform (Linden *et al.*, 2007). The key goals of this activity are (Pohl *et al.*, 2005a):

- Define variability and commonality of the software product line;
- Determine the set of product line planned members (scope); and
- Specify and develop reusable artifacts that accomplish the desired variability and further instantiated to derive product line members.

This activity (Figure 2.2) is iterative, and its inputs and outputs affect each other. This context influences the way in which the core assets are produced. The set of inputs needed to accomplish this activity are following described (Northrop, 2002). *Product constraints* commonalities and variations among the members that will constitute the product line, including their behavioral features; *Production constraints* commercial, military, or company-specific standards and requirements that apply to the products in the product line; *Styles, patterns, and frameworks* relevant architectural building blocks that architects can apply during architecture definition toward meeting the product and production constraints; *Production strategy* the whole approach for realizing the core assets, it can be performed starting with a set of core assets and deriving products (top down), starting from a set of products and generalizing their components in order to produce product line assets (bottom up) or both ways; *Inventory of preexisting assets* software and organizational assets (architecture pieces, components, libraries, frameworks and so on) available at the outset of the product line effort that can be included in the asset base.

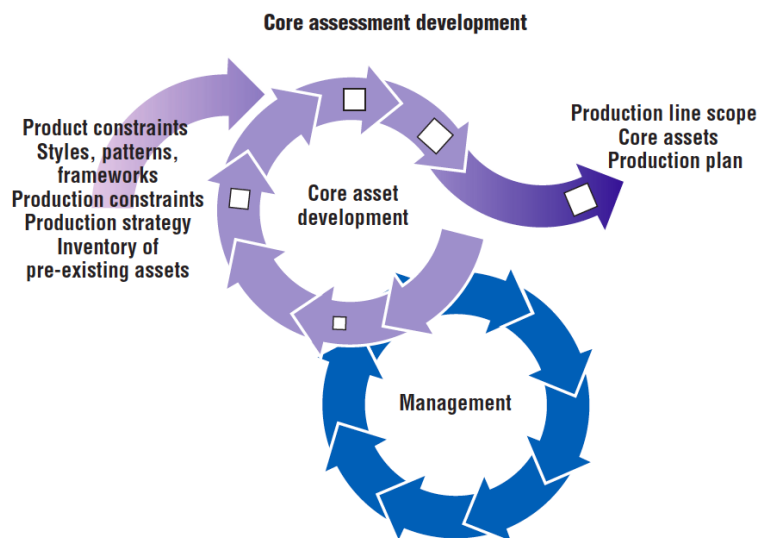


Figure 2.2 Core Asset Development (Northrop, 2002).

Based on previous information (inputs), this activity is subdivided in five disciplines: (i) domain requirements, (ii) domain design, (iii) domain realization (implementation), (iv) domain testing and (v) evolution management, all of them administered by the management activity (Pohl *et al.*, 2005a). These disciplines are responsible for creating the core assets, as well as, the following outputs (Figure 2.2) (Clements and Northrop, 2001): *Product line scope* the description of the products derived from the product line

or that the product line is capable of including. The scope should be small enough to accommodate future growth and big enough to accommodate the variability. *Core assets* comprehend the basis for production of products in the product line, besides the reference architecture, that will satisfy the needs of the product line by admitting a set of variation points required to support the spectrum of products, these assets can also be components and their documentation. The *Production plan* describes how the products are produced from core assets, it also describe how specific tools are to be applied in order to use, tailor and evolve the core assets.

2.2.2 Product Development

The product development main goal is to create individual (customized) products by reusing the core assets previously developed. The CAD outputs (product line scope, core assets and production plan), in conjunction with the requirements for individual products are the main inputs for PD activity (Figure 2.3).

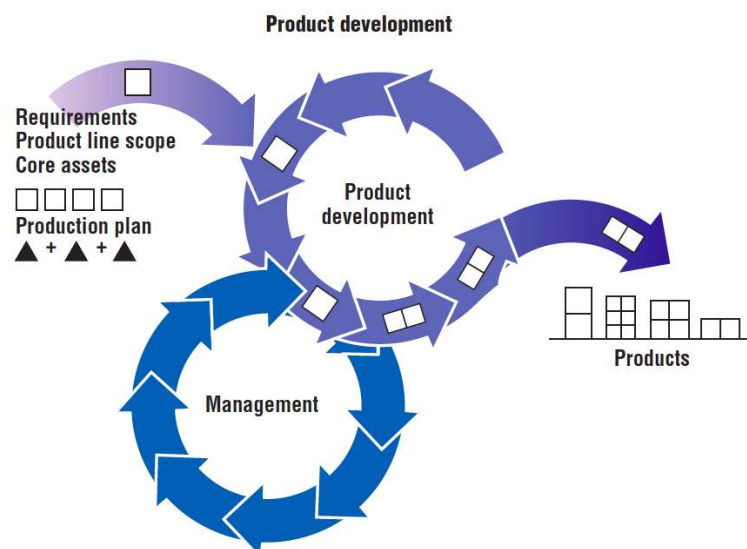


Figure 2.3 Product Development (Northrop, 2002).

In possession of the production plan, which details how the core assets will be used in order to build a product, the software engineer can assemble the product line members. The product requirement is also important to realize a product. Product engineers have also the responsibility to provide feedback on any problem or deficiency encountered in the core assets. It is crucial to avoid the product line decay and keep the core asset base healthy.

2.2.3 Management

The management of both technical and organizational levels are extremely important to the software product line effort. The former supervise the CAD and PD activities by certifying that both groups that build core assets and products are engaged in the activities and to follow the process, the latter must make sure that the organizational units receive the right and enough resources. It is, many times, responsible for the production strategy and the success or failure of the product line.

2.3 SPL Variability Management

During Core Asset Development, variability is introduced in all domain engineering artifacts (requirements, architecture, components, test cases, etc.). It is exploited during Product Development to derive applications tailored to the specific needs of different customers.

According to Svahnberg *et al.* (2005), variability is defined as "*the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context*". It is described through variation points and variants. While, the variation point is the representation of a variability subject (variable item of the real world or a variable property of such an item) within the core assets, enriched by contextual information; the variant is the representation of the variability object (a particular instance of a variability subject) within the core assets (Pohl *et al.*, 2005a).

The variability management involve issues, such as: variability identification and representation, variability binding and control (de Oliveira *et al.*, 2005). Three questions are helpful to variability identification, *what vary* the variability subject, *why does it vary* the drivers of the variability need, such as stakeholder needs, technical reasons, market pressures, etc. The later, *how does it vary* the possibilities of variation, also known as variability objects.

The variability binding indicates the lifecycle milestone that the variants related with a variation point will be realized. The different binding times (e.g.: link, execution, post-execution and compile time) involves different mechanisms (e.g.: inheritance, parameterization, conditional compilation) and are appropriate for different variability implementation schemes. The different mechanisms result in different types of defects, test strategies, and test processes (McGregor *et al.*, 2004a).

Finally, the purpose of variability control is to defining the relationship between artifacts in order to control variabilities.

2.4 SPL Adoption Strategies

With the growth of competitiveness among companies, there is a need for three goals: a faster development, better quality and less time-to-market. It makes SPL paradigm invited for these companies. Based on business goals, adoption strategies and their pros and cons the organization should decide the best way to introduce SPL concepts in its context. In Pohl *et al.* (2005a), four transition strategies are described, as follows:

- **Incremental Introduction** It starts small and expands incrementally, it may occur in two ways, *expanding organizational scope* which starts with a single group doing SPL engineering and other groups are added incrementally after the first group succeed and *expanding investment* which starts with a small investment that is incrementally increased, depending on the achieved success.
- **Tactical Approach** starts introducing partially SPL concepts in sub-process and methods, starting from the most problematic sub-process. It is often used when architects and engineers drive this introduction.
- **Pilot Project Strategy** this strategy may be start using one of the several alternative ways, such as, starting as a potential first product, starting as a toy product, starting as a product prototyping.
- **Big Bang Strategy** the SPL adoption is done by the organization at once. The domains completely performed and the platform is built, after that, the PD starts and the products are derived from the platform.

Another point of view is presented by Krueger (2002), which advocates three adoption models: using the *proactive approach*, organization analyzes, designs and implements the overall SPL to support the full scope of products needed on the foreseeable horizon. In *reactive approach*, the organization incrementally grows their SPL as the demand arises for new products or new requirements on existing products. Finally, using *extractive approach* the organization capitalizes on existing custom software systems by extracting the common and varying source code into a single production line.

2.5 Industrial Experiences with SPL

The reduction of cost and time to market, the improvement of product quality, and an increased responsiveness to change the technology and customer requirements, are all

critical issues that companies must face to be competitive in today's market (Sellier *et al.*, 2007). Software product line engineering has showed to be an efficient way to achieve these goals.

Because software product line engineering requires long-term planning, the companies that have used it successfully are, often, large ones that can afford to take the long view (Knauber *et al.*, 2000). Table 2.1 shows some successful industrial cases of applying the software product line engineering paradigm are summarized, by describing the previous and current scenario, some challenges and some results and metrics.

Table 2.1 Software Product Line Industrial Cases (Pohl *et al.*, 2005a; Linden *et al.*, 2007).

Company	Case Study	Previous Scenario	Challenges	Current Scenario	Results and Metrics
CELSIUS TECH	Ship System 2000	Systems comprise 1-1.5 million SLOC of Ada code, are hard-real-time, embedded, and safety-critical.	Support systems with more than fifty variants; Create a family members including systems for ships from coastal corvettes to cruisers to submarines; Complexity increase; Late system testing; Not a trivial assembling process.	Able to slash production time, build more systems with fewer people, and increase quality	Decrease time to market - 9 years to 3 years; Reduce costs; Schedule on time and predictable met; Reduce the number of developers - 210 to 30; Stable architecture from start of new project.
Naval Undersea Warfare Center	A-7E Operational Flight Program	The NUWC develops and supports different range facilities, including those to test and evaluate systems for the military forces of the USA; In the past, these range facilities were built for specific categories of weapon systems and missions, but these systems have become more and more complex.	Manage the commonality and complexity of the range facilities; Structure the software product line by a reference architecture intended to cover the complete set of range operations; Use the reference architecture for building range systems, some assets have to be tailored for range-unique capabilities.	In the year 2004, the software product line included seven systems already installed, with five to six new projects per year.	Cost reduce about 50% using RangeWare SPL; The development time has been reduced from years to months; Staff resources are cut by up to 75%; Increase customer satisfaction.
CUMMINS Inc.	Diesel engine SPL	Modern engines can contain over 100KSLOC of software to micro-control ignition to produce an optimum mix of power, economy, and emissions.	In 1993, faced with the need to produce almost 20 new systems but with staff and resources available only for six.	TheCummins SPL covers 9 basic engine types ranging over 4- 18 cylinders and 4-164 liters of displacement, with 12 kinds of electronic control modules, 5 kinds of processors, and 10 kinds of fuel systems.	To date, 20 basic software builds have been parlayed into well over 1000 separate products; Cycle time has been reduced from around 250 person months to a few person months; Productivity improvement of 3.6, and an ROI of 10:1.
General Motors	General Motors Powertrain	Powertrains consist of an engine, transmissions and the associated control system; In the control system, there are electrical components, an electronic control module, and the software that runs this system (General Motors Powertrain SPL).	GMPT began its transition to a product line approach for its embedded powertrain control software in the late 1990's.	Controller products built using the GMPT software product line cleanly interface with over 100 vehicle platforms.	The GMPT software product line is now the basis for nearly all new control modules being developed within GMPT; GMPT expects to take the number of software sets supporting gasoline engine programs from 17 down to 3.
Nokia	Mobile Phones	The initial software architecture for this product line addressed variations in hardware, communication standards, and user interfaces.	Language Challenge Abstract; The Hardware Challenge; The Feature Challenge.	32 different phones are manufactured covering six different protocol standards, a wide variety of functional features and capabilities, different user interface designs, and many platforms and environments.	Nokia Mobile Phones is the world's largest mobile phone manufacturer, and they believe that software product line engineering has helped it to reach that position.

2.6 Chapter Summary

Software Product Lines is an approach to software reuse that during the last years has proven its applicability in a broad range of situations, producing impressive results (Weiss and Krueger, 2006). To achieve all software product lines benefits, three essential activities must be followed: Core Asset Development, Product Development and Management. The assets are created during core asset development phase and further instantiated during product development to derive products.

In this chapter an overview of SPL was presented, discussing its essential activities, the concept regarding to variability and how to manage it, as well as, some SPL adoption strategies. It also presents some successful industrial cases of applying the software product line engineering approach.

The next Chapter presents an overview on the software testing area discussing their fundamental concepts, testing levels, testing strategies and some black-box and white-box methods in order to define a base for the approach defined in this work.

“Testing can never demonstrate the absence of errors in software, only their presence.”

E.W.Dijkstra

3

Overview on Software Testing

3.1 Introduction

The growing development of complex and large software products requires many activities that need to be suitably coordinated to meet the desired customer requirements. Two set of activities are required to achieve this goal, activities responsible for the development of software products, and activities which aims at checking the quality of both, the development process and artifacts. The set of activities related to evaluate and verify the quality of products, is often referred as testing or quality process.

Software testing is an iterative process that tends to be considered a part of development, it is really its own discipline and should be tracked as its own project. While working closely with development, it should be independent enough to be able to cancel or delay product delivery if the quality requirements are not met.

This closely relation between software testing and development is not punctual but it spans through the whole software life-cycle: it starts with non-executable artifacts (e.g. requirements, architecture design, documents, etc.) using reviews, inspections and walkthroughs, next in executable artifacts (code) and goes beyond product deployment (maintenance) and post mortem analysis (Baresi and Pezzè, 2006).

The rest of the chapter discusses several important issues of testing and is organized as follows. Section 3.2 introduces some software testing fundamental concepts. Section 3.3 discusses the testing process importance and its main activities. Section 3.4 presents the V-model and describes its main levels. Section 3.5 approaches the regression testing technique. Section 3.6 discusses about the two software testing strategies, the black-box and white-box. Section 3.6.1 and 3.6.2 describe some black-box and white-box methods. Section 3.7 summarizes the relation between SPL and software testing, and, finally, Section 3.8 summarizes this chapter.

3.2 Fundamental Concepts

Some concepts and terminology are important to understand and apply a software testing process. In this section, they are described.

- **Validation and Verification**

Software testing is a broader topic that is often referred as validation and verification. One of the most important distinction to make is, *validation* refers to the process of evaluating at the end of software development to ensure compliance with intended usage, “*Are we building the right product?*”. *Verification* is the process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase, “*Are we building the product right?*” (Ammann and Offutt, 2008).

- **Static and Dynamic Analysis**

By the way, there are two ways to evaluate the software and their artifacts: through *static analysis* which verifies against a static specification (structure of the artifact), are not performed in executable code and can be executed manually or with a set of tools; and, using *dynamic analysis* methods, where the software is executed using a set of inputs and comparing their output behavior to what is expected and they are performed in executable (Burnstein, 2003). The use of static analysis at the beginning, is important as means to identify problems as early as possible, since a later identification can impact the project as a whole, increasing its cost.

- **Error, Fault and Failure**

Another important concept regarding to software testing are the distinction among *error, fault and failure* (Figure 3.1). The first, refers to a mistake, misconception, or misunderstanding by a developer, a manifestation of some fault. The second, it is a software anomaly that may cause it to work incorrectly, and not according to its specification, a static defect. It is introduced into the software as a result of an error. The later, comprehends the inability of a system or component to perform its required functions as established by the requirements (Burnstein, 2003).

System Specification

The system should calculate $(3 * a) ^ 4 / 2$ for a given input a .

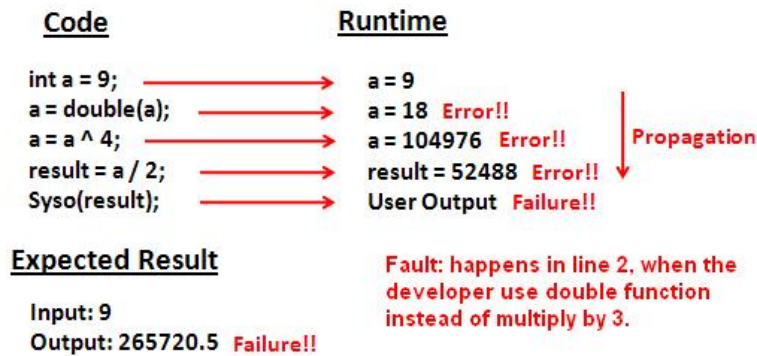


Figure 3.1 Difference among Error, Fault and Failure

3.3 The Testing Process

Testing concepts, strategies, techniques, and measures need to be integrated into a defined and controlled process which is applied by people (Abran *et al.*, 2004).

Some organizations postpone software activities to the end of the development. It makes testing be compressed, not enough resources remains (time and budget), problems from previous stages are solved by taking time and money and the managers do not have enough time to plan for testing. The testers cannot find errors at the last minute and make a bad product good, thus, high quality should be part of the process from the beginning (Ammann and Offutt, 2008).

The testing process starts even if no code, through static analysis (walkthroughs and revisions) and continues with dynamic analysis (with executable code) interacting with the development phases. The integration between testing activities and the software development lifecycle, can make dramatic improvements in the effectiveness and efficiency of testing, and influence the software development process in such a way that high quality software is more likely to be built (Ammann and Offutt, 2008).

The testing process involve four main activities (Figure 3.2), they are performed in each testing level (Burnstein, 2003). The *Planning*, involve the coordination of personnel, management of available test facilities and hardware. The *Test Design* is based on the testing level to be performed and the particular testing techniques. Besides the test cases (including input data, and expected outputs for each test case), test scripts and test suites

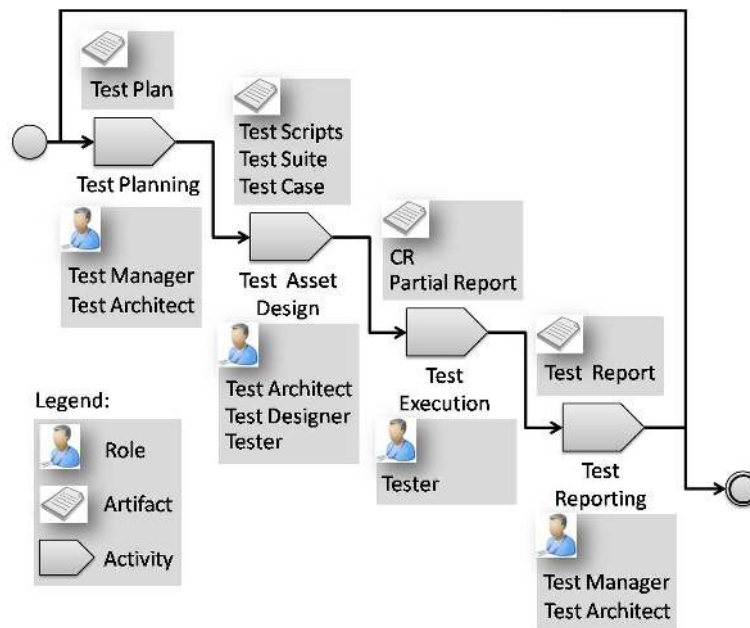


Figure 3.2 Testing level activities.

are also developed during this activity. The *Execution*, comprehends all steps necessary to run the test cases and scripts created previously. An automatic execution is invited in order to reduce time and cost. During the *Reporting* all testing execution results must be evaluated to determine whether or not the test has been software performed as expected and did not have any major unexpected outcomes. This activity is important to calibrate the testing level activity, identifying problems during test planning, design or even execution.

3.4 Testing Levels

In order to represent the association between development and testing phases, the V-model (Figure 3.3) is adopted. It was defined in the 1980s, in reaction to the waterfall model of software development (Rook, 1986).

Figure (3.3), shows how a testing phases interact with its respective development stage. Each of the testing phases will be detailed in the next subsections.

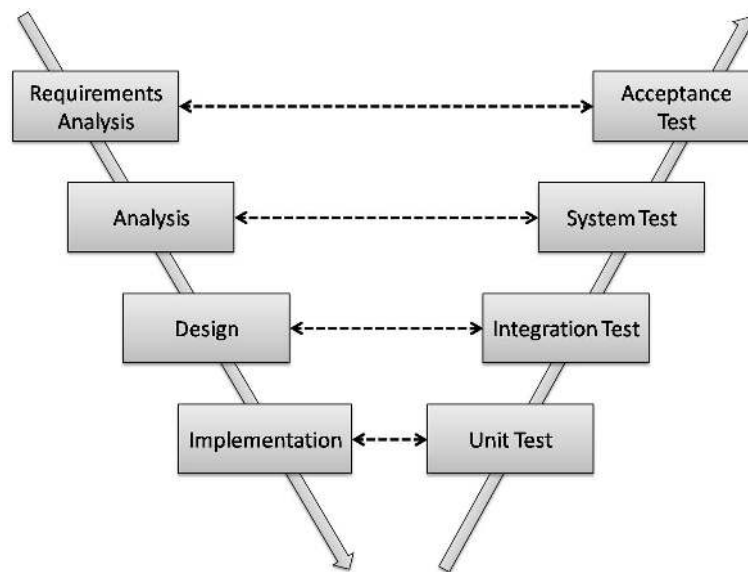


Figure 3.3 The V-Model.

3.4.1 Unit Testing

It is the lowest level of testing (Ammann and Offutt, 2008), and its main goal is to detect functional and structural defects in the unit, insuring that each individual software unit is functioning according to its specification. A unit is the smallest possible testable software component, in procedural programming a unit is traditionally viewed as a function or procedure, in object-oriented systems both classes and methods and a unit may also be considered a component (COTS) (Burnstein, 2003).

The decision about which consider as a unit is extremely important and impacts the whole test process. In case where the method is considered a unit, it may interact with other methods within a class, in some cases additional code, called *test harness* (e.g. *stubs and drivers*), must be developed to represent the absent methods within the class. The lower the granularity, the higher the test harness, consequently higher the test cost. This decision can also influence the type of defects found during the tests, for instance, considering a class, the defects due to encapsulation, polymorphism and inheritance can be detected.

Unit test is crucial since it exercises a small and simple portion of a software making easier to locate and repair a defect. It is important that the unit tests should be performed by an independent team (other than developers). In many cases, the unit tests are performed informally by a developer, under this ad-hoc approach, defects are not recorded

by developers and do not become part of the unit history. It can cause troubles during safely critical tasks and reuse (Burnstein, 2003).

3.4.2 Integration Testing

The integration level is responsible for detecting defects that occur in the units interface, this is also where object-oriented features (e.g.: inheritance, polymorphism and dynamic binding) are tested (Ammann and Offutt, 2008). The integration tests should only be performed in units that have been passed through unit tests.

The integration tests can be performed using two strategies, incremental (top-down and bottom-up) and non-incremental (also called Big-Bang) (Muccini and van der Hoek, 2003). In both strategies, only one unit is added to the growing subsystem or cluster. While in non-incremental strategy all units are integrated to be further tested, in incremental strategy, a unit is integrated into a set of previously integrated modules (set of units) which were prior approved (Burnstein, 2003). The last strategy can be performed in two ways, **top-down** or **bottom-up** strategies, they are used with traditional, hierarchically structured software (Abran *et al.*, 2004).

3.4.3 System Testing

The goal of this level is to ensure that the system performs according to its requirements. Evaluating both functional behavior and quality requirements such as, security, performance and reliability. This level is useful for detecting external hardware and software interface defects, for example, those causing race conditions, deadlocks and exception handling.

Several types of system tests are shown in Figure 3.4 and following described, some inputs are also displayed.

- *Functional Tests*: They are responsible for ensure that the behavior of the system adheres to its requirements specification, all functional requirements should be achievable by the system.
- *Stress and Load Tests*: It aims to try to break the system, finding scenarios under which it will crash. Race conditions are deadlocks often uncovered by performing stress testing.
- *Security Tests*: Insure that the system is safe and secure is a big task, performed by developers and test specialists. This type of testing aims to evaluate systems

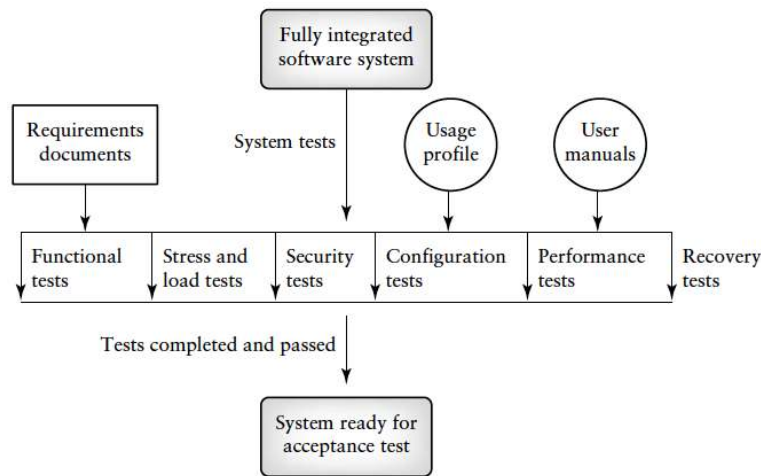


Figure 3.4 Types of System Tests (Burnstein, 2003).

characteristics related to availability, integrity and confidentiality of system services and data.

- *Configuration Tests*: It allows developers and testers evaluate the system performance and availability when hardware exchanges and reconfigurations happens. In additional, it shows the correct operation of configuration menus and commands.
- *Performance Tests*: It aims to test non-functional requirements (quality attributes) which describes quality levels expected for the software, for example, memory use, response time and delays. In a general way, it sees if the software meets the performance requirements.
- *Recovery Tests*: Comprehends the tests responsible for determine if the system could return to a well-known state, without compromise any previous transaction.

3.4.4 Acceptance Testing

After the software has passed through the system test level, acceptance tests allow users to evaluate the software in terms of their expectations and goals. The acceptance tests are based on requirements, user manual or even system tests. In this level, the software must be executed under the real-world conditions on operational hardware and software.

When the software is developed for mass market (for example, COTS), testing it for individual clients is not practical, in this case, two stages of acceptance tests are applied. The first, called alpha tests, takes place on developer's site "in-house" (Abran *et al.*,

2004). The last, beta tests, sends the software to users who install it and use it under real-world conditions (Burnstein, 2003).

3.5 Regression Testing

Regression testing is not considered a testing level, but it is a technique used to retest a software when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes (Burnstein, 2003). It can be applied in any testing level (Abran *et al.*, 2004).

Changes to software are often classified as corrective, perfective, adaptive, and preventive. All of these changes require regression testing (Ammann and Offutt, 2008). It will be detailed in Chapter 6.

3.6 Testing Strategies

In order to maximize the use of time and resources, the test cases must be effective, having a good possibility of revealing defects. The main benefits of achieve this efficiency, are: (i) efficient use of organizational resources, (ii) higher probability for test reuse, (iii) closer adherence to testing and project schedules and budgets and (iv) a higher-quality software product delivery (Burnstein, 2003).

Two basic strategies are used to help the test case design, the *black-box* (or functional) and *white-box* (clear or glass-box), as is shown in Figure 3.5. Both strategies should be used to achieve a high-quality software, they complement each other.

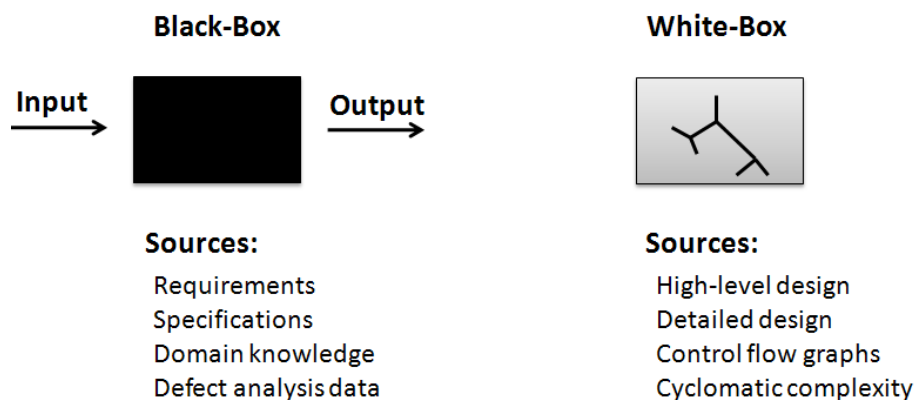


Figure 3.5 Testing Strategies (Burnstein, 2003).

- *Black-Box*: testing strategy that considers external descriptions of the software, including specifications, requirements, and design to design test cases. The internal program structure is not used. In addition, this strategy supports testing the product against the end user external specifications. It is usually applied to smaller-sized pieces of software such as module or function (Burnstein, 2003; Ammann and Offutt, 2008).
- *White-Box*: strategy that requires knowledge of the internal structure, the source code internals of the software, specifically including branches, individual conditions, and statements in order to design test cases. The size of the software under test using this approach can vary from a simple module, member function, or a complete system (Burnstein, 2003; Ammann and Offutt, 2008).

3.6.1 Black-Box Testing Methods

Assuming that infinite time and resources are not available to test all possible inputs, it is prohibitively expensive. For this reason, it is necessary to select a set of inputs (valid or invalid) in order to design effective test cases that gives the maximum yield of defects for time and effort spent. In order to help this test case design, a combination of methods are used to detect different types of defects. A set of black-box methods are described next (Burnstein, 2003).

- *Random Testing*: When a tester randomly selects inputs from the domain in order to execute the test. Although random test inputs may save some time and effort that more thoughtful input selection methods require, this selection has little chance of producing an effective set of data.
- *Equivalent Class Partitioning*: Comprehends the partitioning of the input domain of the software under-test. The finite number of equivalence classes allows the tester to select a given member of an equivalence class as a representative of that class.
- *Boundary Value Analysis*: Requires that the tester select elements close to the edges, so that both the upper and lower edges of an equivalence class are covered by test cases. These values are often valuable in revealing defects and it is used to refine the results of equivalence class partitioning.
- *Cause-and-effect Graphing*: It is a technique used to combine conditions and derive an effective set of test cases that may disclose inconsistencies in a specification.

One advantage of this method come from exercising combinations of test data may not be considered using other black-box testing techniques.

- *State Transition Testing:* Based on states and finite-state machines this method allow the tester to view the developing software in term of its states, transitions between states, and the inputs and events that trigger state changes.
- *Error Guessing:* Based on developer/tester past experience with similar code under-test, and their intuition regarding to where the defects are in the code. A high expertise is required.

3.6.2 White-Box Testing Methods

When the tester has knowledge of the internal logic structure of the software under test, a white-box testing method is invited. The goal of the tester is determine if all the logical and data elements in the software unit are working properly. It is most useful when testing small components. Some white-box methods are following described (Burnstein, 2003).

- *Statement Testing:* It aims to test the statements of a module under test. If the statement coverage criterion is set to 100%, the tester should develop a set of test cases, that when executed, all of the statements in the module are executed at least once.
- *Branch Testing:* A similar idea can be viewed here, instead of all statements, only decision elements in the code (if-then, case, loop) are executed.
- *Path Testing:* The tester first identifies a set of independent paths, sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node, following design test cases for each path.
- *Loop Testing:* The purpose of this method is to verify loop constructs which can be classified in four categories, simple, nested, concatenated and unstructured.

3.7 SPL and Software Testing

Software product lines promise benefits such as improvements in time to market, cost reduction, high productivity and quality (Clements and Northrop, 2001). These goals will only be achieved if quality attributes (correctness and reliability) are continuing

objectives from the earliest phases of development (Ammann and Offutt, 2008; McGregor, 2001b). Thus, a product line organization should define a set of activities that validate the correctness of what has been built and that verify that the correct product has been built. Software testing is one approach to validate and verify the artifacts produced in software development.

Testing in a product line organization includes activities from the validation of the requirements to verification activities carried out by customers to complete the acceptance of a product. In the SPL context, it includes testing of core assets, responsible for verify the common parts (commonality) among products, the product development testing which aims to verify the product specific parts (variability) and the interaction between them.

The same opportunities for large-scale reuse exist for assets created to support the testing process as for assets created for development. Since the cost of all of the test assets for a project can approach that for the development assets, savings from the reuse of test assets and savings from testing early in the development process can be just as significant as savings from development assets (McGregor, 2001b).

3.8 Chapter Summary

Testing in the context of a product line includes testing the core assets software, the product specific software, and their interactions. Testing is conducted within the context of the other development activities. In this chapter, some fundamental concepts, testing levels, strategies and methods were presented. The chapter also discussed about SPL and testing.

In order to better understand the SPL and Testing state-of-the-art, a mapping study was performed and presented in the next chapter.

“Have a Healthy Disregard for the Impossible.”

Larry Page

4

A Mapping Study on Software Product Line Testing

In software development, Testing is an important mechanism both to identify defects and assure that completed products work as specified. This is a common practice in single-system development, and continues to hold in Software Product Lines (SPL). Even though extensive research has been done in the SPL Testing field, it is necessary to assess the current state of research and practice, in order to provide practitioners with evidence that enable fostering its further development. This chapter focuses on Testing in SPL and has the following goals: investigate state-of-the-art testing practices, synthesize available evidence, and identify gaps between required techniques and existing approaches available in the literature. A systematic mapping study Petersen *et al.* (2008), which is an evidence-based approach, applied in order to provide an overview of a research area, and identify the quantity and type of research and results available within it, was conducted with a set of nine research questions, in which 120 studies, dated from 1993 to 2009, were evaluated. Although several aspects regarding testing have been covered by single-system development approaches, many can not be directly applied in the SPL context due to specific issues. In addition, particular aspects regarding SPL are not covered by the existing SPL approaches, and when the aspects are covered, the literature just gives brief overviews. This scenario indicates that additional investigation, empirical and practical, should be performed. The results can help to understand the needs in SPL Testing, by identifying points that still require additional investigation, since important aspects regarding particular points of software product lines have not been addressed yet.

The remainder of this chapter is organized as follows: In Section 4.2, the method used in this study is described. Section 4.3, presents the planning phase and the research

questions addressed by this study. Section 4.4, describes its execution, presenting the search strategy used and the resultant selected studies. Section 4.5, presents the classification scheme adopted in this study and reports the findings. In section 4.6, the threats to validity are described, Section 4.7, presents the related work. Section 4.8, draws some conclusions and provides recommendations for further research on this topic. Section 4.9 presents the chapter summary.

4.1 Introduction

The increasing adoption of Software Product Lines practices in industry has yielded decreased implementation costs, reduced time to market and improved quality of derived products Denger and Kolb (2006); Northrop and Clements (2007). In this approach, as in single-system development, testing is essential Kauppinen (2003) to uncover defects Pohl and Metzger (2006); Reuys *et al.* (2006). A systematic testing approach can save significant development effort, increase product quality and, customer satisfaction and lower maintenance costs Juristo *et al.* (2006a).

As defined in McGregor (2001b), testing in SPL aims to examine core assets, shared by many products derived from a product line, their individual parts and the interaction among them. Thus, testing in this context encompasses activities from the validation of the initial requirements to activities performed by customers to complete the acceptance of a product, and confirms that testing is still the most effective method of quality assurance, as observed in Kolb and Muthig (2003).

However, despite the obvious benefits aforementioned, the state of software testing practice is not as advanced in general as software development techniques Juristo *et al.* (2006a) and, the same holds true in the SPL context Kauppinen and Taina (2003); Tevanlinna *et al.* (2004). From an industry point of view, with the growing SPL adoption by companies Weiss (2008), more efficient and effective testing methods and techniques for SPL are needed, since the currently available techniques, strategies and methods make testing a very challenging process Kolb and Muthig (2003). Moreover, the SPL Testing field has attracted the attention of many researchers in the last years, which result in a large number of publications regarding general and specific issues. However, the literature has provided lots of approaches, strategies and techniques, but rather surprisingly little in the way of widely-known empirical assessment of their effectiveness.

This chapter presents a systematic mapping study Petersen *et al.* (2008), performed in order to map out the SPL Testing field, through synthesizing evidence to suggest

important implications for practice, as well as identifying research trends, open issues, and areas for improvement. Mapping study Petersen *et al.* (2008) is an evidence-based approach, applied in order to provide an overview of a research area, and identify the quantity and type of research and results available within it. The results are gained from a defined approach to locate, assess and aggregate the outcomes from relevant studies, thus providing a balanced and objective summary of the relevant evidence. Hence, the goal of this investigation is to identify, evaluate, and synthesize state-of-the-art testing practices for product lines in order to present what has been achieved so far in this discipline. We are also interested in identifying practices adopted in single systems development that may be suitable for SPL.

The study also highlights the gaps and identifies trends for research and improvements. Moreover, it is based on analysis of interesting issues, guided by a set of research questions. This systematic mapping process was conducted from July to December in 2009.

4.2 Literature Review Method

The method used in this research is a Systematic Mapping Study (henceforth abbreviated to as 'MS') (Budgen *et al.*, 2008; Petersen *et al.*, 2008). A MS provides a systematic and objective procedure for identifying the nature and extent of the empirical study data that is available to answer a particular research question Budgen *et al.* (2008).

While a Systematic Review (SR) is a mean of identifying, evaluating and interpreting all available research relevant to a particular question Kitchenham and Charters (2007), a MS intends to 'map out' the research undertaken rather than to answer detailed research questions (Budgen *et al.*, 2008; Petersen *et al.*, 2008). A well-organized set of good practices and procedures for undertaking MS in the software engineering context is defined in (Budgen *et al.*, 2008; Petersen *et al.*, 2008), which establishes the base for the study presented in this paper. It is worthwhile to highlight that the importance and use of MS in the software engineering area is increasing (Afzal *et al.*, 2008; Bailey *et al.*, 2007; Budgen *et al.*, 2008; Condori-Fernandez *et al.*, 2009; Juristo *et al.*, 2006b; Kitchenham, 2010; Petersen *et al.*, 2008; Pretorius and Budgen, 2008), showing the relevance and potential of the method. Nevertheless, of the same way as systematic reviews (Bezerra *et al.*, 2009; Chen *et al.*, 2009; Lisboa *et al.*, 2010; Moraes *et al.*, 2009), we need more MS related to software product lines, in order to evolve the field with more evidence Kitchenham *et al.* (2004).

A MS comprises the analysis of primary studies that investigate aspects related to predefined research questions, aiming at integrating and synthesizing evidence to support or refute particular research hypotheses. The main reasons to perform a MS can be stated as follows, as defined by Budgen *et al.* (2008):

- To make an unbiased assessment of as many studies as possible, identifying existing gaps in current research and contributing to the research community with the reliable synthesis of the data;
- To provide a systematic procedure for identifying the nature and extent of the empirical study data that is available to answer research questions;
- To map out the research that has been undertaken;
- To help to plan new research, avoiding unnecessary duplication of effort and error;
- To identify gaps and clusters in a set of primary studies, in order to identify topics and areas to perform more complete systematic reviews.

The experimental software engineering community is working towards the definition of standard processes for conducting mapping studies. This effort can be checked out in Petersen *et al.* (2008), a study describing how to conduct systematic mapping studies in software engineering. The paper provides a well defined process which serves as a starting point for our work. We merged ideas from Petersen *et al.* (2008) with good practices defined in the SR guidelines published by Kitchenham and Charters (2007). This way, we could apply a process for mapping study including good practices of conducting systematic reviews, making better use of the both techniques.

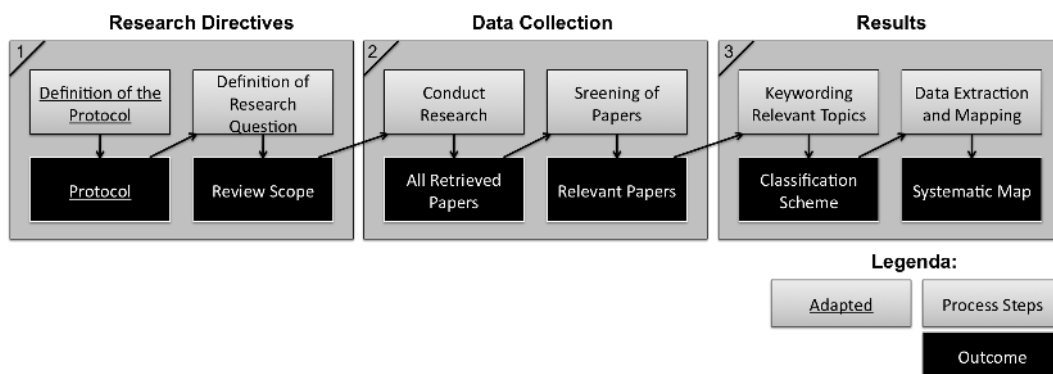


Figure 4.1 The Systematic Mapping Process (adapted from Petersen *et al.* (2008)).

This blending process enabled us to include topics not covered by Petersen *et al.* (2008) in their study, such as:

- **Protocol.** This artifact was adopted from systematic review guidelines. Our initial activity in this study was to develop a protocol, i.e. a plan defining the basic mapping study procedures. Searching in the literature, we noticed that some studies created a protocol (e.g. (Afzal *et al.*, 2009)), but others do not (e.g. Condori-Fernandez *et al.* (2009); Petersen *et al.* (2008)). Even though this is not a mandatory artifact, as mentioned by Petersen *et al.* (2008), authors who created a protocol in their studies encourage the use of this artifact as being important to evaluate and calibrate the mapping study process.
- **Collection Form.** This artifact was also adopted from systematic review guidelines and its main purpose is to help the researchers in order to collect all the information needed to address the review questions, study quality criteria and classification scheme.
- **Quality Criteria.** The purpose of quality criteria is to evaluate the studies, as a means of weighting their relevance against others. Quality criteria are commonly used when performing systematic literature reviews. The quality criteria were evaluated independently by two researchers, hopefully reducing the likelihood of erroneous results.

Some elements, as proposed by Petersen *et al.* (2008) were also changed and/or rearranged in this study, such as:

- **Phasing mapping study.** As can be seen in Figure 4.1, the process was explicitly split into three main phases: *1 - Research Directives*, *2 - Data Collection* and *3 - Results*. It is in line with systematic reviews practices Kitchenham and Charters (2007), which defines *planning*, *conducting* and *reporting* phases. Phases are named differently from what is defined for systematic reviews, but the general idea and objective for each phase was followed. In the first, the protocol and the research questions are established. This is the most important phase, since the research goal is satisfied with answers to these questions. The second phase comprises the execution of the MS, in which the search for primary studies is performed. This consider a set of inclusion and exclusion criteria, used in order to select studies that may contain relevant results according to the goals of the research. In third phase, the classification scheme is developed. The results of a meticulous analysis

performed with every selected primary study is reported, in a form of a mapping study. All phases are detailed in next sections.

4.3 Research Directives

This section presents the first phase of the mapping study process, in which the protocol and research questions are defined.

4.3.1 Protocol Definition

The protocol forms the research plan for an empirical study, and is an important resource for anyone who is planning to undertake a study or considering performing any form of replication study.

In this study, the purpose of the protocol is to guide the research objectives and clearly define how it should be performed, through defining research questions and planning how the sources and studies selected will be used to answer those questions. Moreover, the classification scheme to be adopted in this study was prior defined and documented in the protocol.

Incremental reviews to the protocol were performed in accordance with the MS method. The protocol was revisited in order to update it based on new information collected as the study progressed.

To avoid duplication, we detail the content of the protocol in the Section 4.4, as we describe how the study was conducted.

4.3.2 Question Structure

The research questions were framed by three criteria:

- **Population.** Published scientific literature reporting software testing and SPL testing.
- **Intervention.** Empirical studies involving SPL Testing practices, techniques, methods and processes.
- **Outcomes.** Type and quantity of evidence relating to various SPL testing approaches, in order to identify practices, activities and research issues concerning to this area.

4.3.3 Research Questions

As previously stated, the objective of this study is to understand, characterize and summarize evidence, identifying activities, practical and research issues regarding research directions in SPL Testing. We focused on identifying *how the existing approaches deal with testing in SPL*. In order to define the research questions, our efforts were based on topics addressed by previous research on SPL testing (Odia, 2007; Kolb and Muthig, 2003; Tevanlinna *et al.*, 2004). In addition, the research questions definition task was aided by discussions with expert researchers and practitioners, in order to encompass relevant and still open issues.

Nine research questions were derived from the objective of the study. Answering these questions led a detailed investigation of practices arising from the identified approaches, which support both industrial and academic activities. The research questions, and the rationale for their inclusion, are detailed below.

- **Q1. Which testing strategies are adopted by the SPL Testing approaches?**
This question is intended to identify the testing strategies adopted by a software product line approach Tevanlinna *et al.* (2004). By strategy, we mean the way in which the assets are tested, considering the differentiation between the two SPL development processes: core asset and product development.
- **Q2. What are the existing static and dynamic analysis techniques applied on the SPL context?** This question is intended to identify the analysis type (*static* and *dynamic testing* McGregor (2001b)) applied along the software development life cycle.
- **Q3. Which testing levels commonly applicable in single-systems development are also used in the SPL approaches?** Ammann and Offutt (2008) and Jaring *et al.* (2008) advocate different levels of testing (unit, integration, system and acceptance tests) where each level is associated with a development phase, emphasizing development and testing equally.
- **Q4. How do the product line approaches handle regression testing along software product line life cycle?** Regression testing is done when changes are made to already tested artifacts (Kauppinen, 2003; Rothermel and Harrold, 1996), to be confident that no new faults were inserted and the new software version still working properly. Thus, this question investigates the regression techniques applied to SPL.

- **Q5. How do the SPL approaches deal with tests of non-functional requirements?** This question seeks clarification on how tests of non-functional requirements should be handled.
- **Q6. How do the testing approaches in an SPL organization handle commonality and variability?** An undiscovered defect in the common core assets of a SPL will affect all applications and thus will have a severe effect on the overall quality of the SPL Pohl and Metzger (2006). In this sense, answering this question requires an investigation into how the testing approaches handle commonality issues through the software life cycle, as well as gathering information on how variability affects testability.
- **Q7. How do variant binding times affect SPL testability?** According to Jaring *et al.* (2008), variant binding time determines whether a test can be performed at a given development or deployment phase. Thus, the identification and analysis of the suitable moment to bind a variant determines the appropriate testing technique to handle the specific variant.
- **Q8. How do the SPL approaches deal with test effort reduction?** The objective is to analyze within selected approaches the most suitable ways to achieve effort reduction, as well as to understand how they can be accomplished within the testing levels.
- **Q9. Do the approaches define any measures to evaluate the testing activities?** This question requires an investigation into the data collected by the various SPL approaches with respect to testing activities.

4.4 Data Collection

In order to answer the research questions, data was collected from the research literature. These activities involved developing a search strategy, identifying data sources, selecting studies to analyze, and data analysis and synthesis.

4.4.1 Search Strategy

The search strategy was developed by reviewing the data needed to answer each of the research questions.

The initial set of keywords was refined after a preliminary search returned too many results with few relevance. We used several combinations of search items until achieve a suitable set of keywords. These are: *Verification, Validation; Product Line, Product Family; Static Analysis, Dynamic Analysis; Variability, Commonality, Binding; Test Level; Test Effort, Test Measure; Non-functional Testing; Regression Testing, Test Automation, Testing Framework, Performance, Security, Evaluation, Validation*, as well as their similar nouns and syntactic variations (e.g. plural form). All terms were combined with the term "Product Line" and "Product Family" by using Boolean "AND" operator. They all were joined each other by using "OR" operator so that it could improve the completeness of the results. The complete list of *search strings* is available in Table 4.1 and also in a website developed to show detailed information on this MS¹.

Table 4.1 List of Search Strings

Research Strings	
1	verification AND validation AND ("product line" OR "product family" OR "SPL")
2	"static analysis" AND ("product line" OR "product family" OR "SPL")
3	"dynamic testing" AND ("product line" OR "product family" OR "SPL")
4	"dynamic analysis" AND ("product line" OR "product family" OR "SPL")
5	test AND level AND ("product line" OR "product family" OR "SPL")
6	variability OR commonality AND testing
7	variability AND commonality AND testing AND ("product line" OR "product family" OR "SPL")
8	binding AND test AND ("product line" OR "product family" OR "SPL")
9	test AND "effort reduction" AND ("product line" OR "product family" OR "SPL")
10	"test effort" AND ("product line" OR "product family" OR "SPL")
11	"test effort reduction" AND ("product line" OR "product family" OR "SPL")
12	"test automation" AND ("product line" OR "product family" OR "SPL")
13	"regression test" AND ("product line" OR "product family" OR "SPL")
14	"non-functional test" AND ("product line" OR "product family" OR "SPL")
15	measure AND test AND ("product line" OR "product family" OR "SPL")
16	"testing framework" AND ("product line" OR "product family" OR "SPL")
17	performance OR security AND ("product line" OR "product family" OR "SPL")
18	evaluation OR validation AND ("product line" OR "product family" OR "SPL")

4.4.2 Data Sources

The search included important journals and conferences regarding the research topic such as *Software Engineering, SPL, Software Verification, Validation and Testing and Software Quality*. The search was also performed using the 'snow-balling' process, following up the references in papers and it was extended to include grey literature sources, seeking relevant white papers, industrial (and technical) reports, thesis, work-in-progress, and books.

We restricted the search to studies published up to December 2009. We indeed did not establish an inferior year-limit, since our intention was to have a broader coverage

¹<http://www.cin.ufpe.br/~sople/testing/ms/>

of this research field. This was decided due to many important issues that emerged ten or more years ago are still considered open issues, as pointed out in Bertolino (2007); Juristo *et al.* (2004).

The initial step was to perform a search using the terms described in 4.4.1, at the digital libraries web search engines. We considered publications retrieved from ScienceDirect, SCOPUS, IEEE Xplore, ACM Digital Library and Springer Link tools.

The second step was to search within top international, peer-reviewed journals published by Elsevier, IEEE, ACM and Springer, since they are considered the world leading publishers for high quality publications Brereton *et al.* (2007).

Next, conference proceedings were also searched. In cases which the conference keep the proceedings in a website, making them available, we accessed the website. When proceedings were not available by the conference website, the search was done through DBLP Computer Science Bibliography ².

When searching conference proceedings and journals, many were the results that had already been found in the search through digital libraries. In this case, we discarded the last results, considering only the first, that had already been included in our results list.

The lists of *Conferences* and *Journals* used in the search for primary studies are available in Appendices B and B.2.

After performing the search for publications in conferences, journals, using digital libraries and proceedings, we noticed that known publications, commonly referenced by other studies in this field, such as important technical reports and thesis, had not been included in our results list. We thus decided to include these grey literature entries. **Grey literature** is used to describe materials not published commercially or indexed by major databases.

4.4.3 Studies Selection

The set of search strings was thus applied within the search engines, specifically in those mentioned in the previous section. The studies selection involved a screening process composed of three filters, in order to select the most suitable results, since the likelihood of retrieving not adequate studies might be high. Figure 4.2 briefly describes what was considered in each filter. Moreover, the Figure depicts the amount of studies remaining after applying each filter.

The inclusion criteria were used to select all studies during the search step. After that, the same exclusion criteria was firstly applied in the studies title and after in the abstracts

²<http://www.informatik.uni-trier.de/~ley/db/>

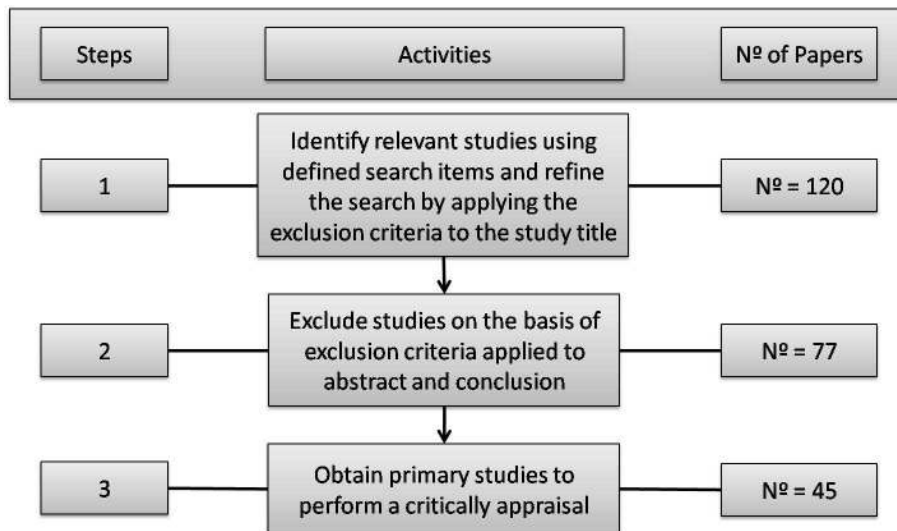


Figure 4.2 Stages of the selection process.

and conclusions. All excluded studies can be seen by differentiating the results among filters. Regarding the inclusion criteria, the studies were included if they involved:

- **SPL approaches which address testing concerns.** Approaches that include information on methods and techniques and how they are handled and, how variabilities and commonalities influence software testability.
- **SPL testing approaches which address static and dynamic analysis.** Approaches that explicitly describe how static and dynamic testing applies to different testing phases.
- **SPL testing approaches which address software testing effort concerns.** Approaches that describe the existence of automated tools as well as other strategies used in order to reduce test effort, and metrics applied in this context.

Studies were excluded if they involved:

- **SPL approaches with insufficient information on testing.** Studies that do not have detailed information on how they handle SPL testing concepts and activities.
- **Duplicated studies.** When the same study was published in different papers, the most recent was included.
- **Or if the study had already been included from another source.**

Figure 4.3 depicts a Bar Chart with the results categorized by source and filter, as described in section 4.4.2. Figure 4.4 shows the distribution of the primary studies, considering the publication year. This Figure briefly gives us the impression that the SPL Testing area is becoming more interesting, whereas the growing number of publications claims the trend that many solutions have become recently available (disregarding 2009, since many studies might not be made available by search engines until the time the search was performed, and thus we did not consider in this study).

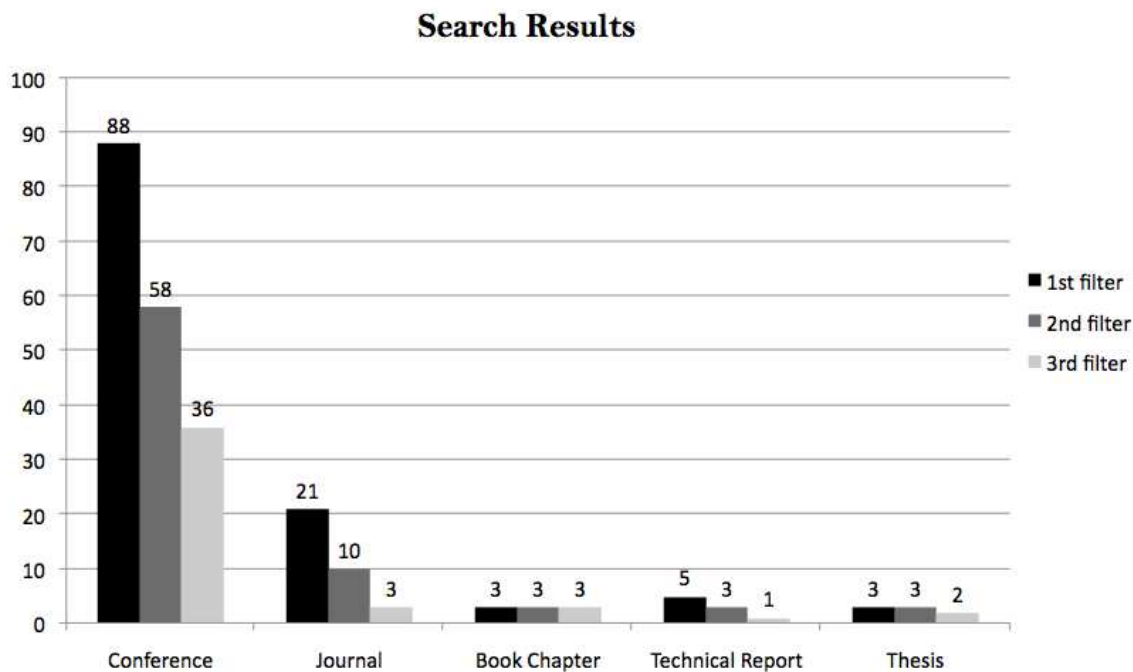


Figure 4.3 Primary studies filtering categorized by source.

An important point to highlight is that, between 2004 and 2008 an important international workshop devoted specifically to SPL testing, the SPLiT workshop³, demonstrated the interest of the research community on expanding this field. Figure 4.5 shows the amount of publications considering their sources. In fact, it can be seen that peaks in Figure 4.4 match with the years when this workshop occurred. All the studies are listed in Appendix C.

³c.f. <http://www.biglever.com/split2008/>

Distribution of primary studies

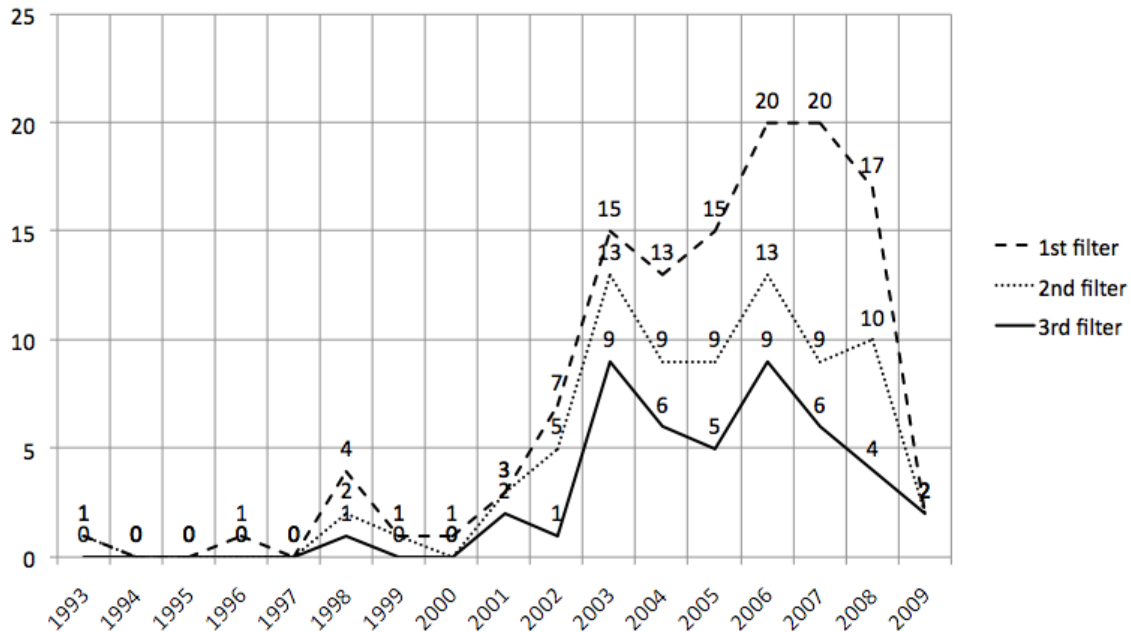


Figure 4.4 Distribution of primary studies by their publication years.

4.4.3.1 Reliability of Inclusion Decisions

The reliability of decisions to include a study is ensured by having multiple researchers to evaluate each study. The study was conducted by two research assistants who were responsible for performing the searches and summarizing the results of the mapping study, with other members of the team acting as reviewers. A high-level agreement was needed before the study was included. In case the researchers did not agree after discussion, an expert in the area was contacted to discuss and give appropriate guidance.

4.4.4 Quality Evaluation

In addition to general inclusion/exclusion criteria, the quality evaluation mechanism, usually applied in systematic reviews Dybå and Dingsøy (2008a,b); Kitchenham *et al.* (2007), was applied in this study in order to assess the trustworthiness of the primary studies. This assessment is necessary to limit any bias in conducting this empirical study, to gain insight into potential comparisons, and to guide interpretation of findings.

The quality criteria we used served as a means of weighting the importance of individual studies, enhancing our understanding, and developing more confidence in the

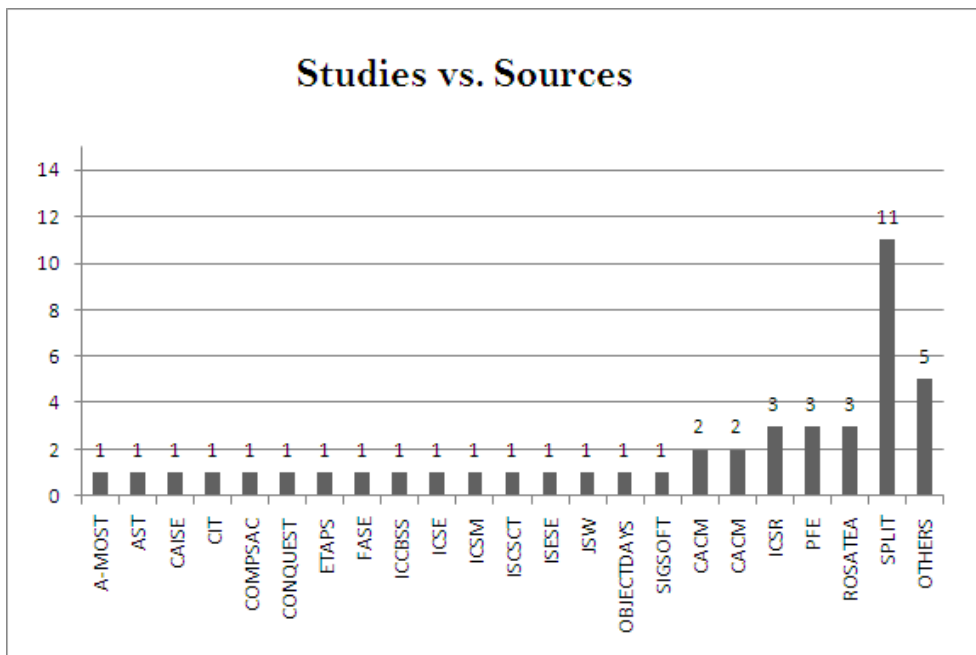


Figure 4.5 Amount of Studies vs. sources.

analysis.

Table 4.2 Quality Criteria

Group	ID	Quality Criteria
A	1	Are there any roles described?
	2	Are there any guideline described?
	3	Are there inputs and outputs described?
	4	Does it detail the test artifacts?
B	5	Does it detail the validation phase?
	6	Does it detail the verification phase?
	7	Does it deal with Testing in Requirements phase?
	8	Does it deal with Testing in Architectural phase?
	9	Does it deal with Testing in Implementation phase?
C	10	Does it deal with Testing in Deployment phase?
	11	Does it deal with binding time?
	12	Does it deal with variability testing?
	13	Does it deal with commonality testing?
	14	Does it deal with effort reduction?
	15	Does it deal with non-functional tests?
16	Does it deal with any test measure?	

As mapping study guidelines Petersen *et al.* (2008) does not establish a formal evaluation in the sense of quality criteria, we chose to assess each of the primary studies by principles of good practice for conducting empirical research in software engineering Kitchenham and Charters (2007), tailoring the idea of assessing studies by a set of criteria to our specific context.

Thus, the quality criteria for this evaluation is presented in Table 4.2. Criteria grouped as *A* covered a set of issues pertaining to quality that need to be considered when appraising the studies identified in the review, according to Kitchenham *et al.* (2002). This criteria group studies which handle process steps, roles, activities and artifacts. Groups *B* and *C* assess the quality considering SPL Testing concerns. The former was focused on identifying how well the studies address testing issues along the SPL development life cycle (e.g.: testing levels). The latter evaluated how well our research questions were addressed by individual studies. This way a better quality score matched studies which covered the larger amount of questions.

The main purpose of this grouping is justified by the difficulty faced in establishing a reliable relationship between final quality score and the real quality of each study. Some primary studies (e.g. one which addresses some issue in a very detailed way) are referenced in several other primary studies, but if we apply the complete quality criteria items, the final score is lower than others which do not have the same relevance. This way, we intended to have a more valid and reliable quality assessment instrument.

Each of the 45 studies was assessed independently by the researchers according to the 16 criteria shown in Table 4.2. Taken together, these criteria provided a measure of the extent to which we could be confident that a particular study could give a valuable contribution to the mapping study. Each of the studies was graded on a trichotomous (*yes*, *partly* or *no*) scale and tagged *1*, *0.5* and *0*. We did not use the grade to serve as a threshold for the inclusion decision, but rather to identify the primary studies that would form a valid foundation for our study. We note that, overall, the quality of the studies was good. It is possible to check every grade in Appendix C, where the most relevant are highlighted.

4.4.5 Data Extraction

The data extraction forms must be designed to collect all the information needed to address the research questions and the quality criteria. The following information was extracted from each study: *title and authors*; *source: conference/journal*; *publication year*; *the answers for research questions addressed by the study*; *summary: a brief overview on its strengths and weak points*; *quality criteria score* according to the Table 4.2; *reviewer name*; and *the date of the review*.

At the beginning of the study, we decided that when several studies were reported in the same paper, each relevant study was treated separately. Although, this situation did not occur.

4.5 Outcomes

In this section, we describe the classification scheme and the results of data extraction. When having the classification scheme in place, the relevant studies are sorted into the scheme, which is the real data extraction process. The results of this process is the *mapping of studies*, as presented at the end of this section, together with concluding remarks.

4.5.1 Classification Scheme

We decided to use the idea of categorizing studies in facets, as described by Petersen *et al.* (2008), since we considered this as a structured way of doing such a task. Our classification scheme assembled two facets. One facet structured the topic in terms of the research questions we defined. The other considered the type of research.

In the second, our study used the classification of research approaches described by Wieringa *et al.* (2006). According to Petersen *et al.* (2008), which also used this approach, the research facet which reflects the research approach used in the papers is general and independent from a specific focus area. The classes that form the research facet are described in Table 4.3.

The classification was performed after applying the filtering process, i.e. only the final set of studies was classified and are considered. The results of the classification is presented at the end of this section (Figure 4.8).

Table 4.3 Research Type Facet

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

4.5.2 Results

In this sub-section, each topic presents the findings of a sub-research question, highlighting evidences gathered from the data extraction process. These results populate the classification scheme, which evolves while doing the data extraction.

4.5.2.1 Testing Strategy

By analyzing the primary studies, we have found a wide variety of testing strategies. Reuys *et al.* (2006) and Tevanlinna *et al.* (2004) present a similar set of strategies to SPL testing development, that are applicable to any development effort since the descriptions of the strategies are generic. We herein use the titles of the topics they outlined, after making some adjustments, as a structure for aggregating other studies which use a similar approach, as follows:

- **Testing product by product:** This approach ignores the possibility of reuse benefits. This approach offers the best guarantee of product quality but is extremely costly. In Jin-hua *et al.* (2008), a similar approach is presented, named as *pure application strategy*, in which testing is performed only for a concrete product in the product development. No test is performed in the core asset development. Moreover, in this strategy, tests for each derived application are developed independently from each other, which results in an extremely high test effort, as pointed out by Reuys *et al.* (2006). This testing strategy is similar to the test in single-product engineering, because without reuse the same test effort is required for each new application.
- **Incremental testing of product lines:** The first product is tested individually and the following products are tested using regression testing techniques Graves *et al.* (2001); Rothermel and Harrold (1996). Regression testing focuses on ensuring that everything used to work still works, i.e. the product features previously tested are re-tested through a regression technique.
- **Opportunistic reuse of test assets:** This strategy is applied to reuse application test assets. Assets for one application are developed. Then, the application derived from the product line use the assets developed for the first application. This form of reuse is not performed systematically, which means that there is no method that supports the activity of selecting the test assets Reuys *et al.* (2006).

- **Design test assets for reuse:** Test assets are created as early as possible in domain engineering. Domain test aims at testing common parts and preparing for testing variable parts Jin-hua *et al.* (2008). In application engineering, these test assets are reused, extended and refined to test specific applications Jin-hua *et al.* (2008); Reuys *et al.* (2006). General approaches to achieve core assets reuse are: repository, core assets certification, and partial integration Zeng *et al.* (2004). Kishi and Noda Kishi and Noda (2006) state that a verification model can be shared among applications that have similarities. The SPL principle *design for reuse* is fully addressed by this strategy, which can enable the overall goals of reducing cost, shortening time-to-market, and increasing quality Reuys *et al.* (2006).
- **Division of responsibilities:** This strategy relates to select testing levels to be applied in both domain and application engineering, depending upon the objective of each phase, i.e. whether thinking about developing *for* or *with* reuse Tevanlinna *et al.* (2004). This division can be clearly seen when the assets are unit tested in domain engineering and, when instantiated in application engineering, integration, system and acceptance testing are performed.

As SPL Testing is a reuse-based test derivation for testing products within a product line, as pointed out by Zeng *et al.* (2004), the *Testing product by product* and *Opportunistic reuse of test assets* strategies cannot be considered “affordable” for the SPL context, since the first does not consider the reuse benefits which results in costs of testing resembling single-systems development. In the second, no method is applied, hence, the activity may not be repeatable, and may not avoid the redundant re-execution of test cases, which can thus increase costs.

These strategies can be considered a feasible grouping of what studies on SPL testing approaches have been addressing, which can show us a more generic view on the topic.

4.5.2.2 Static and Dynamic Analysis

An effective quality strategy for a software product line requires both static and dynamic analysis techniques. Techniques for static analysis are often dismissed as more expensive (cost for performing it in different products), but in a software product line, the cost of static analysis can be amortized over multiple products.

A number of studies advocate the use of inspections and walkthroughs Jaring *et al.* (2008); McGregor (2001b); Tevanlinna *et al.* (2004) and formal verification techniques, as static analysis techniques/methods for SPL, to be conducted prior to dynamic analysis, i.e.

with the presence of executable code. McGregor (2001b) presents an approach for Guided Inspection, aimed at applying the discipline of testing to the review of non-software assets. In Kishi and Noda (2006), a model checker is defined that focuses on design verification instead of code verification. This strategy is effective because many defects are injected during the design phase Kishi and Noda (2006).

Regarding dynamic analysis, some studies Jaring *et al.* (2008); Kolb and Muthig (2006) recommend the V-model phases, commonly used in single-systems, to structure a series of dynamic analysis. The V-model gives equal weight to development and testing rather than treating testing as an afterthought Goldsmith and Graham (2002). However, despite the well-defined test process presented by V-model, its use in SPL context requires some adaptation, as applied in Jaring *et al.* (2008).

The relative amount of dynamic and static analysis depends on both technical and managerial strategies. Technically, series of factors such as test-first development or model-based development determine the focus. Model-based development emphasizes static analysis of models while test-first development emphasizes dynamic analysis. Managerial strategies such as reduced time to market, lower cost and improved product quality determine the depth to which analysis should be carried.

4.5.2.3 Testing Levels

Some of the analyzed studies (e.g. Jaring *et al.* (2008); Kolb and Muthig (2006)) divide SPL testing according to the two primary software product line activities: *core asset* and *product development*.

Core asset development: Some testing activities are related to the development of test assets and test execution to be performed to evaluate the quality of the assets, which will be further instantiated in the application engineering phase. The two basic activities include developing test artifacts that can be reused efficiently during application engineering and applying tests to the other assets created during domain engineering Kamsties *et al.* (2003); Pohl *et al.* (2005b). Regarding types of testing, the following are performed in domain engineering:

- **Unit Testing:** Verification of the smallest unit of software implementation. This unit can be basically a class, or even a module, a function, or a software component. The granularity level depends on the strategy adopted. The purpose of unit testing is to determine whether this basic element performs as required through verification of the code produced during the coding phase.

- **Integration Testing:** This testing is applied as the modules are integrated with each other or within the reference in domain-level V&V when the architecture calls for specific domain components to be integrated in multiple systems. This type of testing is also performed during application engineering McGregor (2002). Li et al. Li *et al.* (2007a) present an approach for generating integration tests from unit tests.

Product development: Activities here are related to the selection and instantiation of assets to build specific product test assets, design additional product specific tests, and execute tests. The following types of testing can be performed in application engineering:

- **System Testing:** System testing ensures that the final product matches the required features Nebut *et al.* (2006). According to Geppert *et al.* (2004), system testing evaluates the features and functions of an entire product and validates that the system works the way the user expects. A form of system testing can be carried out on the software architecture using a static analysis approach.
- **Acceptance Testing:** Acceptance testing is conducted by the customer but often the developing organization will create and execute a preliminary set of acceptance tests. In a software product line organization, commonality among the tests needed for the various products is leveraged to reduce costs.

A similar division is stated by McGregor (2002), in which the author defines two separated test processes used in product line organization, *Core Asset Testing* and *Product Testing*.

Some authors Olimpiew and Gomaa (2005a); Reuys *et al.* (2006); Wübbeke (2008) also include *system testing* in core asset development. The rationale for including such a level is to produce abstract test assets to be further reused and adapted when deriving products in the product development phase.

4.5.2.4 Regression Testing

Even though regression testing techniques have been researched for many years, as stated in Engström *et al.* (2008); Graves *et al.* (2001); Rothermel and Harrold (1996), no study gives evidence on regression testing practices applied to SPL. Some information is presented by a few studies Kolb and Muthig (2003); Muccini and van der Hoek (2003), where just a brief overview on the importance of regression testing is given, but they do not take into account the issues specific to SPLs.

McGregor (2001b) reports that when a core asset is modified due to evolution or correction, they are tested using a blend of regression testing and development testing. According to him, the modified portion of the asset should be exercised using:

- Existing functional tests if the specification of the asset has not changed;
- If the specifications has changed, new functional tests are created and executed;
and
- Structural tests created to cover the new code created during the modification.

He also highlights the importance of regression test selection techniques and the automation of the regression execution.

Kauppinen and Taina (2003) advocate that the testing process should be iterative, and based on test execution results, new test cases should be generated and tests scripts may be updated during a modification. These test cases are repeated during regression testing each time a modification is made.

Kolb (2003) highlights that the major problems in a SPL context are the large number of variations and their combinations, redundant work, the interplay between generic components and product-specific components, and regression testing.

Jin-hua *et al.* (2008) emphasize the importance of regression testing when a component or a related component cluster are changed, saying that regression testing is crucial to perform on the application architecture, which aims to evaluate the application architecture with its specification. Some researchers also developed approaches to evaluate architecture-based software by using regression testing Harrold (1998); Muccini *et al.* (2005, 2006).

4.5.2.5 Non-functional Testing

Non-functional issues have a great impact on the architecture design, where predictability of the non-functional characteristics of any application derived from the SPL is crucial for any resource-constrained product. These characteristics are well-known quality attributes, such as response time, performance, availability, scalability, etc., that might differ in instances of a product line. According to Ganesan *et al.* (2005), testing non-functional quality attributes is equally important as functional testing.

By analyzing the studies, it was noticed that some of them propose the creation or execution of non-functional tests. Reis (2006) presents a technique to support the development of reusable performance test scenarios to be further reused in application

engineering. Feng *et al.* (2007) highlight the importance of non-functional concerns (performance, reliability, dependability, etc.). Ganesan *et al.* (2005) describe a work intended to develop an environment for testing the response time and load of a product line, however due to the constrained experimental environment there was no visible performance degradation observed.

In single-system development, different non-functional testing techniques are applicable for different types of testing, the same might hold for SPL, but no experience reports were found to support this statement.

4.5.2.6 Commonality and Variability Testing

Commonality, as an inherent concept in the SPL theory, is naturally addressed by many studies, such as stated by Pohl *et al.* (2005b), in which the major task of domain testing is the development of common test artifacts to be further reused in application testing.

The increasing size and complexity of applications can result in a higher number of variation points and variants, which makes testing all combinations of the functionality almost impossible in practice. Managing variability and testability is a trade-off. The large amount of variability in a product line increases the number of possible testing combinations. Thus, testing techniques that consider variability issues and thus reduce effort are required.

Cohen *et al.* (2006) introduce cumulative variability coverage, which accumulates coverage information through a series of development activities, to be further exploited in a target testing activities for product line instances.

Another solution, proposed by Kolb and Muthig (2006), is the imposition of constraints in the architecture. Instead of having components with large amount of variability it is better for testability to separate commonalities and variabilities and encapsulate variabilities as subcomponents. Aiming to reduce the retest of components and products when modifications are performed, independence of feature and components, as well as the reduction of side effects, reduce the effort required for adequate testing.

Tevanlinna *et al.* (2004) highlight the importance of asset traceability from requirements to implementation. There are some ways to achieve this traceability between test assets and implementation, as reported by McGregor *et al.* (2004b), in which the design of each product line test asset matches the variation implementation mechanism for a component.

The selected approaches handle variability in a range of different manners, usually expliciting variability as early as possible in UML use cases Hartmann *et al.* (2004); Kang

et al. (2007); Rumbaugh *et al.* (2004) that will further be used to design test cases, as described in the requirement-based approaches Bertolino and Gnesi (2003a); Nebut *et al.* (2003). Moreover, model-based approaches introduce variability into test models, created through use cases and their scenarios Reuys *et al.* (2005, 2006), and specifying variability into feature models and activity diagrams Olimpiew and Gomaa (2005a, 2009). They are usually concerned about reusing test case in a systematic manner through variability handling as Al-Dallal and Sorenson (2008); Wübbeke (2008) report.

4.5.2.7 Variant Binding Time

According to McGregor *et al.* (2004b), the binding of different variants requires different binding time (*Compile Time, Link Time, Execution Time and Post-Execution Time*), which requires different mechanisms (e.g. inheritance, parameterization, overloading and conditional compilation). They are suitable for different variability implementation schemes. The different mechanisms result in different types of defects, test strategies, and test processes.

This issue is also addressed by Jaring *et al.* (2008), in their Variability and Testability Interaction Model, which is responsible for modeling the interaction between variability binding and testability in the context of the V-model. The decision regarding the best moment to test a variant is clearly important. The earliest point at which a decision is bound is the point at which the binding should be tested.

In our findings, the approach presented in Reuys *et al.* (2006) deals with testing variant binding time as a form of ensuring that the application comprises the correct set of features, as the customer looks forward. After performing the traditional test phases in application engineering, the approach suggests tests to be performed towards verifying if the application contains the set of functionalities required, and nothing else.

4.5.2.8 Effort Reduction

Some authors consider testing the bottleneck in SPL, since the cost of testing product lines is becoming more costly than testing single systems Kolb (2003); Kolb and Muthig (2006). Although applications in a SPL share common components, they must be tested individually in system testing level. This high cost makes testing an attractive target for improvements Northrop and Clements (2007). Test effort reduction strategies can have significant impact on productivity and profitability McGregor (2001a). We found some strategies regarding this issue. They are described as follows:

- **Reuse of test assets:** Test assets - mainly test cases, test scenarios and test results - McGregor (2001a) are created to be reusable, which consequently impacts the effort reduction. According to Kauppinen and Taina (2003) and Zeng *et al.* (2004), an approach to achieve the reuse of core assets comes from the existence of an asset repository. It usually requires an initial testing effort for its construction, but throughout the process, these assets do not need to be rebuilt, they can be rather used as is. Another strategy considers the creation of test assets as extensively as possible in domain engineering, anticipating also the variabilities by creating documents templates and abstract test cases. Test cases and other concrete assets are used as is and the abstract ones are extended or refined to test the product-specific aspects in application engineering. In Li *et al.* (2007b), a method for monitoring the interfaces of every component during test execution is proposed, observing commonality issues in order to avoid repetitive execution. As mentioned before in section 4.5.2.6, the systematic reuse of test assets, especially test cases, are the focus of many studies, each offering novel and/or extended approaches. The reason for dealing with assets reuse in a systematic manner is that it can enable effort reduction, since redundant work may be avoided when deriving many products from the product line. In this context, the search for an effective approach has been noticed throughout the past recent years, as can be seen in McGregor (2001a, 2002); Nebut *et al.* (2006); Olimpiew and Gomaa (2009); Reuys *et al.* (2006). Hence, it is feasible to infer that there is not a general solution for dealing with systematic reuse in SPL testing yet.
- **Test automation tools:** Automatic testing tools to support testing activities Condron (2004) is a way to achieve effort reduction. Methods have been proposed to automatically generate test cases from single system models expecting to reduce testing effort Hartmann *et al.* (2004); Li *et al.* (2007a); Nebut *et al.* (2003), such as mapping the models of an SPL to functional test cases in order to automatically generate and select functional test cases for an application derived Olimpiew and Gomaa (2005b). Automatic test execution is an activity that should be carefully managed to avoid false failures since unanticipated or unreported changes can occur in the component under test. These changes should be rejected in the corresponding automated tests Condron (2004).

4.5.2.9 Test Measurement

Test measurement is an important activity applied in order to calibrate and adjust approaches. Adequacy of testing can be measured based on the concept of a coverage criterion. Metrics related to test coverage are applied to extract information, and are useful for the whole project. We investigated how test coverage has been applied by existing approaches regarding SPL issues.

According to Tevanlinna *et al.* (2004), there is only one way to completely guarantee that a program is fault-free, to execute it on all possible inputs, which is usually impossible or at least impractical. It is even more difficult if the variations and all their constraints are considered. Test coverage criteria are a way to measure how completely a test suite exercises the capabilities of a piece of software. These measures can be used to define the space of inputs to a program. It is possible to systematically sample this space and test only a portion of the feasible system behavior Cohen *et al.* (2006). The use of covering arrays as a test coverage strategy is addressed in Cohen *et al.* (2006). Kauppinen and Tevanlinna Kauppinen *et al.* (2004) define coverage criteria for estimating the adequacy of testing in a SPL context. They propose two coverage criteria for framework-based product lines: *hook* and *template coverage*, that is, variation points open for customization in a framework are implemented as *hook classes* and stable parts as *template classes*. They are used to measure the coverage of frameworks or other collections of classes in an application by counting the structures or hook method references from them instead of single methods or classes.

4.5.3 Analysis of the Results and Mapping of Studies

The analysis of the results enables us to present the amount of studies that match each category addressed in this study. It makes it possible to identify what have been emphasized in past research and thus to identify gaps and possibilities for future research Petersen *et al.* (2008).

Initially, let us analyze the distribution of studies regarding our analysis point of view. Figures 4.6 and 4.7, that present respectively the frequencies of publications according to the classes of the research facet and according to the research questions addressed by them (represented by Q1 to Q9). Table 4.4 details Figure 4.7 showing which papers answer each research question. It is valid to mention that, in both categories, it was possible to have a study matching more than one topic. Hence, the total amount verified in Figures 4.6 and 4.7 exceeds the final set of primary studies selected for detailed analysis.

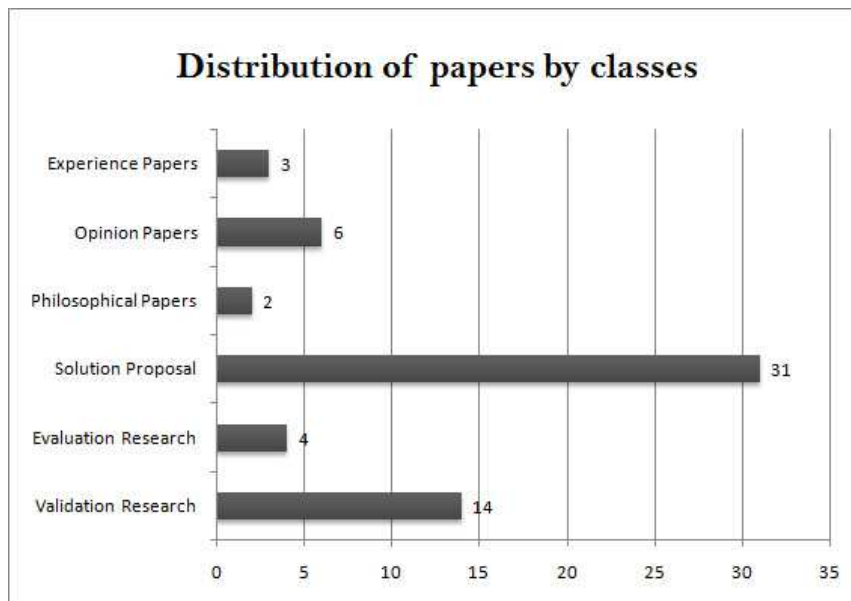


Figure 4.6 Distribution of papers according to classification scheme.

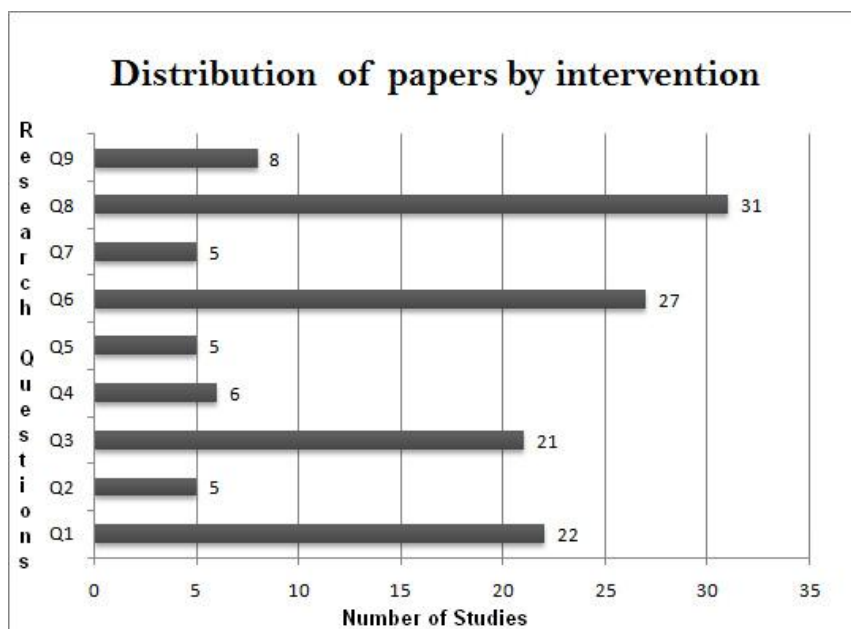


Figure 4.7 Distribution of papers according to intervention.

When merging these two categories, we have a quick overview of the evidence gathered from the analysis of the SPL testing field. We used a bubble plot to represent the interconnected frequencies, as shown in Figure 4.8. This is basically a x-y scatterplot with bubbles in category intersections. The size of a bubble is proportional to the number of articles that are in the pair of categories corresponding to the bubble coordinates Petersen *et al.* (2008).

The classification scheme applied in this paper enabled us to infer that researchers are mostly in the business of proposing new techniques and investigating their properties more than evaluating and/or experiencing them in practice, through proposing new solutions, as seen in Figure 4.8. **Solution Proposal** and **Validation Research** are together, the topics with more entries, if we consider categories considered in this study. Topics such as **Q1** (testing strategies), **Q3** (testing levels), **Q6** (commonality and variability analysis) and **Q8** (effort reduction), join the amount of papers devoted to propose solution for the problems they cover. They have really been the overall focus of researchers. On the other hand we have pointed out topics in which new solutions are required, it is the case of **Q2** (static and dynamic analysis interconnection in SPL Testing), **Q4** (regression testing), **Q5** (non-functional testing), **Q7** (variant binding time) and **Q9** (measures).

Although some topics present a relevant amount of entries in this analysis, such as **Q1**, **Q3**, **Q6** and **Q8**, as aforementioned, these still lack field research, since the techniques

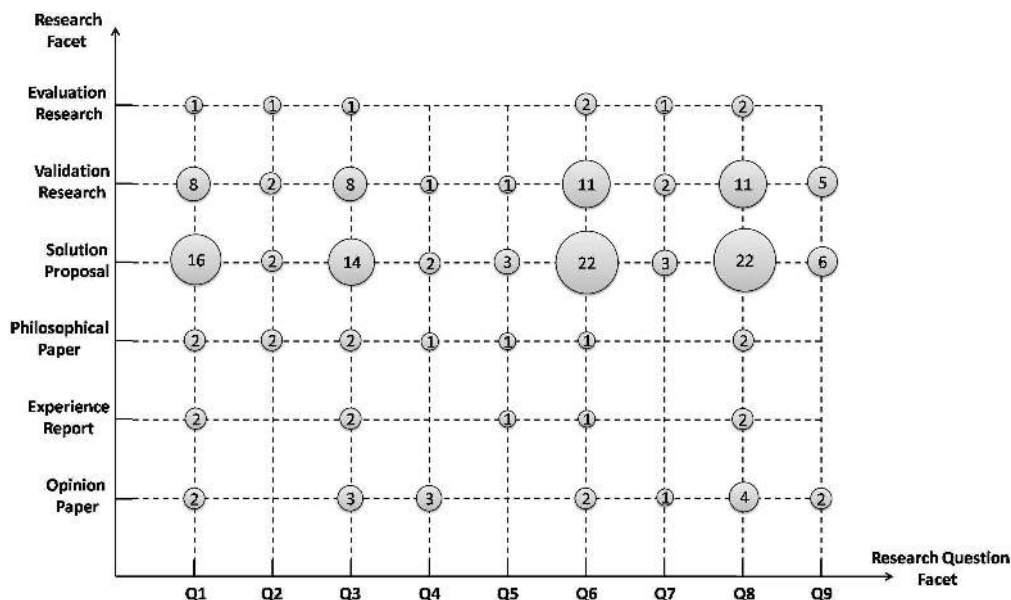


Figure 4.8 Visualization of a Systematic Map in the Form of a Bubble Plot.

Table 4.4 Research Questions (RQ) and primary studies.

RQ	Primary Studies
Q1	Al-Dallal and Sorenson (2008); Bertolino and Gnesi (2003a,b); Odia (2007); Jaring <i>et al.</i> (2008); Jin-hua <i>et al.</i> (2008); Kang <i>et al.</i> (2007); Kauppinen <i>et al.</i> (2004); Kishi and Noda (2006); Kolb (2003); Kolb and Muthig (2003, 2006); McGregor (2001b, 2002); Olimpiew and Gomaa (2005a, 2009); Reis (2006); Reis <i>et al.</i> (2007a); Reuys <i>et al.</i> (2005, 2006); Wübbeke (2008); Zeng <i>et al.</i> (2004)
Q2	Al-Dallal and Sorenson (2008); Denger and Kolb (2006); Odia (2007); Kishi and Noda (2006); McGregor (2001b)
Q3	Al-Dallal and Sorenson (2008); Odia (2007); Geppert <i>et al.</i> (2004); Jaring <i>et al.</i> (2008); Kauppinen (2003); Kamsties <i>et al.</i> (2003); Jin-hua <i>et al.</i> (2008); Kolb and Muthig (2003, 2006); Li <i>et al.</i> (2007a,b); McGregor (2001b, 2002); Muccini and van der Hoek (2003); Olimpiew and Gomaa (2005a); Nebut <i>et al.</i> (2006); Pohl and Sikora (2005); Reis <i>et al.</i> (2007a); Reuys <i>et al.</i> (2006); Wübbeke (2008); Zeng <i>et al.</i> (2004)
Q4	Harrold (1998); Jin-hua <i>et al.</i> (2008); Kauppinen and Taina (2003); Kolb and Muthig (2003); McGregor (2001b); Muccini and van der Hoek (2003)
Q5	Feng <i>et al.</i> (2007); McGregor (2001b, 2002); Nebut <i>et al.</i> (2003); Reis (2006)
Q6	Al-Dallal and Sorenson (2008); Beatriz Pérez Lamanha (2009); Bertolino and Gnesi (2003a,b); Cohen <i>et al.</i> (2006); Condrón (2004); Odia (2007); Feng <i>et al.</i> (2007); Geppert <i>et al.</i> (2004); Jaring <i>et al.</i> (2008); Kamsties <i>et al.</i> (2003); Kang <i>et al.</i> (2007); Kishi and Noda (2006); Kolb and Muthig (2006); Li <i>et al.</i> (2007a,b); McGregor <i>et al.</i> (2004b); Nebut <i>et al.</i> (2006); Olimpiew and Gomaa (2009); Pohl and Metzger (2006); Pohl and Sikora (2005); Reis (2006); Reis <i>et al.</i> (2007a); Reuys <i>et al.</i> (2005, 2006); Wübbeke (2008); Zeng <i>et al.</i> (2004)
Q7	Cohen <i>et al.</i> (2006); Jaring <i>et al.</i> (2008); Jin-hua <i>et al.</i> (2008); McGregor <i>et al.</i> (2004b); Pohl and Metzger (2006)
Q8	Al-Dallal and Sorenson (2008); Bertolino and Gnesi (2003a); Condrón (2004); Odia (2007); Feng <i>et al.</i> (2007); Ganesan <i>et al.</i> (2005); Geppert <i>et al.</i> (2004); Jaring <i>et al.</i> (2008); Kang <i>et al.</i> (2007); Kauppinen (2003); Kauppinen and Taina (2003); Kishi and Noda (2006); Kolb and Muthig (2006); Li <i>et al.</i> (2007a,b); McGregor (2001b); Nebut <i>et al.</i> (2003, 2006); Olimpiew and Gomaa (2009); Pohl and Metzger (2006); Reis <i>et al.</i> (2007a); Reuys <i>et al.</i> (2005, 2006); Zeng <i>et al.</i> (2004)
Q9	Al-Dallal and Sorenson (2008); Ganesan <i>et al.</i> (2005); Jin-hua <i>et al.</i> (2008); Kauppinen (2003); Olimpiew and Gomaa (2009); Reuys <i>et al.</i> (2006)

investigated and proposed are mostly novel and have usually not yet been implemented in practice. We realize that currently, **Evaluation Research** is weak in SPL Testing papers. Regarding the maturity of the field in terms of evaluation research and solution papers, other studies report results in line with our findings, e.g. Šmite *et al.* (2010). Hence, we realize that this is not a problem solely to SPL testing, but rather it involves, in a certain way, other software engineering practices.

We also realize that researchers are not concerned about **Experience Reports** on their personal experience using particular approaches. Practitioners in the field should report results on the adoption, in the *real world* of the techniques proposed and reported in the literature. Moreover, authors should **Express Opinions** about the desirable direction of SPL Testing research, expressing their experts viewpoint.

In fact, the volume of literature devoted to testing software product lines attests to the importance assigned to it by the product line community. In the following subsection we detail what we considered most relevant in our analysis.

4.5.3.1 Main findings of the study

We identified a number of **test strategies** that have been applied to software product lines. Many of these strategies address different aspects of the testing process and can be applied simultaneously. However, we have no evidence about the effectiveness of combining strategies, and in which context it could be suitable. The analyzed studies do not cover this potential. There is only a brief indication that the decision about which kind of strategy to adopt depends on a set of factors such as software development process model, languages used, company and team size, delivery time, budget, etc. Moreover, it is a decision made in the planning stage of the product line organization since the strategy affects activities that begin during requirements definition. But it still remains as hypotheses, that need to be supported or refuted through formal experiments and/or case studies.

A complete testing process should define both **static and dynamic analyses**. We found that even though some studies emphasize the importance of static analysis, few detail how this is performed in a SPL context Kishi and Noda (2006); McGregor (2001b); Tevanlinna *et al.* (2004), despite its relevance in single-system development. Static analysis is particularly important in a product line process since many of the most useful assets are non-code assets and particularly the quality of the software architecture is critical to success.

Specific **testing activities** are divided across the two types of activities: **domain engineering and application engineering**. Alternatively, the testing activities can be grouped into **core asset and product development**. From the set of studies, around four Jaring *et al.* (2008); Jin-hua *et al.* (2008); Kauppinen (2003); Odia (2007) adopt (or advocate the use of) the V-model as an approach to represent testing throughout the software development life cycle. As a widely adopted strategy in single-system development, tailoring V-model to SPL could result in improved quality. However, there is no consensus on the correct set of testing levels for each SPL phase.

We did not find evidence regarding the impact for the SPL of not performing a specific **testing level** in domain or application engineering, is there any consequence if, for example unit/integration/system testing was not performed in domain engineering? We need investigations to verify such an aspect. Moreover, what are the needed adaptations for the V-model to be effective in the SPL context? This is a point which experimentation is welcome, in order to understand the behavior of testing levels in SPL.

A number of the studies addressed, or assumed, that testing activities are automated (e.g. Condrón (2004); Li *et al.* (2007a)). In a software product line automation is more

feasible because the resources required to automate are amortized over the larger number of products. The resources are also more narrowly focused due to the overlap of the products. Some of the studies illustrated that the use of domain specific languages, and the tooling for those languages, is more feasible in a software product line context. Nevertheless, we need to understand if the techniques are indeed effective when applying them in an industrial context. We lack studies reporting results of this nature.

According to Kolb (2003), one of the major problems in testing product lines is the large number of variations. The study reinforces the importance of handling variability testing during all software life cycle.

In particular, the effect of **variant binding time** concerns was considered in this study. A well-defined approach was found in Jaring *et al.* (2008), with information provided by case studies conducted in an important electronic manufacturer. However, there are still many issues to be considered regarding variation and testing, such as what is the impact of designing variations in test assets regarding effort reduction? What are the most suitable strategy to handle variability within test assets: use cases and test cases or maybe sequence or class diagrams? How to handle traceability and what is the impact of not handling such an issue, in respect to test assets. We also did not find information about the impact of different binding times for testing in SPL, e.g. compile-time, scoping-time, etc. We also lack evidences on this direction.

Regression testing does not belong to any point in the software development life cycle and as a result there is a lack of clarity in how regression testing should be handled. Despite this, it is clear that regression testing is important in the SPL context. Regression testing techniques include approaches to selecting the smallest test suite that will still find the most likely defects and techniques that make automation of test execution efficient.

From the amount of studies analyzed, a few addressed **testing non-functional requirements** Feng *et al.* (2007); McGregor (2001b, 2002); Nebut *et al.* (2003); Reis (2006). They point out that during architecture design static analysis can be used to give an early indication of problems with non-functional requirements. One important point that should be considered when testing quality attributes is the presence of trade-offs among them, for example, the trade-off between modularity and testability. This leads to natural pairings of quality attributes and their associated tests. When a variation point represents a variation in a quality attribute, the static analysis should be sufficiently complete to investigate different outcomes. Investigations towards making explicit which techniques currently applied for single-system development can be adopted in SPL are needed, since studies do not address such an issue.

Our mapping study has illustrated a number of areas in which additional investigation would be useful, specially regarding evaluation and validation research. In general, SPL testing lack evidence, in many aspects. Regression test selection techniques, test automation and architecture-based regression testing are points for future research as well as techniques that address the relationships between variability and testing and techniques to handle traceability among test and development artifacts.

4.6 Threats to Validity

There are some threats to the validity of our study. They are described and detailed as follows:

- **Research Questions:** The set of questions we defined might not have covered the whole SPL testing area, which implies that one may not find answers to the questions that concern them. As we considered this as a feasible threat, we had several discussion meetings with project members and experts in the area in order to calibrate the questions. This way, even if we had not selected the most optimum set of questions, we attempted to deeply address the most asked and considered *open issues* in the field.
- **Publication Bias:** We cannot guarantee that all relevant primary studies were selected. It is possible that some relevant studies were not chosen throughout the searching process. We mitigated this threat to the extent possible by following references in the primary studies.
- **Quality Evaluation:** The quality attributes as well as the weight used to quantify each of them might not properly represent the attributes importance. In order to mitigate this threat, the quality attributes were grouped in subsets to facilitate their further classification. It happens when a study receive a good pontuation regarding to some specific criteria, but when comparing it with papers which handle a broad context it could be wrongly treated as irrelevant.
- **Unfamiliarity with other fields:** The terms used in the search strings can have many synonyms, it is possible that we overlooked some work.

4.7 Related Work

As mentioned before, the literature on SPL Testing provides a large number of studies, regarding both general and specific issues, as will be discussed later on in this study. Amongst them, we have identified some studies developed in order to gather and evaluate the available evidence in the area. They are thus considered as having similar ideas to our mapping study and are next described.

A survey on SPL Testing was performed by Tevanlinna *et al.* (2004). They studied approaches to product line testing methodology and processes that have been developed for or that can be applied to SPL, laying emphasis on regression testing. The study also evaluates the state-of-the-art in SPL testing, up to the date of the paper, 2004, and highlighted problems to be addressed.

A thesis on SPL Testing published in 2007 by Odia (2007), investigated testing in SPL and possible improvements in testing steps, tools selections and application applied in SPL testing. It was conducted using the systematic review approach.

A systematic review was performed by Lamancha *et al.* (2009) and published in 2009. Its main goal was to identify experience reports and initiatives carried out in Software Engineering related to testing in software product lines. In order to accomplish that, the authors classified the primary studies in seven categories, including: Unit testing, Integration testing, functional testing, SPL Architecture, Embedded system, testing process and testing effort in SPL. After that a summary of each area was presented.

These studies can be considered good sources of information on this subject. In order to develop our work, we considered every mentioned study, since they bring relevant information. However, we have noticed that important aspects were not covered by them in an extent that should be possible to map out the current status of research and practice of the area. Thus, we categorized a set of important research areas under SPL testing, focusing on aspects addressed by the studies mentioned before as well as the areas they did not address, but are directly related to SPL practices, in order to perform critical analysis and appraisal. In order to accomplish our goals in this work, we followed the guidelines for mapping studies development presented in Budgen *et al.* (2008). We also included threats mitigation strategies in order to have the most reliable results.

We believe our study states current and relevant information on research topics that can complement others previously published. By current, we mean that, as the number of studies published has increased rapidly, as shown in Figure 4.4, it justifies the need of more up to date empirical research in this area to contribute to the community

investigations.

4.8 Concluding Remarks

The main motivation for this work was to investigate the state-of-the-art in SPL testing, through systematically mapping the literature in order to determine what issues have been studied, as well as by what means, and provide a guide to aid researchers in planning future research. This research was conducted through a Mapping Study, a useful technique for identifying the areas where there is sufficient information for a SR to be effective, as well as those areas where more research is needed Budgen *et al.* (2008).

The number of approaches that handle specific points in a testing process make the analysis and the comparison a hard task. Nevertheless, through this study we are able to identify which activities are handled by the existing approaches as well as understanding how the researchers are developing work in SPL testing. Some research points were identified throughout this research and these can be considered an important input into planning further research.

Searching the literature, some important aspects are not reported, and when they are found just a brief overview is given. Regarding industrial experiences, it is noticed they are rare in literature. The existent case studies report *small projects*, containing results obtained from in company-specific application, which makes impracticable their reproduction in other context, due to the lack of details. This scenario depicts the need of experimenting SPL Testing approaches not in academia but rather in industry. This study identified the growing interest in a well-defined SPL Testing process, including tool support. Our findings in this sense are in line with a previous study conducted by Lamancha *et al.* (2009), which reports on a systematic review on SPL testing, as mentioned in Section 4.7.

This mapping study also points out some topics that need additional investigation, such as quality attribute testing considering variations in quality levels among products, how to maintain the traceability between development and test artifacts, and the management of variability through the whole development life cycle.

4.9 Chapter Summary

Testing is an important mechanism both to identify defects and assure that completed products work as specified. This chapter had the following goals: investigate state-of-

the-art testing practices, synthesize available evidence, and identify gaps between needed techniques and existing approaches, available in the literature. Section 4.5.3.1 presented the main findings of this mapping study that served as base to define the dissertation proposal.

The next Chapter presents the Integration Testing approach proposed by this work.

“Innovation distinguishes between a leader and a follower.”

Steve Jobs

5

A SPL Integration Testing Approach

The Software Product Lines approach involves two development processes, core asset and product development. The prior intends to develop assets to be further instantiated in the latter. From a testing perspective, such a division demands for testing issues to be considered in both processes. Although existing literature presents some information on integration testing for SPL, they usually discuss concerns about test assets generation, despite other several important issues that a process should assemble, such as guidelines, activities, steps, inputs and outputs, roles, and division of responsibilities regarding the both SPL processes. In summary, the existing approaches do not present systematic solutions, which can represent an extra effort to apply a process in the real context. In this chapter, we present a first step in this scenario. In the context of the RiPLE process, a major effort to establish an integrated framework for developing SPLs, an approach is proposed for dealing with integration testing in both core asset and product development. In order to analyze and refine it, an example is discussed in the conference management domain, in which we explain every step of the approach.

The chapter is organized as follows. The next Section outlines some related work. Section 5.3 provides an overview on unit and integration testing, i.e., levels related to this chapter. Section 5.4 describes the main roles and its attributions, as well as the concepts of the Eclipse Process Framework (EPF). In Section 5.5, the main strategies on integration testing are discussed, serving as basis for Section 5.6 in which our approach is detailed. Section 5.7 shows an example of applying the proposed approach. Finally, Section 5.8 presents the chapter summary.

5.1 Introduction

SPL is an efficient approach that aids organizations to develop quality products from reusable core assets rather than from scratch (Kim *et al.*, 2006). This approach is supported by two major processes: *core asset development* (CAD), consisting of analyzing the domain of a product line, develop the architecture and producing the reusable assets; and *product development* (PD), in which products are derived based on the assets prior developed. In this latter, assets are reused rather than producing everything from scratch (Clements and Northrop, 2001; Linden *et al.*, 2007).

The both processes require related but distinct treatments. Such treatment does not stop at development but should extend also to testing activities (Kang *et al.*, 2007). Thus, it is necessary to establish the relationship among testing levels and the SPL processes, thus, it could be feasible for an organization to establish a strategy regarding applying a suitable approach for testing in this context.

As discussed in the literature (Beizer, 1990; Rothermel and Harrold, 1996; McGregor, 2001b; Tevanlinna *et al.*, 2004), there are three main levels of testing: **unit testing**, **integration testing**, and **system testing**. In unit testing, each developer tests a unit before integrating it to the rest of the code. This is usually performed using white-box techniques (i.e. with access to the code), however, black-box techniques can also be applied. In integration testing, aims to test the interaction among the components and make sure that they follow the interface specifications and work properly. Integration testing can involve a combination of white-box, exercising the code and black-box testing, usually performed in the interface components, selecting valid and invalid inputs in order to determine the correct output. System testing tests the features and functions of an entire product and validates that the system works in the way the user expects. Black-box techniques are usually adopted. While for unit and integration testing we need the source code, system testing can be done independently from source code (Geppert *et al.*, 2004).

Once the units were tested during the unit test level, they need to be integrated to compose the SPL reference architecture (CAD) and product specific architectures (PD). This union is tested during integration testing level, which aims to detect defects that occur on the interfaces of units and assemble the individual units into working modules, subsystems and, finally, complete the architecture. In order to address this issue, a systematic approach for testing in SPL was designed, considering both processes core asset and product development.

Hence, this is the main focus of this chapter, to present an approach to provide a

systematic way to use an SPL architecture for code integration testing, considering the both SPL processes. The Data flow, activities, roles and guidelines are prescribed in order to give users useful directions towards the use the approach. The use of the approach is illustrated with an example applied with excerpts of a SPL project. It was developed in the RiPLE - RiSE process for Product Line Engineering - context, a larger effort in defining a complete process for SPL, encompassing issues ranging from scoping to evolution management.

5.2 Integration Testing in SPL

Beizer (1990) lists three major divisions regarding dynamic testing: *unit*, *integration* and *system testing*. This division is also adopted by McGregor in (McGregor, 2001b), where he defines each test level. Regarding integration testing, he advocates that the focus should be on testing the interactions that occur among tested units. It is a cumulative effort and also a shared responsibility between CAD and PD builders. The integration testing continues iteratively until the integrated units compose the desired product. Due to the number of variants for each variation point, it makes impossible to test all combinations of all variants. McGregor proposes two techniques to mitigate this problem: combinatorial test design and incremental integration tests.

In (Knauber and Hetrick, 2005), Knauber et.al. advocate a similar division: testing at component level, feature level and product level. In this approach, the features are considered integration units, as each product has a set of features it implies that the tests at this level are customized according to the decision model. Any defect discovered in this level should be fed back into component tests level in order to identify whether the problem is in the generated test case or in the component.

Reuys *et al.* (2006) define a method for system and integration testing for SPL. In this work, interactions among components are considered in addition to the interactions between users and system. The interactions are described in domain architecture scenarios that contain component interactions derived. It will lead to build domain integration test cases. However, they rather do not address the effect of different forms of integration strategies nor the additional variability that is contained in domain architecture models.

Li *et al.* (2007a) propose a method for generating integration test cases of product lines members from module unit tests. Each product line member has a set of integration tests, each of which describes interactions among modules and functions that need to be tested. The number of product line members configurations are decided by all combination

of the variability parameters. Moreover, the constraints among variabilities should be considered. To help the identification of valid products the decision model is used in this approach.

Reis *et al.* (2007b) define an automated integration test technique that can be applied during domain engineering. It generates integration test case scenarios that cover all interactions between the integrated components, which are described in the test model.

In summary, besides none of the analyzed approaches deal with both SPL integration testing levels, during *Core Asset Development* and *Product Development*, they do not define in a systematic and structured way, a process view, where a set of roles, activities and steps, are defined to perform integration tests. Moreover, the proposed approach can be applied for testing the reference architecture conformance (specification vs. code) and product specific architectures, as well as, testing the integration of product specific components to the corresponding product architecture.

5.3 Unit and Integration Testing

This section aims to describe the context in which the integration testing approach was inserted, as well as, its relation with the unit testing level. The approaches (integration and regression described in the next chapter) proposed by this dissertation, is part of a more general process called RiSE Product Line Engineering (RiPLE), which concerns with the full software life-cycle for software product lines. A more detailed view of these approaches can be viewed in RiPLE WebSite ¹. In this site, the artifacts, activities and steps are described.

The first step when dynamically testing a software is to define which software portion will represent a unit (e.g. methods or procedures (McGregor, 2001b), classes, components (Markus Gälli and Nierstrasz, 2005), etc). Such a decision will serve as a background for applying a specific unit testing strategy. For example, if a component is defined as a unit, the strategy should define that methods and classes are tested individually and then their interaction thus, this modular and cohesive element has been tested according to the purpose of *unit testing*. Figure 5.1 describes the RiPLE unit test level main flow.

In *Unit Testing*, the main goal is to ensure that this portion is working properly. A product comprises several units, which, at this point, it should be tested individually. This activity enables to find, and then correct, errors at a fine grained level, which can reduce the error propagation. After verifying individually the units, they need to be

¹<http://www.cin.ufpe.br/sople/testing/epf/>

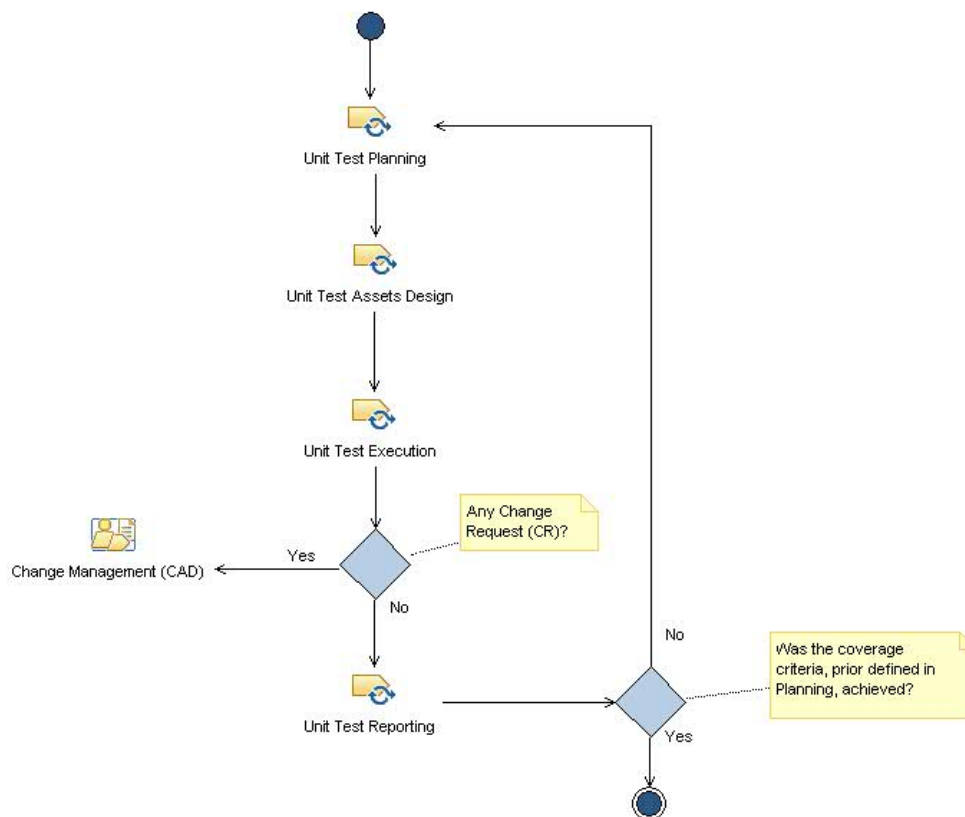


Figure 5.1 RiPLE Unit Testing level main flow.

integrated to compose modules, sub-systems, the SPL reference architecture and further instantiate it to assemble specific product architectures. The test in which units are joined is known as *Integration Test*. In summary, this level aims to evaluate the connection among components and modules by stressing their interfaces.

The use of both testing phases is advocated since they are responsible for detecting different types of faults. Whereas *unit testing* independently tests methods, classes, and the interaction among these pieces which comprises a component, *integration testing* is responsible for testing the interaction among components interfaces and the integration between modules.

If a fault is detected inside a component, during integration testing, it should be analyzed and then forward back to the unit test level, to be re-evaluated. All information is recorded in the associated component test report and the test plan is updated, thus, next turn, the test suite will provide a test in this direction. The same is applicable to module

and subsystems.

Information from unit tests are further used during integration and system test levels, e.g., coverage criteria and pass/fail rate. Whereas in integration test level, this rate can be used to define/build the module and subsystem test planning. In system test level, it can be used to define the test planning regarding the derived application.

5.4 Roles and Attributions

The approaches (unit and integration testing) define a set of roles. It is worthwhile to mention these do not represent the function an engineer should assume in an organization, but rather the role she/he is likely to have in the context of a specific project.

- **Test Manager** - Responsible for preparing the test plan, negotiating the test objectives and products, analyzing test effort, test resources management and test the environment management, keeping track test activities, setting an acceptance criteria based on the project budget and helping the project manager to keep the software testability during the development.
- **Test Architect** - Responsible for identifying the test target, defining features, components and modules to be tested, as well as the test execution management. Usually, a product and a set of features are assigned to a test architect. This assignment aims to designate a person to solve eventual problems regarding to a feature or product under testing. They are responsible for creating and managing and after scheduling their execution. After this execution, they should assemble a test report to be attached with the SPL asset and serve as input to further test manager planning.
- **Test Designer** - Responsible for functional test design (considering white-box and black-box techniques), test maintenance and test case validation. Depending on the features size and the amount of features, some companies designate an entire team to perform these activities.
- **Tester** - Responsible for test design according to the required technique, test execution, change request reporting, test harness development and test environment setup. The tester should provide information to assembling test reports.

5.4.1 Method Content and Processes

Both approaches (Chapters 5 and 6) proposed in this dissertation are modelled inside the Eclipse Process Framework (EPF)², which aims at providing an extensible framework and exemplary tools for software process engineering - method and process authoring, library management, configuring and publishing a process.

EPF uses the Software Process Engineering Meta-Model (SPEM), that defines a formal language for describing development processes. EPF is based on SPEM 2.0, released on April, 2008 (SPEM, 2008).

Since, one of the main goals of EPF is to provide the reuse among sets of reusable activities (called Method contents), the EPF structure is divided into two main categories.

- **Method Content:** A set of defined tasks flow, roles, artifacts and guides to accomplish some goal.
- **Processes:** Process flows which consumes the method contents, reusing the previously defined activities.

Both method contents, and processes are divided into some concepts, according to the SPEM (SPEM, 2008).

The Method Content contains the following concepts:

- **Role:** Roles define a set of related skills, competencies and responsibilities. Roles perform tasks.
- **Work Product:** Work Products (in most cases) represent the tangible things used, modified or produced by a Task.
- **Tasks:** A Task defines an assignable unit of work (usually a few hours to a few days in length).
- **Guidance:** Guidance may be associate with Roles, Tasks, and Work Products, and may have the form of a checklist, an example, a template and etc.

The Processes contains the following concepts:

- **Capability Pattern:** Capability Patterns define the sequence of related Tasks, performed to achieve a greater purpose.

²Eclipse Process Framework web site - <http://www.eclipse.org/epf/>

- **Delivery Process:** Defined using Work Breakdown Structures and/or Activity Diagrams. Defines end-end full-lifecycle process and may include iterations, phases, milestones.

The main benefits from using EPF are:

- **Reuse:** The method contents can be reused throughout the processes.
- **Web Site generation:** EPF generates automatically a web site containing all information of the modeled process, making the publication of the process a very easy task.

5.5 Integration Testing Strategies

Once the units were reviewed and successfully passed through the unit tests, according to the coverage criteria previously defined (during planning activity in unit testing level), they need to be integrated and hence tested.

The integration of a system can be tested incrementally or using a big-bang approach (Muccini and van der Hoek, 2003). In **nonincremental (or “big-bang”) approach**, all units (methods, classes or components) are independently tested and they are then combined (Burnstein, 2003; Myers, 2004). After that, the entire program is tested as a whole. A disadvantage of this approach is the difficulty to find defects. Since all components are integrated together at the same time, it is hard to find which integration causes a fault. In **incremental approach**, a unit is integrated into a set of previously integrated modules (set of units) which were prior approved (Burnstein, 2003), which makes easy to identify the defective integration. The incremental approach can be performed in two ways, using **top-down** or **bottom-up** strategies.

In **top-down**, the components and modules are integrated downwards through the control hierarchy, beginning with the main control unit, e.g., component or module. It is important to highlight the need of drivers and stubs when using this approach. The components can be incorporated to the main control unit in either a *depth-first* or *breadth-first* manner. Figure 5.2, in which structure charts, or even call graphs as they are otherwise known, are presented to sample the integration strategies. These charts show hierarchical calling relationships among units. Each node, or rectangle in a structure chart, represents a unit, and the edges or lines between them represent calls between the units (Burnstein, 2003).

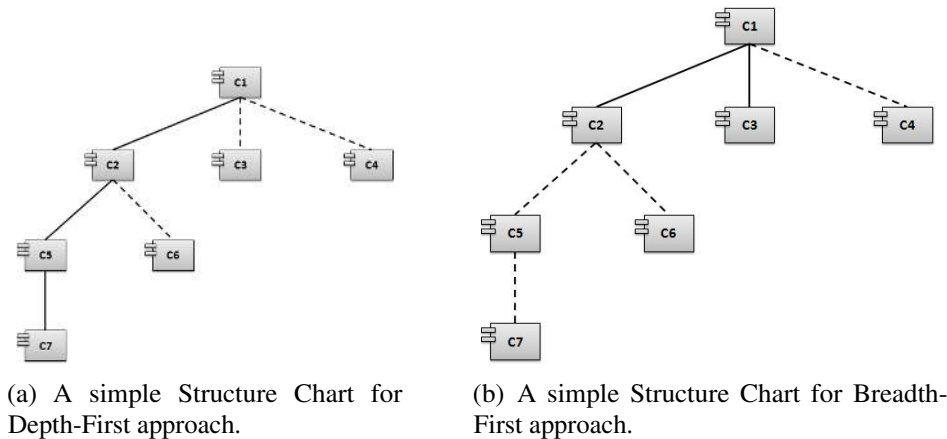


Figure 5.2 Two top-down manners to integrate components and modules.

In the simple chart in Figure 5.2, the rectangles C1-C7 represent all the system units, the dashed lines represent the components that are not linked yet, on the other hand, solid lines represent the components that were linked. Edges, from an upper-level unit to one below indicate that the upper level units calls the lower one.

Using the *depth-first* manner, the C2 component is integrated with the main component C1, right after C5 and C7 integration to the structure (see Figure 5.2(a)). When integrating C2 to C1, it is clear the need of stubs to represent components C3 and C4.

In the *breadth-first* manner, the C2 and after C3 are integrated to the C1 main component. Every component directly subordinated to each level is incorporated, moving across the structure horizontally (Burnstein, 2003) (see Figure 5.2(b)). Since not all components are ready, stubs are thus needed. As the components are incorporated to the main structure, the stubs are replaced by real components.

Top-down integration ensures that the upper-level modules are tested early in integration (Burnstein, 2003). Stubs development is indeed required, in order to drive significant data upward. As a consequence, it can allow system to be demonstrated earlier, since every behavior of software can be present. However, this strategy is relevant if major flaws occur toward the top of the program (Myers, 2004). If they are complex and need to be redesigned there will be more time to do so.

In **bottom-up** integration, the lowest level units are firstly combined. These do not call other units. Drivers should be developed to coordinate test case input and output. Next, units are integrated on the next upper level of the structure chart, whose subordinate units have already been tested. After a unit has been tested, drivers are removed and the actual components are combined moving upward in the structure (Myers, 2004).

According to (Burnstein, 2003), the advantage of bottom-up integration is that the lower-level units are usually well tested early in the integration process, an important strategy if units are supposed to be reused. On the other hand, since the upperlevel units are tested later in the integration process, they may not be tested, due to time constraints or any other reason. Moreover, with this strategy postpones, the system does not fully exist until the last unit is integrated.

Each strategy has its set of advantages and disadvantages, which make difficult to choose the best one. Thus, in many cases, a combination of approaches should be used.

5.6 Integration Testing approach for SPL

In this section, the proposed approach for integration testing in SPL is described. The two processes, *Core Asset Development (CAD)* and *Product Development (PD)*, are considered in this approach, in a way which two different but also complementary standpoints dealing with such a level are performed. In the former, where assets are prepared to establish a common architecture, the focus is on test the integration among the modules and components that will compose the architecture. In the latter, where components are integrated in order to realize products, tests are focused on integration between product specific parts and the reference architecture. This architecture is intended to support the diverse products in a product line, considering the decisions and principles for each SPL member (Kolb and Muthig, 2006).

To better isolate the objectives, this integration testing level will be separately performed during the two processes, where each one will be following detailed. Figure 5.3 shows a resumed flow used for both integration in CAD and PD. Although they are presented as sequentially initiated, this process represents an incremental and iterative development step, since feedback connections enable refinements along the approach. This flow illustrates the approach workflow comprising its activities, inputs, outputs, tasks and involved roles. Regarding the latter, in this process, responsibilities are assigned for these activities to four stakeholders/roles that we believe represent the key participants in the SPL testing process: test managers, test architects, test designers and testers/developers.

5.6.1 Integration Testing in Core Asset Development (CAD)

Integration Testing in Core Asset Development aims to test the interaction on the SPL common components and the reference architecture as well. The integration testing main

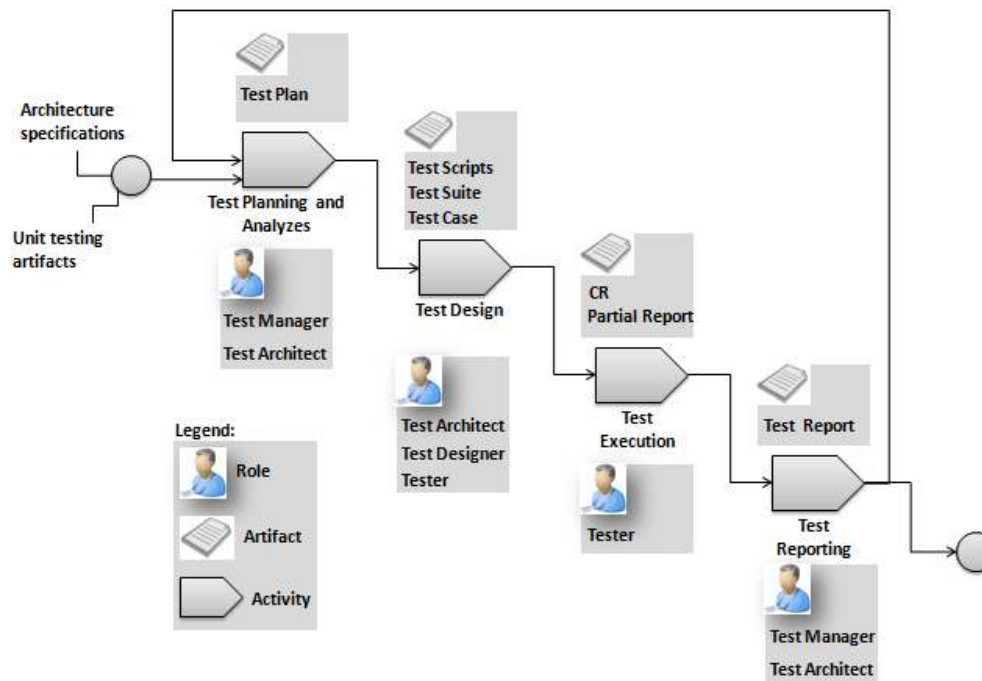


Figure 5.3 An overview on the RiPLE-TE Integration approach work flow.

flow to CAD, can be viewed in Figure 5.4. As input of this testing level, we should consider:

- unit tested components;
- feature dependency diagram;
- feature model;
- architectural views (behavioral and structural);
- use cases; and
- requirements.

The *unit tests* in the components should be previously performed in order to find errors and correct them at a more fine-grained level, which can aid in avoiding error propagation. The *feature dependency diagram* (see example in Figure 5.7) provides information about the operational dependencies among features - e.g., Usage, Modification, Exclusive-Activation, Subordinate-Activation, Concurrent-Activation and Sequential-Activation dependency (Lee and Kang, 2004). This information will be useful when designing test

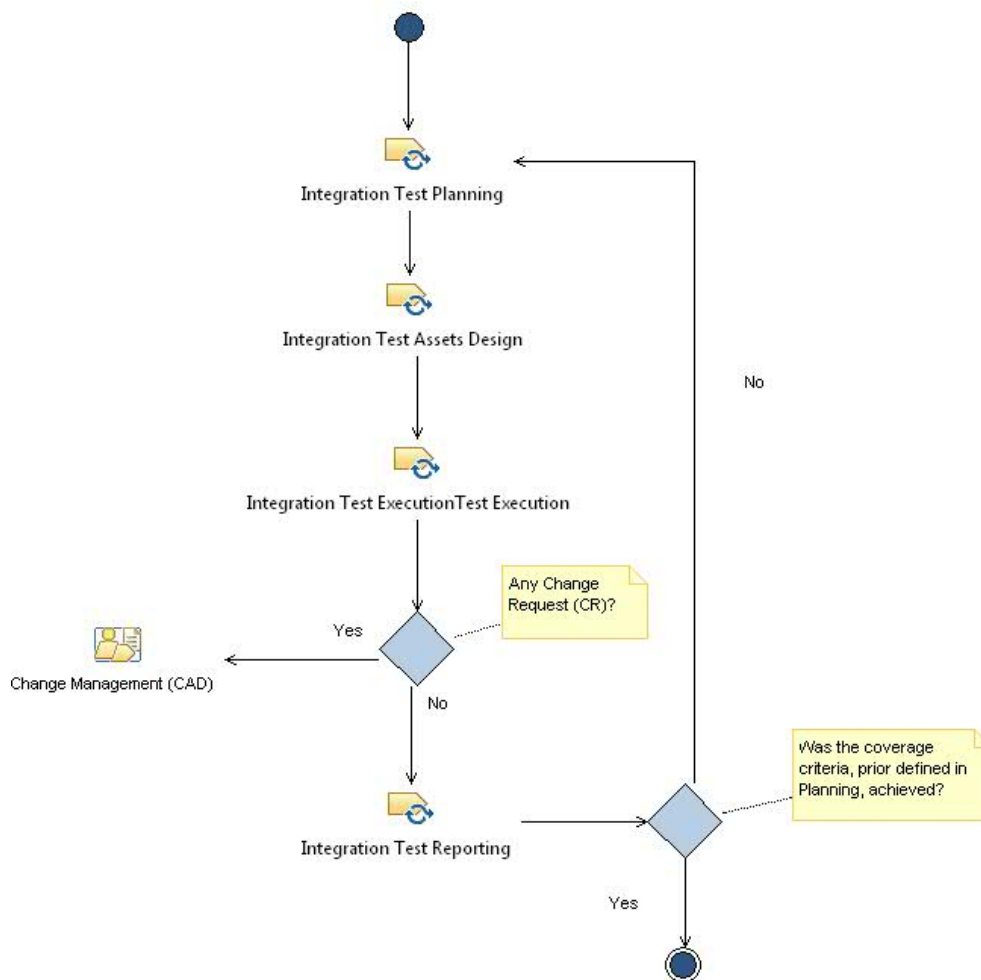


Figure 5.4 RiPLE Integration Testing level (CAD) main flow.

cases, since from the relationship among features, the test cases are designed reflecting this interactions. For example, in a scenario where one feature excludes the presence of another one, the test cases should be prepared to handle this dependency.

The test cases are designed by using the information available on feature dependency diagram. To achieve it, the test designer make use of the use cases provided by the Riple-RE Neiva (2008) which considers these dependencies among features. This way, the same information used to build use cases are also applied to design test cases.

In the *feature model* some dependencies and links can be extracted in order to support the test design activity, for example, some features have strong dependency relation, where

both of them should be used at the same time, using this information the test designers are capable to identify and create test cases which exercises the integration between this two features. These information are also important during test suite composition, i.e., in a scenario where each feature has its respective test suite, the dependency information is important in order to help the test selection process. All these information are provided by RiPLE-SC (Scoping) and RiPLE-RE (Requirement).

Regarding *architectural views*, we suggest, based on Muccini et al.'s proposal (Muccini *et al.*, 2006), the adoption of the *behavioral view* represented by sequence diagrams and the *structural view*, which comprises class and module diagrams. Whereas behavioral view provides information about the architecture functionalities, the structural view provides information about its topological structure. The information provided by the architectural views is useful in order to test the conformance of a software architecture against its specification.

In cases which we do not have detailed information on the architecture or even if it does not represent the actual system architecture, due to lacks in update or any other reason, we should access the component code and search for pieces of code where the relationship among the components occur, such as interfaces, in case of object-oriented development, and benefit from this information to generate test cases.

This phase has as output the tested reference architecture, the test plan and report artifacts (Described in ³) regarding to each intermediary module, as well as, the tested modules.

When performing Integration Testing in CAD, besides suitably handling inputs and outputs, variability concerns should be also considered, since it directly influences the way components interact with each other. Just to illustrate this point, these interactions can occur in different ways, such as: *(i)* the variability may occur inside the component; *(ii)* in a way where the components interact; or *(iii)* the component is realized a variant (Jin-hua *et al.*, 2008).

In Figure 5.5, all interactions can be viewed. Where *VP* is variation point and *V1*, *V2* and *V3* are variants from a variation point.

Variability causes a combinatorial explosion implying that it is no possible to thoroughly test the integration among all the components. However, as Li et. al. pointed out (Jin-hua *et al.*, 2008), not all components interactions are realized in CAD. Such a decision may be postponed to the PD process, as we will see in next subsection.

³www.cin.ufpe.br/sople/testing/epf/

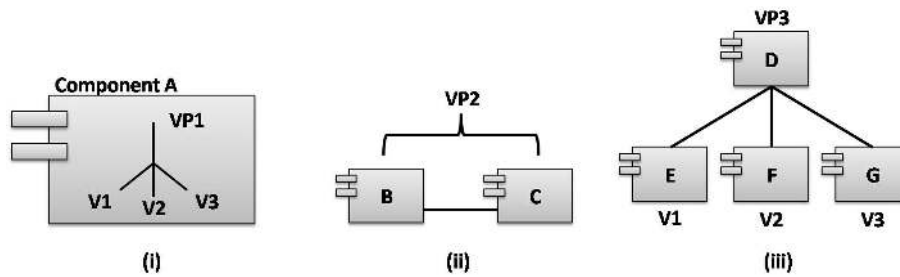


Figure 5.5 Variability influence in components interactions.

In order to reduce this testing effort, a prioritization and test criteria should be established. The idea is to prioritize the components from the reference architecture and the ones which implement the most common or complex variation points. According to (Jin-hua *et al.*, 2008), the integration testing perform those tests which test common interactions and those that contain few variable interactions. The test criteria can be useful when the test architect, is analyzing the structural view and the dependency diagram, and he can capture some critical components and interactions. By looking the behavioral view, he can capture software functionalities that he is interested to test. In (McGregor, 2001b), he highlights two distinct techniques: (i) a combinatorial test design and (ii) perform the integration testing using a incremental strategy, to mitigate the variant combinatorial problem. The use of both techniques are recommended in this approach.

The product line architecture have numerous points at which it can varied in order to produce different products. It begins as requirements until architectural point of variation. This variation are propagated down to the method level where the number and types of parameters are varied. It is virtually impossible to test all combinations of theses variants. At these various points, choices must be made in order to select which values to use during a specific test. The combinatorial test design support the design of test cases by identifying various levels of combinations of input values for the asset under test. The number of variation points and the number of different variants at each point make an exhaustive test set much too large to be practical. Combinatorial design allows the selection of a less-than exhaustive test set. By selecting all pair-wise combinations, as opposed to all possible combinations, the number of test cases is dramatically reduced McGregor (2001b).

The incremental strategy is implemented as the products are tested. Firstly, as the product specific components are integrated to the reference architecture, the integration tests are performed incrementally after each integration McGregor (2001b). Another

strategy used to reduce this effort is the use of incremental strategy over products. In this approach the first product is tested individually and the following products are tested using regression testing techniques Tevanlinna *et al.* (2004).

Following, the steps needed to produce assets for integration testing in CAD are described, based on the explanation aforementioned.

- **Architecture Specification Analysis:** The test architect analyzes the structural view (class diagram, component and module diagrams) to capture structural issues in the components, modules and architecture. Based on this view, he can observe if the implementation of a module or architecture is in conformance with its specification, by looking the way in which the components and modules interacts. The same is applied for the behavioral view (sequence and communication diagrams), where he attempts to capture issues regarding functionalities. By analyzing this view, the test architect is able to understand how the components and modules work, looking the sequence diagram he has a more accurate view of the methods and classes, as well as, how they interact.
- **Test Criteria:** Due to the amount of classes and components, as well as, the amount of links among them, which could likely be excessive, the architect can face visualization problems; in order to mitigate it, he should select critical points to test. Each criteria highlights a specific perspective of interest for a test session.
- **Test Design:** In this step, the architectural test cases are created, based on previous information. By looking the sequence diagrams, the test cases are extracted. They are composed by two portions, an input which works as a stimulus and a sequence of events which represents a path through the architecture. By observing the sequence diagram, the test case steps and constraints (if exists) are identified, to compose the test case as a whole. White-box techniques can be used in this step.

After selecting the more suitable elements, considering the constraints involved since planning phase, and thus producing the useful assets (i.e., test suites), the next step is to execute them. In this step, the test cases execute the paths previously designed. After executing the tests and, supported by an automatic coverage tool (e.g. (Ecl, 2009; Clo, 2009; EMM, 2009; Cov, 2009)), the test engineer can observe the test case effectiveness and decide if it pass/fail according to the architecture specifications. He can also confirm if the test covers the desired portion of code.

5.6.2 Integration Testing in Product Development (PD)

The inputs for Integration Testing in CAD can also be used in PD, with the addition of *product map* and/or *decision model*, since these assets hold information on the features (specific functionalities) of each product that will be realized in the product line. The general view of this level is showed in Figure 5.6.

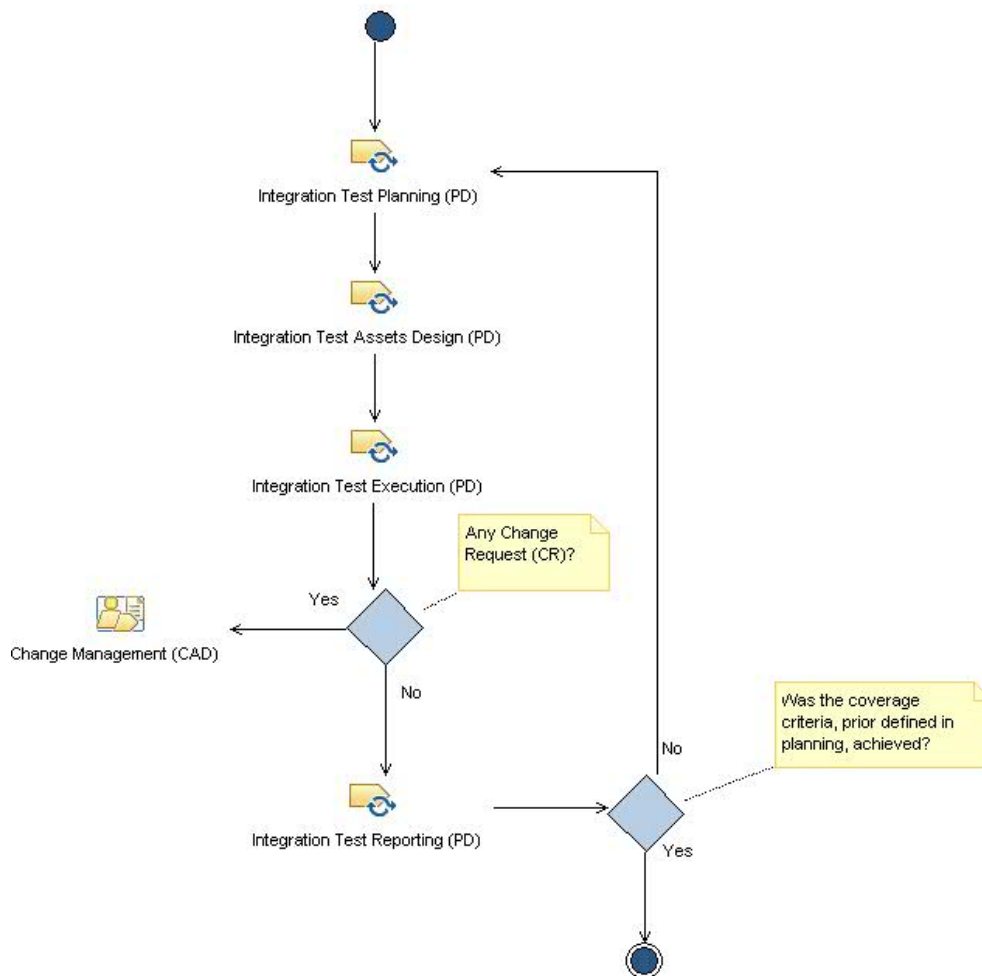


Figure 5.6 RiPLE Integration Testing level (PD) main flow.

As the products are implemented, the reference architecture is instantiated and adapted in order to meet the product specific needs. The term adaptation refers to the binding of optional and alternative variants, modification of components dependencies, and the addition of new components, which result in multiple product architectures within the

same line. The products, thus, differ in availability or comprehensiveness of their features (Knauber and Hetrick, 2005), which implies that each component should be tested in every possible configurations. Clearly, the effort to test every configuration is almost unfeasible (Muccini and van der Hoek, 2003).

The same set of steps proposed in the previous subsection is used to produce integration test cases during PD. As output of this phase, we have the accomplished product architecture, and its respective reports and test plans.

5.7 Example using the Approach

We illustrate our integration testing approach with a simple example to understand it in a better way. Its main goal is to use and understand the proposed approach in the sense of activities, roles, artifacts, strategies. This initial use was intended to find elements to be further refined and hence calibrate the proposed approach.

It was used within a graduate course at the UFPE⁴ (Federal University of Pernambuco, Brazil), in which a software factory composed by 9 M.Sc and 4 Ph.D. students, 1 customer and 3 domain experts was defined, in order to work in a SPL project. The conference management domain was chosen to be applied in this project, aiming at developing a core asset base, and next derived a set of three products. This system is responsible for manage manuscripts, conferences and workshops. In this study, we are using the manuscript management domain. Firstly, the author can submit a manuscript, the conference committee evaluate it and starts the review process designating reviewers to start the manuscript evaluation. After that, the author receives a notification which contains the reviewers decision.

This project was conducted following the RiPLE (the RiSE Process for Product Line Engineering), including the whole set of disciplines it encompasses: RiPLE-SC (Scoping), RiPLE-RE (Requirements), RiPLE-DE (Design), RiDE (Implementation), RiPLE-EM (Evolution Management), and thus RiPLE-TE (Testing). In this context, the RiPLE-TE-Integration (henceforth named RiPLE-TE-IT) approach for architectures verification and validation was also applied.

The first step towards the use of the RiPLE-TE-IT was to devise a *test plan*. This document encompassed the list of the components to be integrated and which paths of this integration would be verified (coverage criteria) at this time. The defined strategy to test the commonality during CAD and the variability in PD should be described in this

⁴www.cin.ufpe.br

5.7. EXAMPLE USING THE APPROACH

document. The use of another testing strategy should be also described in the test plan, since it will guide the overall testing activities. A summary regarding to each SPL testing strategy was previously mentioned in Chapter 4, Section 4.5.2.1.

To better understand how the integration tests are designed, excerpts from the feature model and feature diagram used in the project were adopted in order to represent feasible feature interactions. By analyzing both diagrams, the *Test Architect* can figure out how the features interaction occur. As an illustrative example, it is possible to see in Figure 5.7 that the review management feature interacts with others by a *decomposition* (i.e. when a feature is decomposed in others) and *usage* (i.e. when a feature require another) relationship. A sequential relationship can be viewed between *Accept/Reject Review* and *Document Acceptance/Rejection* features. The decomposition relationship is naturally expressed in the feature model.

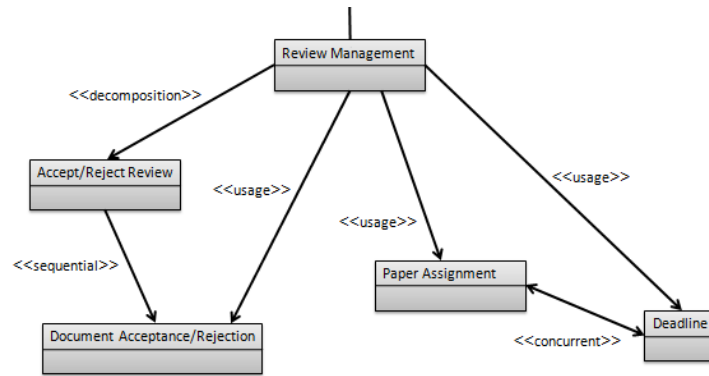


Figure 5.7 Feature Dependency Diagram.

Features	RiSE Chair Conference		RiSE Chair Journal		RiSE Chair Plus		Total		Scope
	Fut	Req	Fut	Req	Fut	Req	D(c)	P(c)	
Notification - Deadline	0	1	0	1	0	1	0,00	0,00	Mandatory ▼
Notification - Document Acceptance/Rejection	0	1	0	1	0	1	0,00	0,00	Mandatory ▼
Notification - Event News	1	0	1	0	0	1	0,00	0,00	Variable ▼
Notification - Paper Assignment	0	1	0	1	0	1	0,00	0,00	Mandatory ▼

Figure 5.8 ProductMap.

In the next step, architectural views (e.g: sequence diagrams, class diagrams, etc.) together with use cases and requirements are analyzed in order to build the sequence diagrams, based on the coverage criteria defined in the test plan. The test cases and scripts are designed based on these set of information. Figure 5.9 shows a high level sequence

diagram, where the user can submit a paper to the system, which assign it to a reviewer. The variation point (VP1) represented by “deadline” feature (Figure 5.7), it is optional and may be bound or not. The same happens to VP2, indicating that the user can receive event news.

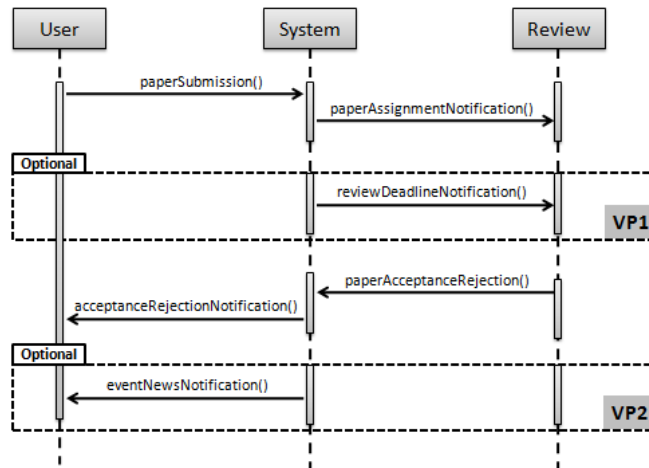


Figure 5.9 Sequence Diagram.

As the requirements, use cases and feature interaction diagram serve as inputs to compose the sequence diagrams. Class diagrams can also be used in order to aid the design of integration tests. Such diagrams enable Test Architects to better understand how the components and modules interact.

The sequence diagram represents the integration between two architecture modules, the “core business” module, composed by submission and review management components, and “notification” module. Figure 5.10 shows the reference architecture modules.

During CAD integration, the components and modules which compose the reference architecture are bound, considering the previous scenario. The variation points are instantiated but the decision regarding the correct moment to test should be aligned with the test strategy previously adopted.

Considering the strategy adopted in the first step (Test Plan), these variation points, features and components will be bound and integration tested according to the product derived from the product line during PD integration. This information can be extracted from the *product map*, Figure 5.8, where three products (RiSE Chair Conference, RiSE Chair Journal and RiSE Chair Plus) are described in terms of Notification features.

The test cases were developed using the Junit Framework (JUn, 2009), which provides an efficient way to generate and automatically execute test cases. A coverage tool (Ecl,

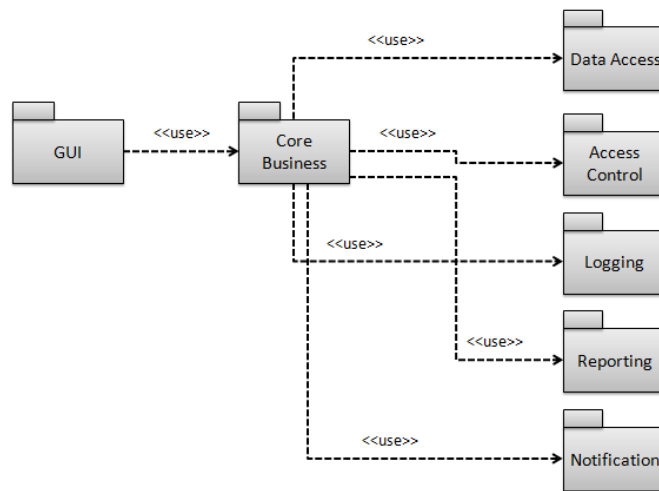


Figure 5.10 Architecture modules.

2009) was also used in order to show the test case coverage. A set of coverage criterias can be used: structural or functional, in our study a structural coverage was used.

After the test execution, a test report is generated gathering information about the issues found, pass/fail rate and coverage criteria adopted.

5.8 Chapter Summary

This chapter provided a systematic approach for dealing with integration testing in the context of software product lines. The way the approach is designed, including roles and attributions, activities, steps, data flow, i.e., a structured process view, can enable testers to adopt such a process in any SPL project.

The idea to design such an approach emerged by searching the literature looking for the state-of-the-art and practices in this topic. We realized that existent processes or methods did not define a systematic or structured way, so that tailoring them to a SPL project could be a very laborous work.

The proposed approach covers the both, core asset development and product development processes, thus, once applying this process, someone can test architecture conformance and product specific architectures, as well as the integration of product specific components to the corresponding product architecture.

Next chapter will present an overview on regression testing and define a regression testing approach for software product line architectures. It also presents some scenarios where this approach can be applied.

“Knowledge is limited; but imagination encircles the world.”

Albert Einstein



A Regression Testing Approach for Software Product Lines Architectures

In the Software Product Lines context, where products are derived from a common platform, the reference architecture is considered the main asset. In order to maintain its correctness and reliability after modifications, a regression was developed. It aims to reduce the testing effort, by reusing test cases, execution results, as well as, selecting and prioritizing an effective set of test cases. In addition, regression testing can find errors that were not detected during unit and integration phases. Taking advantage of SPL architectures similarities, this approach can be applied among product architectures and between the reference and product architecture.

The remaining of this chapter is organized as follows: in the next Section, it is presented a background with some regression testing concepts (maintenance categories, regression testing types and test classes) jointly with some general information inherent to typical selective retest technique and its associated problems. In Section 6.4, a brief overview about integration testing is described. Section 6.5 shows three scenarios where the defined approach can be applied. In Section 6.6 the architecture regression testing approach is described. Section 6.2, presents a succinct discussion and the related work. Finally, Section 6.7 presents the chapter summary.

6.1 Introduction

In order to achieve the ability to produce individualized products, companies need high investments which lead sometimes to high prices for a individualized product. Thus, many companies, started to introduce the common platform in order to assemble a greater variety of products, by reusing the common parts. In the Software Product Lines (SPL)

context, this common platform is called the reference architecture, which provides a common, high-level structure for all product line applications (Pohl *et al.*, 2005a).

In addition, software architectures are becoming the central part during the development of quality systems (Shaw and Clements, 2006), being the first model and base to guide the implementation (Muccini *et al.*, 2006) and provide a promising way to deal with large systems (Harrold, 2000). Nevertheless, it evolves over time in order to meet customer needs, environment changes, improvements or corrective modifications. Thus, in order to be confident that these modifications conform with the architecture specification, did not introduce unexpected errors and that the new features work as expected, regression test is performed (Orso *et al.*, 2004).

Considering testability in architecture design, testing activities can be made more efficient and effective (Kolb and Muthig, 2006), since when a modification occur few paths will be affected. Thus, few test needs to be rerun, few obsolete test cases will be removed and few new test cases need to be designed (created). Moreover, when the changes are so common (Svahnberg and Bosch, 1999), maintainability is one important criteria when developing software (Staff, 1992); if the regression testability is not considered since early phases, more hard and expensive are the modifications and test and retest activities.

The main problem when considering a retest-all strategy, useful in safety-critical domains, is that it can consume excessive time and resources (Rothermel and Harrold, 1996). Thus, the adoption of a regression test selection technique is inviting in some scenarios and domains. For example, in avionics context, where the reduction of one test case may save thousands in testing resources (Harrold *et al.*, 2001). Basically, it selects a set of test case from existing test suites to test the original version, avoiding the execution of all test cases. However, the test selection technique is only justifiable when the cost to select test cases is less than to run the entire test suite.

Moreover, Harrold (2000) advocates that regression testing can be used during maintenance to test new or modified portions or during development phase, to test similar products, safety-critical software and software under constant evolution. In addition, it can be useful to make confidence in the correctness of the software, increasing its reliability Wahl (1999), as well as, identifying errors that were missed, after applying traditional code-level testing Muccini *et al.* (2006). However, applying regression testing to SPL is not trivial and requires some extra efforts Kolb (2003). According to Kolb (2003), the major problems in testing product lines are the large number of variations, redundant work, the interplay between generic components and product-specific components, and regression testing.

SPL testing is a multi-faceted problem that has connections to regression testing, testing of incomplete programs, and efficient use of reusable test assets. The SPL members can be seen as variants of each other, which makes regression testing inviting. In this sense, an architecture regression testing approach was defined taking advantage of regression test benefits and SPL architectures similarities, by selecting and prioritizing a set of effective and efficient test cases, based on information previously collected. There are some scenarios, considering SPL context (*Core Asset Development (CAD)* and *Product Development (PD)*), where the use of the proposed regression testing approach is useful, for example: (i) during reference architecture evolution and modification, (ii) when changes in the product architecture (PA1) should be propagated through the overall product line, (iii) maintenance of the conformance among product architectures and the reference architecture, and (iv) to address the problems raised from a typical selective retest technique.

6.2 Other Directions in SPL Regression Testing

The formal notations used to describe software architecture specification serve as basis on which effective testing approaches and techniques can be developed. For example, Knodel and Lindvall (Duszynski *et al.*, 2009) present a tool which analysis the compliance of existing systems to control and asses its implementation with their architectures. Kolb and Muthig (Kolb and Muthig, 2006) consider that test can be more efficient and effective by considering testability in architectural design. Winbladh et. al. (Winbladh *et al.*, 2006) present a specification-based testing approach that verifies software specifications, as software architecture against system goals. Muccini and Hoek (Muccini and van der Hoek, 2003) report that test product line architectures is more complex than software architectures and present some activities.

In an important survey in the testing area, Bertolino (2007) proposes a roadmap to address some testing challenges, discussing some achievements and pinpoint some dreams. Concerning to SPL, she describes the challenge “Controlling evolution” as a way to achieve the dream “Efficacy-maximized test engineering” highlighting the importance of effective regression testing techniques to reduce the amount of retesting, to prioritize regression testing test cases and reduce the cost of their execution. Briefly, it is important to scale up regression testing in large composition system, define an approach to regression testing global system properties when some parts are modified and understand how to test a piece of architecture when it evolves.

6.3 A Regression Testing Overview

Although unit, integration and system test levels have their importance to detect specific defects, regression testing is a way to efficiently test the conformance after a modification. Instead of submitting the modified software to all test levels, regression testing is applied, reducing costs and detecting faults early.

In the following sections, some concerns related to regression testing are detailed, as means to provide information to better understand the proposed approach.

6.3.1 Maintenance Categories

Characterized by their huge cost and expensive implementation, maintenance initiates after the product release and aims to correct, keep the software updated, as well as fit with the environment new needs. According to (Pressman, 2001), around 20% of all maintenance work is spent fixing mistakes, the remaining 80% are spent adapting the system according to the external environment needs, making enhancements requested by users and reengineering an application for future use. One way to reduce the maintenance cost can be achieved by an efficient and effective regression testing.

In (Lientz and Swanson, 1980) and ISO/IEC 14764 (Iso, 2006), four categories of maintenance are defined, as follows:

- *Adaptive Maintenance*: Aims to adapt the system in response to data requirements or environment changes;
- *Perfective Maintenance*: Addresses the modifications after product delivery to handle any enhancements in respect of system performance or maintainability improvements;
- *Corrective Maintenance*: It is a reactive modification of a system, usually called “fixes” and performed after delivery. It is responsible for fix discovered problems (software, implementation and performance failures); and
- *Preventive Maintenance*: It is concerned to correct and detect faults before it becomes a fault, preventing problems in the future.

During the adaptive or perfective maintenance, the software specification is modified to join the improvements or adaptations (Wahl, 1999). In corrective maintenance, the specification may not be modified or no new modules may not be added. Most of the

changes imply in addition, modification and deletion of instructions (Leung and White, 1989). Preventive maintenance is usually performed on critical systems (Abran *et al.*, 2004).

In (Hatton, 2007), Hatton analyzes the first three categories (Adaptive, Corrective and Perfective) using five studies and indicates the distribution regarding to the spent time over these categories. The results are presented in Table 6.1.

Table 6.1 Software Maintenance Categories Distribution Hatton (2007).

Study Authors	Adap.(%)	Corr.(%)	Perf.(%)
Dekleva	46	18	25
Helms and Weiss	29	19	28
Glass	42	37	23
Sneed	52	9	35
Kemerer and Slaughter	83	12	5

6.3.2 Corrective vs Progressive Regression Testing

Based on the possible modifications, regression testing can be classified in two classes (Staff, 1992),(Leung and White, 1989).

Corrective Regression which is often performed after some corrective action on the software, it is applied when specifications are unmodified (e.g. when the code is not in conformance with the specification). When the modification affects only some instructions and design decisions (e.g. changing only the way used to implement the variability inheritance, parameterization and design patterns without modify the specification), it makes the test cases from the previous test plan be more reused. However, when they involve possible changes to the control and data flow structures, some existing test cases likely to be no longer valid to testing that portion of the software. Since program failures can occur any time, this type of regression testing should be applied for every correction.

Progressive Regression is typically performed after adaptive and perfective maintenance (Section 6.3.1), it is used when specifications are modified (e.g. the addition of a new feature or functionality). This specification modification is caused by new enhancements or new data requirements, which should be incorporated in the system. In order to handle the testing of this modification, new test cases need to be designed. This type of regression testing is performed during regular intervals since adaptive or perfective maintenance is typically done at a fixed interval (e.g. every six months).

6.3.3 Test Case Classes

Let P be a program, let P' be a modified version of P , and let T be a test suite created to test P . The main idea behind regression testing techniques is to select a subset of tests T' of T to make confidence that P' was correctly modified, still working properly as previously and no new errors were inserted (Rothermel and Harrold, 1996).

During architecture evolution or modification it may affect the specification, as a result, the architecture structure implementing the specification must be changed. However, when the specification is not modified only the architecture structure is changed.

In (Leung and White, 1989) and (Briand *et al.*, 2009), the authors categorize test cases created in the previous phase (integration testing) from the previous test plan in the following classes:

- *Reusable Tests*: Responsible for testing a unmodified portion of the specification and architecture structure. They are still valid but do not need to be executed again to guarantee the regression testing safety;
 - *Retestable Tests*: This class includes all tests that should be repeated because the software structure was modified, even though the specification regarding to the software structure are not modified. They are still valid and need to be rerun;
 - *Obsolete Tests*: Comprehend the test cases that cannot be executed on the new version, since they become invalid for the new context. According to (Leung and White, 1989), there are three reasons for that:
 - The structural tests (based on the control and data structures) are designed to increase the structural coverage of software. Since the structure can be changed of different versions of the software, some test cases become obsolete, because they are not contributing with the software structural coverage;
 - Due to some changes in a specific software component, some test cases may not be testing the same structure, despite they correctly specify the input/output relation; and
 - When the test cases specify an incorrect input/output relation. It happens when a specification is modified and the related tests are not according updated.
 - *Unclassified Tests*: Involve the test cases which may either be *retestable* or *obsolete*. According to (Leung and White, 1989), two new classes of test cases can be included in the test plan:
-

- *New-structural tests*: Include tests which aims to test the modified software structure. Often they are design to improve the structural coverage;
- *New-specification tests*: Comprehend the test cases that evaluate the new code generated from the modified portion of the specification.

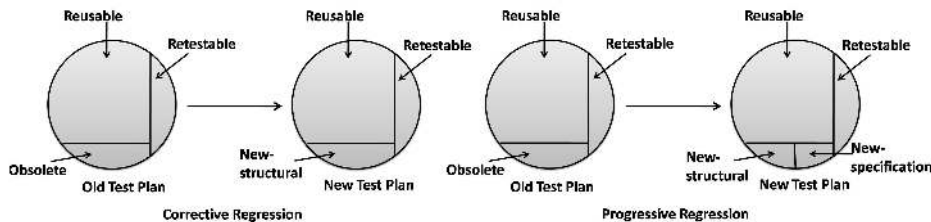


Figure 6.1 Corrective and Progressive Regression Testing Leung and White (1989).

In order to better understand the relation between the types of regression testing (Section 6.3.2) and they correspondent test classes, Figure 6.1 shows this relation (adapted from (Leung and White, 1989)). Left side of Figure 6.1 shows that after performing a modification, obsolete test cases are removed and new-structural tests are added to the new test plan. Right side of Figure 6.1 shows that besides remove obsolete test cases, new-structural and new-specification test cases need to be design to test the modified version of the software. Since in this type of regression testing the specification is modified, new-specification test cases are developed.

6.3.4 Typical Selective Retest Technique

When regression testing approach is applied, an important prerogative is how to select a subset of test cases, from the original test suite, as means to test the modified version of the software (Orso *et al.*, 2004). To address this problem, a retest selection technique can be useful. Harrold *et. al.* in (Rothermel and Harrold, 1996), (Harrold, 1998), (Rothermel and Harrold, 1994), (Rothermel and Harrold, 1997), (Todd Graves, 1998) describe the steps involved in it, as well as, the problems arise from each step.

1. Select a set of test cases T' to execute on P' ;
2. Test the modified program P' with T' in order to establish the correctness of P' with respect to T' ;
3. If needed, create a new test suite T'' , a set of new-specification and new-structural test cases (Section 6.3.3) to test P' ;

4. Test P' with T'' in order to establish the correctness of P' with respect to T'' ;
5. Repeat the *step 3* selecting the test cases from T , T' and T'' to be executed in P' .

The first problem arises from *step 1*, which is the regression test selection problem: select a subset T' from T to test P' (Orso *et al.*, 2004). The coverage identification problem raised from *step 3*, consists in identifying portions of P' or its specification that needs additional testing. *Steps 2* and *4* involve the test suite execution problem: regarding to efficiently execute the test suites and checking their results. At last, the test suite maintenance problem addressed by *step 5*, the problem to update and store test information.

Selection regression testing idea came up from the need of reducing the cost of regression testing. For this reason, it has been broadly studied (Wahl, 1999). This cost reduction is achieved by reusing existing tests and identifying the modified portion of the software and specification that needs to be tested (Rothermel and Harrold, 1996).

For a regression test selection technique to be cost effective, the effort and time devoted during the test selection and its executions needs to be less than the overall cost to execute all test cases from the test suite (e.g. Retest-All) (H K N Leung, 1991). The test suite is another factor to be considered, since it needs to be big enough to justify the necessity of a test selection technique (Wahl, 1999).

6.4 Regression at Integration Level

As mentioned by McGregor (McGregor, 2001b), regression testing is a technique rather than a testing level. Burstein *et. al.* (Burnstein, 2003) define it, as being “the retesting of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes”. Considering this point of view, regression testing can be performed after any test level, in our context, it will be performed after integration testing, since the purpose of the approach is to verify the integration among modules and components which composes the SPL architecture.

In order to figure out which point of the testing process this approach is applied, a brief contextualization about integration test is presented.

After unit testing level, where the components are individually tested, integration testing comes in scene. The *product map*, which is a SPL artifact build during scoping phase that groups all products and its respective features Bayer *et al.* (1999), is analyzed

in order to identify commonality among SPL members. In addition, by analyzing the feature model and feature dependency diagram, the *Test Architect* can understand the relationship among features. These information are important during test design and test suite composition, since it shows the product features, as well as, shows how it interacts. Thus, the test cases are designed considering these interactions. During design, it serves as input to define modules and components that composes the reference architecture and, during testing, it guides the design of test cases which evaluate the interaction between components and modules (integration testing).

The architecture diagram, composed by components and modules are also important during testing phases, since it provides the way in which the components and modules interacts.

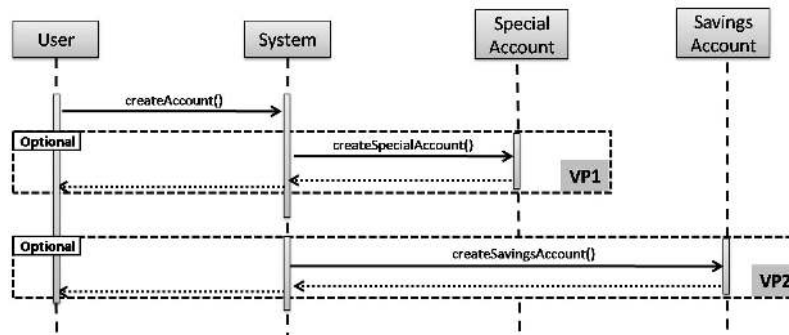


Figure 6.2 A Sequence Diagram with two variation points.

Based on previous information (feature model, product map, feature dependency and architecture diagrams) and architecture views, the integration tests can be designed. Whereas behavioral view (sequence diagrams) provides information about the functionalities of the architecture, the structural view (component and module class diagrams) give us information about the architecture structure. Figure 6.2 shows a sequence diagram in a scenario where a user requests for a account creation and the system can create two types of accounts, special and savings account. Two variation points (optional features) can be viewed, the first *VP1* represents the functionality responsible for creating the special account, the second one *VP2* represents the functionality that aims to create the savings account. The bind of *VP1*, *VP2* or both, should be specified in the product specific architecture. It is important to note that a test case can be designed to verify different scenarios and configurations, depending on the feature(s) that was/were bound.

6.5 Regression Testing in SPL Architectures

There are three different scenarios where the regression testing approach is attractive. Figure 6.3 shows all of them, which are following described:

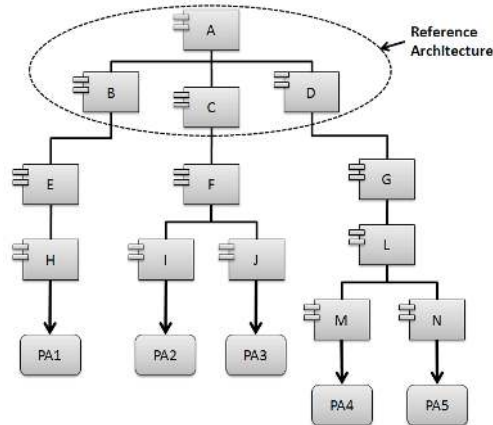


Figure 6.3 Similarities among product architectures.

- *Scenario 1:* Given a Reference Architecture (RA) Figure 6.4, composed by the integration of components (A,B,C and D), which has its conformance verified during integration testing (Section 6.4). Imagine that a component A should be modified to A' in order to reflect a change, due to a evolution or corrective action. A new version of the reference architecture (RA') is developed. Considering these two versions, the original V1 (A,B,C and D) and the new one V2 (A',B,C and D), they need to be applied against a regression testing approach which aims to gather confidence that the new version is free of faults and still working properly.

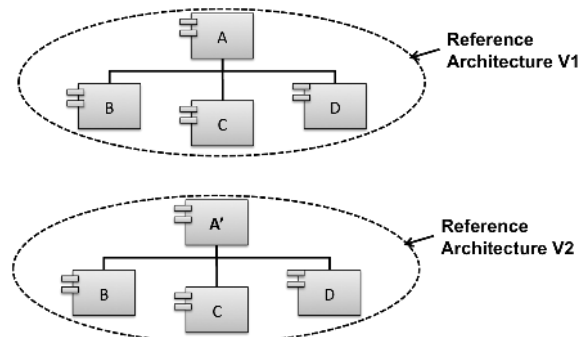


Figure 6.4 Two Reference Architecture Versions.

- *Scenario 2:* Given the reference architecture (RA) and the product architecture PA1. Considering these two architectures as difference versions V1 (A,B,C and D) and V2 (A,B,C,D,E and H), the regression testing approach can be useful (Figure 6.5). This scenario is also considered a testing strategy commonly used in SPL testing exploiting the existing commonalities among SPL members. While the first product is tested individually, the following products are tested using regression testing techniques (Tevanlinna *et al.*, 2004).

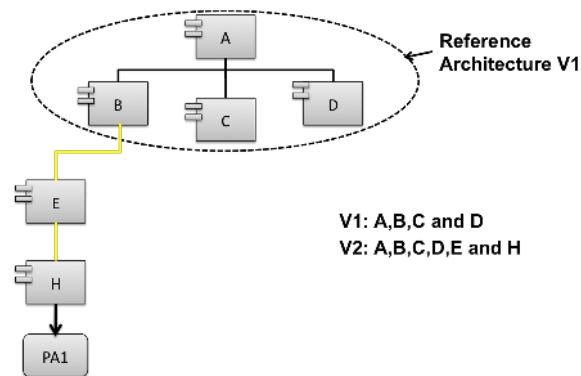


Figure 6.5 Reference Architecture and Product Specific Architecture.

- *Scenario 3:* Given two product line architectures PA4 and PA5, considering both as two different versions V1 (A,B,C,D,G,L and M) and V2 (A,B,C,D,G,L and N), the regression testing approach can be applied in this context. By observing the Figure 6.6, the reader can be induced to think that since the PA4 was previously tested during integration testing, PA5 can be verified reusing the common tests and only considering the integration of the last component N. It is a wrong consideration because the integration of the last component could bring faults in the previous tested structure, for this reason, the application of a regression testing approach is crucial to understand the impact of the last integration.

The regression testing approach defined was considered in two ways. Firstly, during CAD when it aims to test the conformance of the reference architecture (RA) after a modification in a component or module which is part of it. Later, during the PD with the purpose to test a product architecture in respect with the reference architecture or others product architectures considering their common features. The product line members are seen as variants of each other, making regression testing in PD attractive. The overall view of the testing approach is showed in Figure 6.12.

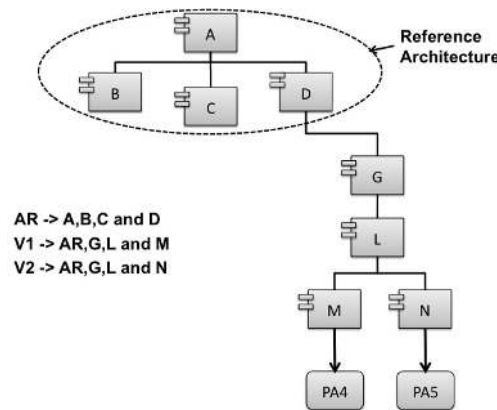


Figure 6.6 Similar Product Architectures.

6.6 A Regression Testing Approach for SPL Architectures

The purpose of regression testing on this phase is to check if new defects are introduced into previous tested architecture and it continues working properly. To be confident that the architecture is working properly, its specification can be used as test oracle to identify when the tests pass or fail.

The main *inputs* are the two versions (modified and original) of the architecture code, the test cases, test scripts and test suites from integration testing level saved to be further reused, all of these artifacts are considered mandatory in this approach. Architectural specifications as *behavioral* and *structural views*, as well as, the *feature model*, *product map*, *feature dependency diagram* can be useful to extract information from the architecture, serving as guide to identify portions that needs to be retested. Using the structural view, the relation among classes and components are clearly specified and identified. From this view and using the use cases (Riple-RE) and sequence diagrams (Riple-DE) previously built. They are used in order to better represent the relation among the components and classes, facilitating the creation of integration testing to be used in the regression approach. The feature model and feature dependency diagram are used to understand the relation among features, for example, in cases where a feature excludes another one and even the presence of optional features (Figure 6.2). These information should be considered when designing integration test cases.

Specific product architectures can be instantiated based on *product maps* and *decision models* which contain information such as *mandatory*, *optional* and *variant* features

for each product. Based on this information, the test architects are able to instantiate a architecture by selecting specific features and components. Thus, when components or modules are modified, regression testing must be performed on the application architecture, as means to evaluate its correctness Jin-hua *et al.* (2008).

6.6.1 Approach Steps

In this section the proposed regression testing approach is described. The overall approach can be viewed in Figure 6.7. Although they are presented as sequentially initiated, this process represents an incremental and iterative development step, since feedback connections enable refinements along the approach. This flow illustrates the approach workflow comprising its activities, inputs, outputs, tasks and involved roles. A complete view of it is shown in Figure 6.12.

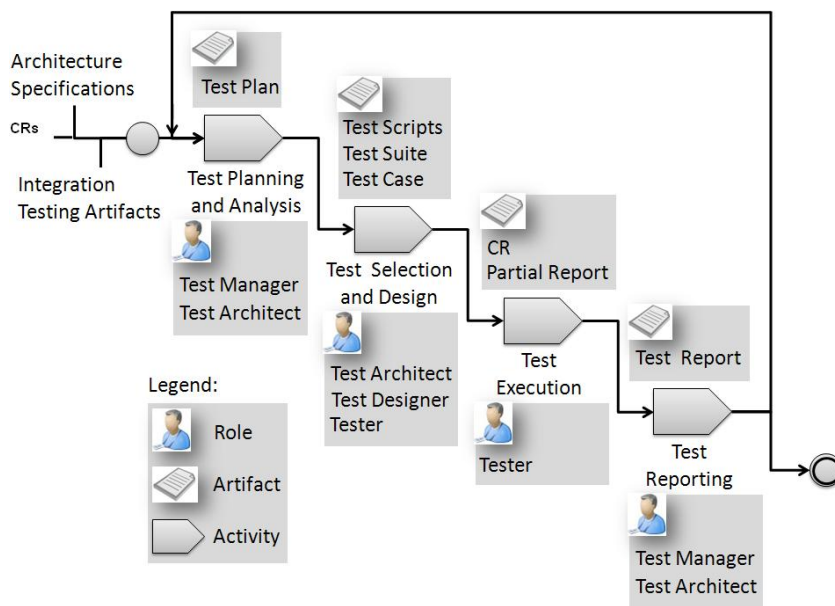


Figure 6.7 The Regression Testing Approach.

6.6.1.1 Planning

The planning is performed as means to guide the test cycle execution. In this phase, the *Test Plan* is created gathering information about the adequacy criteria, the coverage measure, resources and associated risks.

The *Test Plan* is a document which describes the scope, approach, resources, schedule of intended testing activities, the testing tasks, who will do each task, any risks requiring contingency planning, as well as a list of CRs that originate the modifications performed in the old version of the architecture. It aids in the identification of test items and the features to be tested. The test plan was done based on IEEE (1998).

6.6.1.2 Analyzes

The customer send a change request describing an issue during the credit function.

1. Analyzes

This step is performed in order to understand how a correction or evolution impact the architecture. By manually analyzing the architecture specification, the modified classes and methods are identified, and the relevant tests can be designed or selected based on this information (e.g.: comparing two class diagrams). It serves as guide to support the next steps, restricting the coverage of the modified version that should be examined.

After processing a change request or receiving an architecture evolution request, the test architect starts the analysis phase. Figure 6.8, shows an illustrative example of a class diagram with five classes of a bank system. In the context where a customer sends a change request describing an issue found in the credit function, the Test Architect will analyze the class diagram in order to identify the impacted classes. Considering that a modification was done in the credit method from Account class, it may cause problems (regarding to business rules) in credit method (Figure 6.9) from SpecialAccount class. If the SavingAccount class has a similar implementation as SpecialAccount, for example using “super.credit(value)”, this class will be also impacted. Considering this scenario the Test Architect can see that the method credit in the classes *Facade*, *Account* and *SpecialAccount* need to be investigated more carefully. It is important to highlight that some classes were removed from Figure 6.8 in order to facilitate the visualization and understanding.

Based on the category of the modification (Section 6.3.1), two types of regression testing (Section 6.3.2) can be performed, both types are handled by the approach.

Firstly, considering a *corrective scenario*, depending on the architecture size, the analyzes of overall architecture structure can be an expensive and hard task. In order to reduce the scope that must be studied to understand a modification, the test architect can optimize the analyzes. Studying the impact of the change by looking at the architecture views, it helps in the identification of test classes (Section 6.3.3) and isolates the area (architecture classes and methods) that needs to be retested, a walkthrough technique is

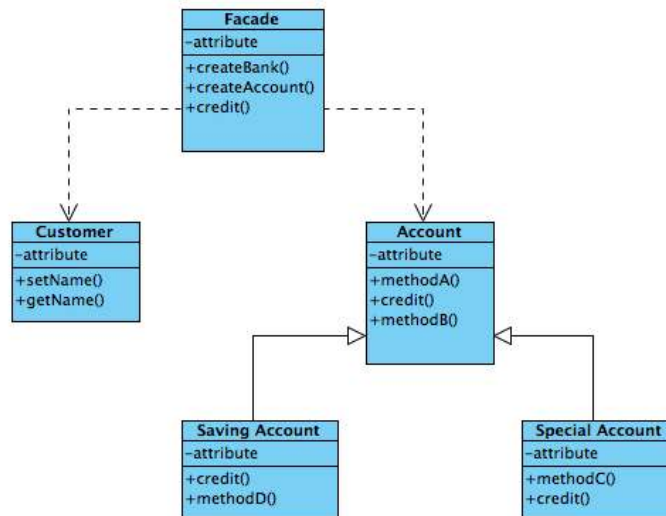


Figure 6.8 An illustrative example of a bank system class diagram.

```

public class SpecialAccount extends Account {
    public void credit (double value) {
        super.credit(value);
        ...
        bonus = bonus + (value * 0.01);
        ...
    }
}
  
```

Figure 6.9 Credit method from Special Account class.

recommended to perform this task. Some components, classes and methods are irrelevant to architecture's regression testing, since their change do not impact in others components, thus these irrelevant components, classes and methods, can be safely removed without lost probable critical paths.

Take into consideration the example described in the previous paragraph (Figure 6.8), the area that needs to be retested is the linking among Facade, Account and SavingAccount classes, on the other hand, tests that exercises the link between Facade and Customer do not need to be retested.

In cases where no information about the modification is available, the use of a diff tool (Textual Comparison) to start the analyzes step is advisable. It is used in the area in which the modifcarion was performed, also considering the code related to that change.

Considering a *progressive scenario*, when the architecture suffers a modification due to an evolution, the impact is also studied, to visualize where the modifications are located. Based on this analyze, the test architect can focus only in that relevant area. It

reinforces the importance to maintain the architecture specifications always converging with its implementation after performing modifications. If this synchronization is not kept, it can cause problems for evolution, maintenance and the comprehensibility of a system, this problem is known as “Architectural Drift” Rosik *et al.* (2008).

This analysis work as a filter in order to identify the modified architecture portion, restricting the search space.

6.6.1.3 Test Design and Selection

2. Graph Generation

After performing the analyzes step, the test architect may need to generate graphs to catch code behaviors. Thus, a graph representation for both versions of modified portions of the architecture (the new and old versions) are generated.

This graphs can be a control flow graph (CFG), program dependence graph, control dependence graph or a Java Interclass Graph (JIG) depending on the test selection technique. CFGs are suitable for representing the control flow in a single procedure, but it cannot handle inter procedural control flows or features of Java language such as polymorphism, dynamic binding, inheritance and exception handling Harrold *et al.* (2001). As much language features the graph represents, more refined will be the analysis, increasing the code coverage and decreasing the number of undetected faults.

Apiwattanapong *et al.* (2007) propose the Enhanced Control-Flow Graphs (ECFG) to suitably represent object-oriented constructs and model their behavior. They also present JDiff tool that generates ECFG representation for two program different versions and compare these versions. This tool considers both, the program structure and semantics of the programming-languages constructs Apiwattanapong *et al.* (2007).

In the proposed approach, the use of this type of tool is optional and depends on the specificity of the fault, in some cases, a simple textual comparison is able to found the critical path (or fault). When using a textual diff tool, it is important to select a person with high experience and knowledge in the architecture (domain) in order to identify the problems.

When textual differentiation is enough, this step and the next one are replaced by a diff tool.

3. Graph Comparison

In order to identify critical edges and understand how the code changed, the graphs are compared. A good knowledge in control flow graphs analysis is required during this step, since the Test Architect will see more easily how the code behaves. Figure 6.10

6.6. A REGRESSION TESTING APPROACH FOR SPL ARCHITECTURES

shows two versions of a program. In yellow are the differences between the two versions. In order to better understand the behavior of the code after this change, the Figure 6.11 shows the ECFG for both versions.

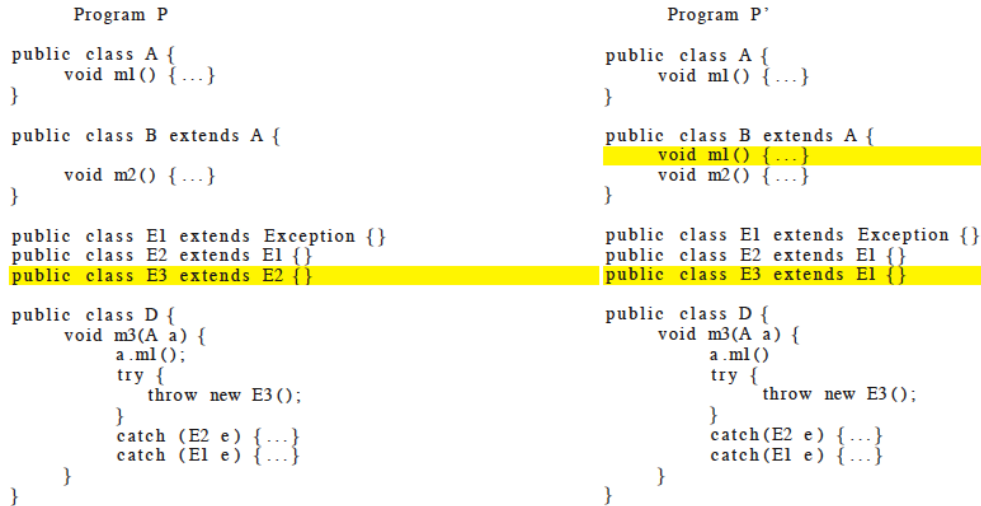


Figure 6.10 Two different versions of a program (Apiwattanapong *et al.*, 2007)

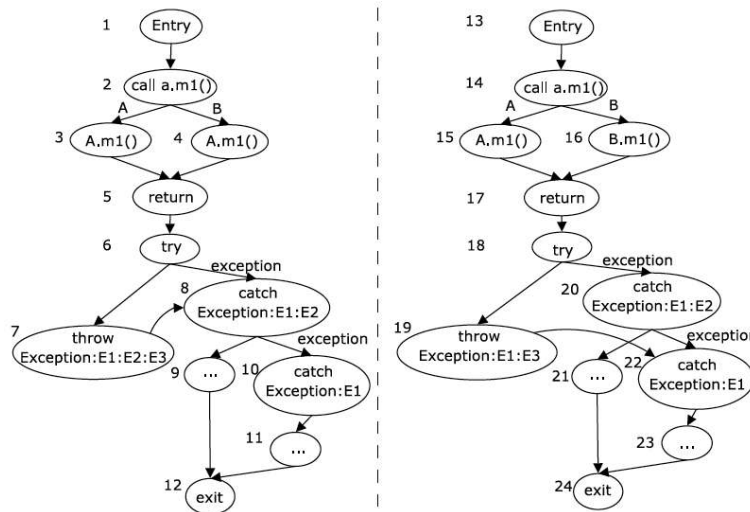


Figure 6.11 Two different versions of a method (Apiwattanapong *et al.*, 2007)

When performing progressive regression testing, the last two steps (graph generation and comparison) are replaced by specification comparison. This step aims to compare the original specification with the modified one, identifying added, deleted or changed components, classes, features.

4. Test Design and Selection

After the graphs, specification or textual(code) comparison, the critical edges and paths are analyzed aiming to find some test that exercise a modified portion of the architecture. In this step, the Test Architect analyses the paths trying to classify the previous designed tests (integration tests) from the repository according to the classes established in Section 6.3.3. It will help during the suite composition, since all relevant test cases will be identified. A good knowledge about the SPL architectures and expertise are required from the Test Architect to perform this step, since he need to understand the change and how it impact over the code, always considering the variation points and its variants.

When the correction or evolution involve structural or specification changes, some test cases (and/or scripts) need to be designed to reflect the new architecture constructs. Not only new test cases need to be designed, but also some of them need to be redesigned (updated) to cover a specific modified portion of the architecture. An important aspect when dealing with test case update and design is how to keep track (mapping) the test case with the architecture code portion. A simple modification may impact in a large number of tests, making update tasks expensive. The more suitable is this mapping, less escaped defects and more easy to maintain the test suite composition, since a simple code modification will revel which test cases should be updated.

5. Instrumentation

To check about the test cases efficiency and coverage, the identified paths and new code from the previous step can be instrumentalized. Doing so, Test Designers will be confident that the tests really exercise the desired paths/code. If the selected test case did not cover the required path, a new test case (or script) should be designed.

In additional, this step can be useful to detect false positive and false negative test cases. *False positive* happens when the verification activities inform that the asset is correct when it is not, it can occur due to a wrong test case or a less than complete test set. This can be a dangerous mistake in critical systems, since we cannot double check every positive result McGregor (2009). *False negative* happens when verification activities indicate that the asset is not correct when it is. It is safer than false positive but also expensive. Resources are used unnecessarily to attempt to fix what is not broken McGregor (2009). Moreover, incorrect test cases may generate false negatives.

6. Test Suite Composition

After the test case selection and design, a test suite is composed using them. The *Test Designer* can create test suites grouping tests based on different information, for example,

a test suite responsible for exercising a determined feature, component or even a specific functionality. This test suite will be used to build the regression test cycle to be further executed.

7. Test Case Prioritization

This prioritization aims to order the test cases and scripts from the test suite, executing tests with highest priority, based on some criterion (e.g. criticality or complexity implementation), earlier than lower priority test cases. Prioritization techniques may be used, some of them take advantage of some information about previous executed test cases to order the test suite Rothermel *et al.* (2001). Testers might wish to schedule test cases in a sequence that cover all the critical (most important or instantiated) variabilities implementation first, exercises features from a specific product, or tests which cover a specific architecture quality attribute. Rothermel et. al. in Rothermel *et al.* (2001) analyze some prioritization technique and show that an improvement can be achieved even with the least expensive of those techniques.

6.6.1.4 Execution

During the test execution phase, the test suites are executed against the modified version in a regression testing cycle. The *Test Engineer* exercises the architecture, executing the test cases. If some inconsistency is observed, he should search in the repository for a CR that reports the problem, in case where no CRs is found, a new one should be raised. The execution results and the new and associated change requests (CRs) with their respective responsibilities are recorded and an investigation starts in order to precisely identify which components, modules, versions and modification caused the failure.

Depending on the failure, the regression test approach will forward the damaged portion to unit test or integration test, for the purpose of creating a test case to cover that path.

6.6.1.5 Reporting

All these information will be gathered to further compose the *Test Report* during reporting phase. This report is extremely important for the *Test Manager* since he will use this information for component, architecture or product schedules and also to build other test plans.

A full view of the regression testing approach is showed in Figure 6.12.

6.7 Chapter Summary

The growth of new technologies, such as component-based systems and product lines, and the emphasis on software quality, reinforce the need of improved testing methodologies (Harrold, 2000). Current test techniques need that software architecture based approach be completely rerun from scratch for a modified software architecture version (Muccini *et al.*, 2006).

This regression testing architecture approach is part of a testing process for SPL projects, in which unit and integration testing are also considered. As reported previously, it is applied during integration testing.

This approach was developed in order to handle test selection problems, raised when a SPL architecture needs to be retested, as well as, to deal with SPL features, reusing test suites and execution results as much as possible. This approach can be applied in three main scenarios: firstly, when the architectures (reference and product architecture) are modified, it is used to compare the two versions and select effective test cases. Secondly, it is useful to maintain the conformance between reference architecture and product architectures, preserving their compatibility. At last, during product derivation, a specific product architecture is instantiated and tested. Taking advantage of the product architectures similarities this approach can be useful to select test cases.

In order to better evaluate the proposed approach an experimental study was performed and presented in the next chapter.

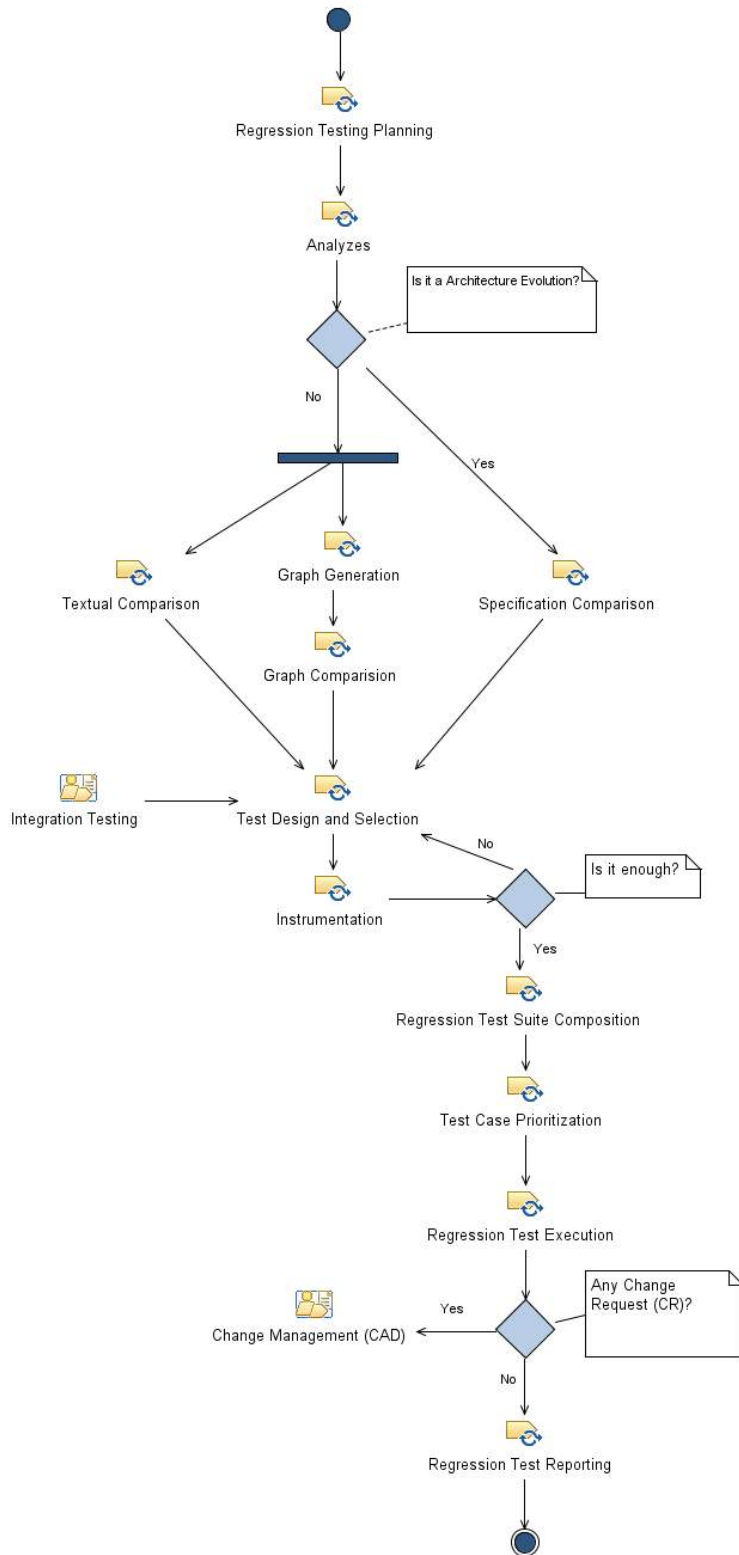


Figure 6.12 The Overall Regression Testing Approach.

“One who tried and did not achieve is superior than that one which never tried.”

Archimedes

7

The Experimental Study

7.1 Introduction

In the previous chapters, a regression approach was defined as being a technique employed during integration level. The integration approach aims to verify if the SPL architecture is in conformance with their specification. When this architecture is modified or suffers some evolution, a regression testing approach is applied in order to verify if the new version still working properly. Some scenarios where the regression approach may be applied were also described, as well as their activities and artifacts.

This chapter describes an experimental study discussing its definition, planning, operation, analysis and interpretation, as well as, other aspects concerning empirical experiments. The remaining of this chapter is organized as follows: Section 7.2 presents the definition of the experiment; in Section 7.3, the planning of the experiment is presented; Section 7.4 describes how the operation of the experiment was performed; in Section 7.5, the analysis and interpretation of the results are presented; and finally, Section 7.6 presents the lessons learned.

7.2 Definition

With the purpose to define this experiment, a mechanism for defining and evaluating a set of operational goals using measurement was used: the Goal/Question/Metric (GQM) mechanism (Basili *et al.*, 1986, 1994) which has three levels:

- *Goal or Conceptual level:* It is defined for an object, for many reasons, with respect to several models of quality, from various standpoints, relative to a particular environment.

- *Question or Operational level*: It is used in order to characterize the way the assessment of a goal is going to be performed. It usually breaks down the issue into its major components.
- *Metric or Quantitative level*: The questions are refined into metrics, which can be classified in two classes. The objective, when it depends only on the object that is been measured and subjective when it depends on both object and the standpoint from which they are taken.

7.2.1 Goal

The goal is formulated from the problem to be solved. In order to capture its definition, a framework has been suggested by Wohlin *et al.* (2000). It is described as follows.

Object of study (*What is studied?*). The object of study of this experiment is the regression testing approach, likewise their activities, steps, artifacts and roles.

Purpose (*What is the intention?*). Verify its applicability in a designed SPL architecture. In addition, metrics are collected with the purpose to improve the approach understandability, completeness, applicability and effectiveness, and minimize the risks of applying it in a real and critical scenario.

Quality focus (*Which effect is studied?*). The benefits gains obtained by the use of the approach, which will be assessed by the number of defects found and the difficulties found by the subjects during its understanding and use.

Perspective (*Whose view?*). There are two perspectives in this experiment, one from the researcher point of view assessing the viability of the approach use and another one from the test engineer.

Context (*Where is the study conducted?*). The experiment environment is composed by seven software testing specialists, all of them M.Sc. students, and one M.Sc. in the component testing area, all of them from the Computer Science department at Federal University of Pernambuco, Brazil. In addition, the experiment will be performed distributed, which means that the subjects are free to choose their work environment (home or university laboratories). Regarding to the data, a set of classes and their integration are used in it, in order to simulate the integration of architecture components. The study is conducted as a Multi-test within object study (Wohlin *et al.*, 2000).

7.2.2 Questions

To achieve the goal previously defined, some quantitative and qualitative questions were defined and described as follows.

- **Effort:**

Q1. How much effort does it take to apply each step defined in the approach?

- **Usability and Understandability**

Q2. Do the subjects have difficulties to understand/apply the approach?

- **Completeness**

Q3. Is there any missing activity, roles or artifact?

- **Effectiveness**

Q4. How many defects were detected using the approach?

Q5. How many tests were correctly classified (Re-testable, Reusable, Obsolete and Unclassified)?

7.2.3 Metrics

Once the questions were defined, they need to be mapped to a measurement value, in order to characterize and manipulate the attributes in a formal way. The metrics are quantitative ways to answer the questions.

M1. Effort to Apply the Approach (EAA)

Related to Question **Q1**, this metric measure the the amount of time spent in order to understand and follow the Regression Testing approach and produce the artifacts proposed.

$$EAA_{step} = \frac{TotalTimeSpentApplyingEachStep}{TotalTimeSpentInTheApproach}$$

M2. Approach Understanding and Application Difficulties (AUAD)

Related to Question **Q2**, this metric aims to identify possible misunderstandings in the approach usage, it is necessary to identify and analyze the difficulties found by users when applying the approach.

AUAD = Number of subjects with difficulties raised during the approach learn and application.

M3. Activities, Roles and Artifacts Missing (ARAM)

Related to Question **Q3**, it intends to identify the activities, roles and artifacts considered absent from the regression testing approach in order to calibrate or even include them, depending on the analysis.

ARAM = Number of missing activity/steps/role/artifact identified during the approach execution.

M4. Number of Defects (ND)

Related to Question **Q4**, it intends to identify the total number of defects in a given time period/activity/step in the software.

ND = The number of seeded defects identified, during the approach execution.

M5. Number of Tests Correctly Classified (NTCC)

Related to Question **Q5**, it aims to identify the correct classification of the test cases used and designed during the approach execution.

NTCC = The number of tests correctly classified (Re-testable, Reusable, Obsolete and Unclassified). It is important to select the test cases that need to be executed in the software new version.

7.2.4 Definition Summary

Analyze the regression testing approach for the purpose of evaluation with respect to understandability, usability, completeness, applicability and effectiveness from the point of view of SPL researchers and test engineers in the context of a software product line project.

7.3 Planning

While the definition determines the foundation for the experiment, *why* the experiment is conducted, the planning prepares for *how* the experiment is conducted. The latter, can be

divided in six steps (Figure 7.1), further detailed in the following sub-sections.

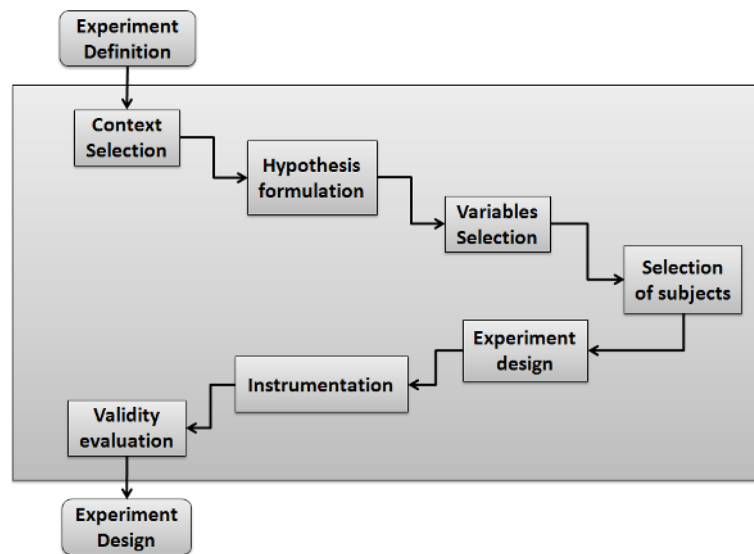


Figure 7.1 Planning phase overview (Wohlin *et al.*, 2000).

7.3.1 Context Selection

In order to achieve the most general results in an experiment, it should be executed in large, real software projects, with professional staff. However, conducting an experiment involves risks and not always the required resources, time and money are available. These issues require a balance between making studies valid to a specific context or valid to the general software engineering domain. Therefore, this context of the experiment can be characterized according to four dimensions (Wohlin *et al.*, 2000).

1. *Off-line versus On-line*: In this case, it is **Off-line**, since it was performed out of the semester lessons.
2. *Student versus Professional*: **Students** are the final user population to use the approach.
3. *Toy versus Real Problem*: The experiment addresses a **toy problem**, since it considers the approach in the integration of classes instead of components.
4. *Specific versus General*: This investigation concerns a **specific problem**, when the code suffer a modification due to a corrective action or evolution.

The context is also characterized according to the number of objects and subjects involved in the experiment. As previously mentioned, this experiment is a **Multi-test within object study**, since it examines one *object* (Regression testing approach) and more than one *subjects* (eight students).

7.3.2 Hypothesis Formulation

The basis for the statistical analysis of an experiment is the hypothesis testing. If the hypothesis can be rejected then conclusions can be drawn based on the hypothesis testing under given risks (Wohlin *et al.*, 2000).

The experiment definition is formalized into hypothesis: (i) The **Null Hypothesis (H_0)** which states that there are no real underlying trends or patterns in the experiment setting. This is the hypothesis that the experimenter wants to reject with as high significance as possible. (ii) The **Alternative Hypothesis (H_α)**, this is the hypothesis in favor of which the null hypothesis is rejected.

As the variables are subdivided in three factors (Section 7.3.3), three null hypotheses must be stated.

H_0 : It determines that the application of the regression testing approach in SPL architectures does not produce benefit that justify its use, demonstrating a poor understandability, effectiveness, completeness and applicability and usability defined in 7.2.4.

$$H0_1 : \mu_{EAA} \geq 20\%$$

$$H0_2 : \mu_{AUAD} \geq 40\%$$

$H0_3 : \mu_{ARAM} < 3$ (Having in mind that the approach has four activities (with twelve different steps), we came to the value of 25% (≈ 3 steps) as being a reasonable number).

H_1 : It determines that there is no benefits or gain of using the regression testing approach to find defects in the SPL architecture.

$$H1_1 : \mu_{ND} \geq 20\%$$

H_2 : It determines that there is no gain of using the regression testing approach in order to classify the existing test cases.

$$H2_1 : \mu_{NTCC} \geq 40\%$$

An important aspect is that these previously defined metrics were never been used, for this reason, an arbitrary value was chosen, based on practical experience and common

sense, since there is no well-known value for it. These arbitrary values will serve as basis to get new values or confirm the previous one, to perform new experiments.

The alternative hypotheses, consequently, are described next:

$H_{\alpha 0}$: It determines that the application of the regression testing approach in SPL architectures produces benefits that justify its use, demonstrating a good understandability, effectiveness, completeness and applicability.

$$H\alpha_1 : \mu_{EAA} < 20\%$$

$$H\alpha_2 : \mu_{AUAD} < 40\%$$

$$H\alpha_3 : \mu_{ARAM} \geq 3$$

$H\alpha 1$: It determines that there are benefits and gains of using the regression testing approach to find defects in the SPL architecture.

$$H\alpha 1_1 : \mu_{ND} < 20\%$$

$H\alpha 2$: It determines that there are gains of using the regression testing approach in order to classify the existing test cases.

$$H\alpha 2_1 : \mu_{NTCC} < 40\%$$

7.3.3 Variables Selection

In the variables selection step, independent and dependent variables are chosen. The *independent variables* are those variables that we can control and change in the experiment. In this study, the independent variable is the code in which the experiment will be performed. The *dependent variables* are mostly not directly measurable and we have to measure it via an indirect measure instead, in its turn, must be carefully validated, because it affects the result of the experiment. The defined dependent variables addressed by this study are: (a) understandability, usability, effectiveness, completeness and applicability of the approach; (b) the number of defects found; and, (c) the number of test cases correctly classified.

7.3.4 Selection of Subjects

All of the subjects of this study have a post-graduation course in the software testing area, being seven M.Sc. students and specialists in software testing and one M.Sc. All eight were selected by convenience sampling, which means that the nearest and most convenient persons are selected as subjects (Wohlin *et al.*, 2000).

7.3.5 Experiment Design

An experiment consists of a series of tests of the treatments, these tests must be carefully planned and designed. In order to design an experiment, the hypothesis should be analyzed to see which statistical analysis we have to perform to reject the null hypothesis. During the design, it is important to determine how many tests the experiment shall have to make sure that the effect of the treatment is visible (Wohlin *et al.*, 2000).

The general design principles are randomization, blocking and balancing, and most experiments designs use some combination of these.

- The **randomization** applies on the allocation of the objects, subjects and in which order the tests are performed. Since we have only one factor (Regression approach) and one treatment, no randomization is required.
- The **blocking** is used to eliminate the undesired effect in the study and therefore the effects between the blocks are not studied. Since this experiment considers only one factor this concept is not applied.
- The **balancing** concerns to the number of subjects per treatment, since the experiment considers only one treatment, the experiment is already balanced.

7.3.6 Instrumentation

In this study, the Regression testing approach documentation will be available for the subjects in order to execute the proposed activities, steps and available tools. A subject training will be conducted with the purpose to provide the basis for the approach use. This training will be divided into two steps: (i) Concepts related to software product lines, variability and software testing; (ii) and Regression testing approach flows, activities, tools and steps.

Due to time and resource limitation, this experimental study will be performed with a set of classes simulating a SPL architecture. It has two versions of a bank system which manages accounts, saving accounts, customers and companies. The first version (V1) was developed with eighteen (18) classes and one interface, and fifty-eight (58) integration test cases used to test the conformance of the system against its specification. A second version of it, was developed with new functionalities (simulating a evolution) and a set of seven faults seeded. This new version (V2), is composed by twenty-four classes and three interfaces. These changes aim to evaluate the regression testing approach in both scenarios during an evolution and correction.

It is important to highlight that these injected faults were inserted based on four sources: (i) McGregor’s SPL fault model (McGregor, 2008) where he summarizes the most common faults found in SPL projects; (ii) based on the mapping study previously performed; (iii) the experimenter knowledge in the application domain; (iv) the most common Java development faults extracted from the internet.

Firstly, both code versions, a set of change requests (three), as well as, a set of previous designed integration test cases were provided, in order to the subjects validate the approach considering the correction scenario. The subjects need to apply the approach aiming to find the faults previously seeded, as well as classify the integration test cases. The need also to apply all approach steps and answer the questionnaire. It is important to reinforce that the steps related to graph generation and graph comparison are optional in the approach, but the subjects were asked to use them at least one time. After report this first result, the class diagrams (from both versions) were provided in order to characterize the evolution scenario. In this context, the subjects should evaluate the specification changes, correctly classify the existing integration tests and create new test cases. The test cases should be designed obeying the same coverage criteria used in the previous designed integration test cases. Figure 7.2 summarize all scenarios.

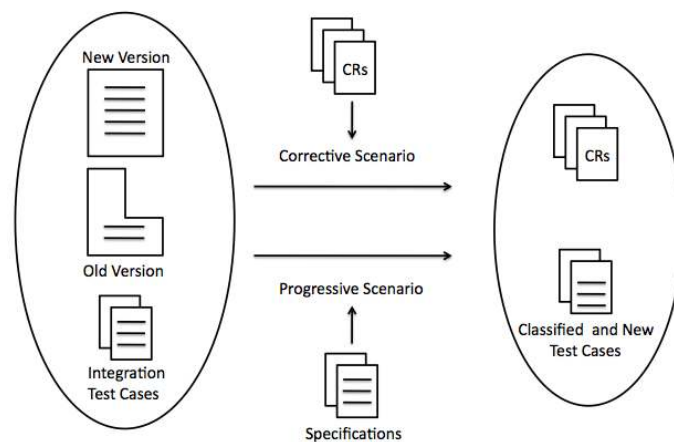


Figure 7.2 Experiment Scenarios.

Before performing the experiment, two pilot projects were conducted with the same structure defined in this planning (Section 7.3). The first pilot was performed by the author of this dissertation, who knows how to use the proposed approach. This pilot aims to detect problems and calibrate the experiment before its real execution. An issue regarding to how the approach deals with specification changes (evolution) was detected during this first pilot, and in order to solve it, a new step was added to address this

problem.

The second pilot was performed by a single subject, who has a certain experience in industrial projects performing test execution and design to regression, integration and exploratory testing. Some problems as code faults (not injected purposely) and absence of new-structural test cases were detected. Modifications on both code versions (new and old) were performed in order to solve these issues. A problem with the background questionnaire, some important questions used to extract the subjects profile were absent, was also detected and was solved adding a new question. During this pilot three new threats were discovered, the code size, the provided CRs and the injected faults, that will be described in threats section.

The results of the experiment will be collected using measurement instruments. Thus, it will be prepared time-sheets to collect the time spent in each activity and step. Furthermore, all subjects will receive a questionnaire (QT1) to evaluate their educational background, participation in software development projects, experience in testing and reuse. In addition, the subjects will receive a second questionnaire (QT2) for the evaluation of subject's satisfaction and difficulties using the proposed approach.

7.3.7 Validity Evaluation

It is important to consider the question of validity already in the planning phase in order to plan for adequate validity of the experiment results. Adequate validity refers to that the results should be valid for the population of interest, firstly, the results should be valid for the population from which the sample is drawn. Secondly, if possible, generalize the results to a broader population.

There are different classification schemes for different types of threats to the validity of an experiment. This experiment adopted the classification proposed by Cook and Campbell (1979) where four types of threats are presented. They are following described.

Conclusion validity: Threats for the conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome.

- *Experience of subjects:* Subjects without experience in regression testing (selection techniques, concepts and so on) also can affect this validity, since it is harder for them to understand the approach. To mitigate the lack of experience, a training in SPL and regression testing will be provided.

- *Experience in Java Development:* Subjects with low experience in software development (using Java) can affect this validity, since it is hard to understand the code and its peculiarities. To mitigate this lack of experience, the versions specifications were provided and we choose a small and common domain (Bank System).
- *Measurement reliability:* Once the measurement is not adequately, it can bring us no reliable data. Aiming to mitigate this threat, it will be validated with RiSE ¹ members.
- *Fishing:* Searching or fishing for specific results is a threat since the analyses are no long independent, and the researchers may influence the results by looking for a specific outcome.

Internal validity: Threats to internal validity are influences that can affect the independent variable with respect to causality, without the researcher's knowledge (Wohlin *et al.*, 2000). The following threats to internal validity are considered:

- *Maturation:* This is the effect that subjects react differently as time passes. Some subjects can be affected negatively (tired or bored) during the experiment, and their performance may be below normal. In order to mitigate this boredom, a familiar domain and a small code version was provided.
- *Instrumentation:* This is the effect caused by the artifacts used for experiment execution, such as data collection forms, code, seeded errors etc. If these are badly designed, the experiment is affected negatively. Two pilot projects were performed in order to have the more suitable experiment scenario.
- *Gained Experience:* It is the effect caused by the experiment execution order, in our case, the corrective scenario was performed before the progressive scenario. The subject gained a certain experience executing the first scenario, reducing the time needed to perform the second scenario. Two groups of subjects need to be used, one for each scenario.
- *Selection:* There were no volunteers in participating in the experiment. Thus, the selected group is more representative for the whole population (since volunteers are generally more motivated and may influence the results).

¹www.rise.com.br

External validity: Threats to external validity are conditions that limit our ability to generalize the results of our experiment to industrial practice.

- *Generalization of subjects:* The study will be conducted with M.Sc. students and one M.Sc. which has knowledge about software testing. Thus, the subjects will not be selected from a general population. In this case, if these subjects succeed using the approach, we cannot conclude that a practitioner testing engineer would use it successfully too. On the other hand, negative conclusions have external validity, i.e., if the subjects fail in using the approach, then this is strong evidence that a practitioner testing engineer would fail too.
- *Generalization of scope:* The experiment will be conducted on a defined time, which could affect the experiment results. The code will be defined according to this schedule to guarantee the complete execution of the approach. Thus, this scenario has a toy size that will limit the generalization. However, negative results in this scope is a strong evidence that in a bigger scope would fail too.

Construct validity: refers to the extent to which the experiment setting actually reflects the construct under study.

- *Mono-Operation Bias:* Since the experiment includes a single treatment, it may under-represent the construct, and thus not give the full picture of the theory.
- *Experimenter Expectancies:* Surely the experimenter expectancies may bias the results, and for that reason, all formal definition and planning of the experiment is being carefully designed beforehand, and reviewed by other RiSE members (performing other experiments) and advisors.

7.4 Operation

The operation phase of an experiment consists of three steps: **preparation** where subjects are chosen and instrumentation are prepared, **execution** where the subjects perform their tasks according to different treatments and data is collected, and **data validation** where the collected data is validated.

7.4.1 Preparation

The subjects were seven M.Sc. students, all of them specialists in software testing area, and one M.Sc. also in the testing area. All of them from RiSE Labs, and representing a

non-random subset from the universe of subjects. The subjects were informed that we would like to investigate the outcome of the approach execution. However, they were not conscious of what aspects we intended to study, i.e., they were not aware of the hypotheses stated. Before the experiment can be executed, all experiment instruments must be prepared and ready. Thus, all instrumentation defined in Section 7.3.6 were provided.

7.4.2 Execution

The experiment was conducted during the first semester of 2010, from February to March. Initially, the subjects were trained in several aspects of SPL, control flow graphs, Junit, EclEmma plugin, JDiff tool and in the applied approach (February), and after, they performed the regression testing approach in the code provided. Most of the students had participated in industrial projects. However, the subjects had low or none industrial experience in reuse activities, such as component development and SPL engineering. On the other hand, all of the subjects are members of the RiSE Labs, and their research area involve these aspects, which give them theoretical knowledge. Regarding to software testing, all of them have a post-graduation in it, and medium industrial experiences. Despite the experience reported regarding to regression testing, they have no or low experience in control flow graph analysis and most of them never used a test selection technique. Table 7.1 summarizes the subject's profile.

Table 7.1 Subject's Profile.

Subject ID	Years since graduation	Testing course during studies	Experience in Java	Exp. in software development	Exp. in soft. testing	Exp. in Regression Testing	Test selection technique	Exp. in CFG analysis	Exp. in SPL development
1	4	Other	High	Academic(2years)	Industry(2years)	Academic(Low) Industry(High 2years)	None	Medium	Academic(1year)
2	3	Graduation Specialization Other	Low	Industry(3years)	Academic(1year) Industry(2years)	Academic(Medium 1year) Industry(Medium 2years)	None	Low	None
3	3	Other	Medium	Academic(1year) Industry(1years)	Academic(1year)	None	None	None	Academic(1year)
4	5	Graduation Specialization Other	High	Industry(5years) Academic(0.5year)	Industry(1year)	Academic(High 5years) Industry(High 7years)	None	Low	Academic(1year)
5	3	Specialization Other	Medium	Industry(1year) Academic(4years)	Industry(2years) Academic(3years)	Academic(None) Industry(None)	None	Medium	Academic(2years)
6	2	Specialization Other	High	Alone(5years) Industry(1year) Academic(2years)	Industry(1year) Academic(1year)	Academic(High 0.5year) Industry(High 1year)	Low	None	Academic(1year)
7	2	Specialization	High	Industry(2years)	Industry(1year)	Academic(None) Industry(None)	None	None	Academic(1year)
8	3	Specialization	High	Alone(4years) Industry(2years) Academic(2years)	Academic(2years) Industry(2years)	Academic(None) Industry(None)	None	None	Academic(1year)

The subjects were suggested to perform the experiment activities on their free time, using the place more convenient for them. Furthermore, we needed only to setup the environment on Eclipse, Junit and EclEmma for the approach execution.

7.4.3 Data Validation

In this phase, all data are checked in order to verify if the data are reasonable and that it has been collected correctly. This deals with aspects such as if the participants have understood the forms and therefore filled them out correctly (Wohlin *et al.*, 2000).

Data were collected from 8 students. By analysing the subjects reports, two problems were observed. Firstly, data from 2 subjects (ID 1,3 - see Table 7.1) will not be considered when evaluating test classification (Test selection and design step), since they did not participate in the experiment seriously, did not understand the questionnaire or even did not answer it correctly. For this reason, these two subjects were excluded from test classification evaluation.

Secondly, a problem was detected in subject (ID 4), he did not report the time used during the report step, for this reason this subject will not be evaluate during the step analysis.

Regarding to the graph generation and graph comparison steps both were considered optional in this experiment, since we are not interested in validate the tool, for this reason both steps will not be considered.

7.5 Analysis and Interpretation

After collecting experimental data in the operation phase, we are able to draw conclusions based on these data. The results obtained with the experimental study are presented.

7.5.1 Effort to Apply the Approach

This aspect was evaluated in two scenarios corrective and progressive. The corrective scenario spent 94.78 hours to be performed, the progressive scenario was executed in 56.06 hours. These numbers are concerning to total number of worked hours of the members in each step. In the next sections, each of these scenarios will be analyzed.

7.5.1.1 Corrective Scenario

In this scenario, the subjects were asked to apply the approach after a modification due to a corrective action performed in the code. In this section, all collected data were analyzed, as well as data validation was carried out in order to identify outliers. Table 7.2 shows the brute data collected after the experiment execution, where “not considered (NC)” means that the step was not report correctly and “not reported (NR)” means that none time was reported by the subject.

Table 7.2 Approach execution effort (minutes) considering corrective scenario.

Subjects/Steps	ID1	ID2	ID3	ID4	ID5	ID6	ID7	ID8	Total
Planning	120	26	90	5	90	255	180	120	886
Analyzes	60	10	90	60	180	60	60	60	580
Graph Generation	NR	20	120	NR	NR	140	150	25	455
Graph Comparison	NR	30	15	NR	NR	NR	60	60	165
Textual Comparison	240	73	240	20	240	90	120	60	1083
Test Design and Selection	NC	55	NC	10	290	25	30	60	813
Instrumentation	30	20	110	5	30	110	120	30	455
Test Suite Composition	10	23	76	5	120	180	90	20	524
Test Case Prioritization	10	10	24	8	30	20	30	10	142
Test Execution	10	60	175	5	60	7	10	120	447
Reporting	50	30	160	NR	30	60	60	90	480

Before analyzing the collected data, some issues were observed. For example, Test Design and Selecion step needed to be submitted to a refinement (exclusion of two subjects (ID 1 and 3)), since the subjects did not report it correctly (Section 7.4.3). In additional, some steps (Graph Generation, Graph Comparison and Reporting) were not reported completely, whereas steps Graph Generation and Graph Comparison were defined as optional by the approach and some subjects did not report it, the Reporting was not reported by the subject ID 4. All, was taken into account during the assessment.

Data validation deals with identifying false data points based on execution of the experiment. Thus, we intend to identify outliers not only based on the experience execution, but instead looking at the results from execution in the form of collected data and taking into account, for example, descriptive statistics. This way, we are able to identify if people have participated seriously in the experiment.

In order to isolate the outliers of this experimental study, the same idea presented in (Almeida, 2007) was used and described in the next paragraphs. All steps were submitted to this outliers identification, in order to explain in details how it was performed, the first one was considered.

As it can be seen in Figure 7.3, subjects ID 4 and ID 6 presented values low (5) and high (255), respectively, when compared with other data points. Thus, these values could



Figure 7.3 Planning step distribution.

be considered as outliers. In order to analyze this aspect, a box plot graphic can be useful (Fenton and Pfleeger, 1998), since it is recommended to visualize the dispersion and skewness of samples. Box plots can be made in different ways (Wohlin *et al.*, 2000). In this dissertation, the approach defined by Fenton and Pfleeger (1998) was chosen. The main difference among the approaches is how to handle the whiskers. Fenton & Pfleeger proposed to use a value, which is the length of the box, multiplied by 1.5 and added or subtracted from the upper and lower quartiles respectively.

The middle bar in the box is the median. The lower quartile q_1 , is the 25% percentile (the median of the values that are less than median), and the upper quartile q_3 is the 75% percentile (the median of the values that are greater than median). The length of the box is $d = q_3 - q_1$.

The tails of the box represent the theoretical bound within all data points are likely to be found if the distribution is normal. The upper tail is $q_3 + 1.5d$ and the lower tail is $q_1 - 1.5d$. Figure 7.4 shows the box plot graphic with its information, considering the planning step.

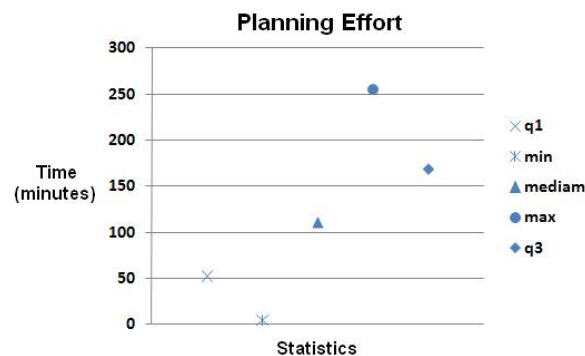
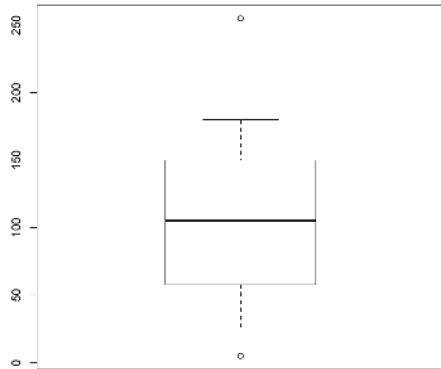
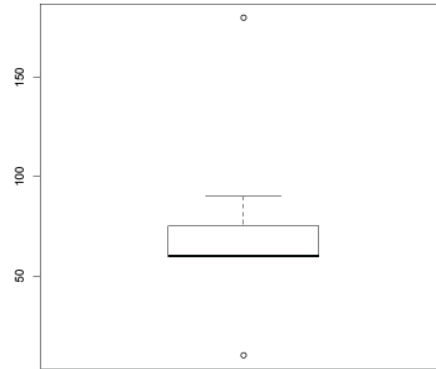


Figure 7.4 Box plot analysis.

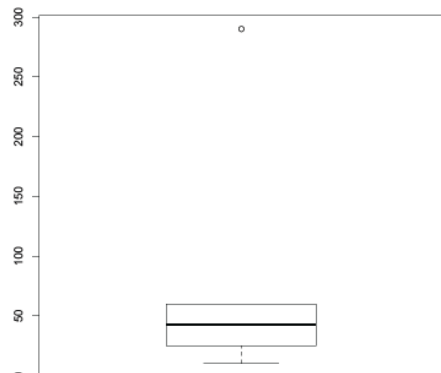
After this data validation some outliers were identified during the approach execution. All steps were analyzed, whereas we identified outliers in Planning, Analyzes, Test Design and Selection, Test Suite Composition, Execution and Reporting steps, as shown in Figures 7.5(a),7.5(b),7.5(c),7.5(d), 7.5(e) and 7.5(f). In Graph Generation, Graph Comparison, Textual Comparison, Instrumentation and Test Case Prioritization steps no outliers were identified. Only the steps where some outliers were identified are displayed.



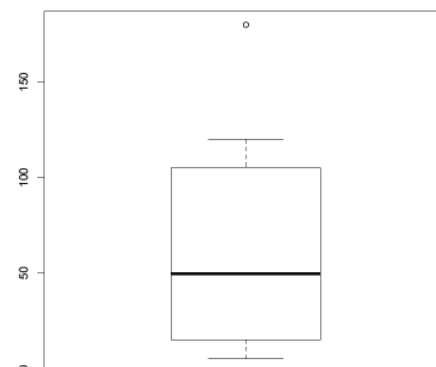
(a) Outliers (IDs 6, 4) from Planning step.



(b) Outliers (IDs 5, 2) from Analyzes step.



(c) Outlier (ID 5) from Test Design and Selection step.



(d) Outlier (ID 6) from Test Suite Composition step.

After performing the outliers analysis, we chose to leave all subjects identified as outliers and consider its times in the effort analysis, since the limited number of subjects. The effort to apply the approach is shown, in Tables 7.3(a) and 7.3(b). The first, shows the effort to apply each step, whereas the second shows only the steps that were completely and correctly reported.

The time spent during the planning step can be justified buy the fact that none of the subjects had performed it previously. Since it was their first time, they need some time to understand the test plan and collect all information in order to fill it. Beside to gather

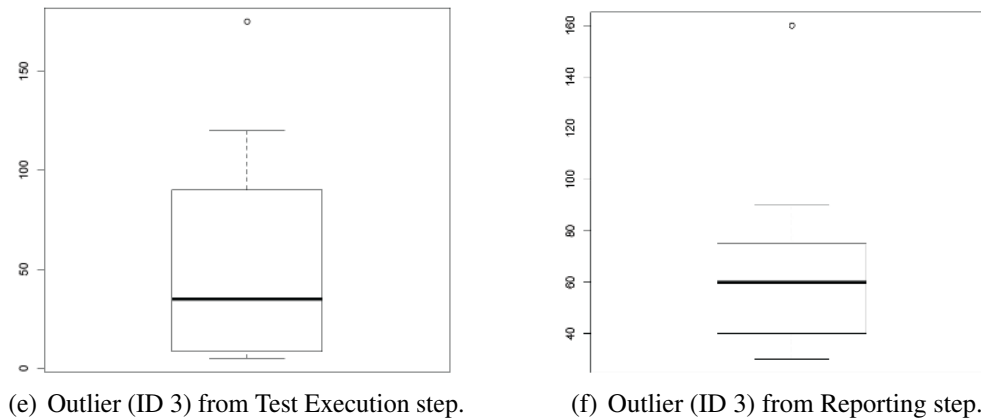


Figure 7.5 Outliers Analysis.

Table 7.3 Effort to apply the approach.

(a) Effort to apply the corrective scenario considering all steps.

Steps	EAA (%)
Planning	15.58
Analyzes	10.20
Graph Generation	8.00
Graph Comparison	2.90
Textual Comparison	19.04
Test Design and Selection	8.26
Instrumentation	8.00
Test Suite Composition	9.21
Test Case Prioritization	2.50
Test Execution	7.86
Reporting	8.44

(b) Effort to apply the corrective scenario considering some steps.

Steps	EAA (%)
Planning	21.52
Analyzes	14.08
Textual Comparison	26.30
Instrumentation	11.05
Test Suite Composition	12.72
Test Case Prioritization	3.44
Execution	10.85

information they should plan the test cycle considering the constraints and information provided for the instrumenter.

Regarding the textual comparison step, the subjects need to compare both code versions, as well as, understand how the change impacts on the domain application rules. They should identify portions of the code in order to discover critical paths, that will be further exercised by the created and selected test cases.

Since the graph generation and comparison steps were considered as optional in our approach, some subjects have the need to use. Some of them figure out the importance of the need of such a tool in order to identify and catch language behaviors. It's important to highlight that most of the subjects complained about boredom when executing these steps.

In this dissertation, we adopted the data presented in Table 7.3(a), which **rejects** the null hypothesis since none of the steps had effort above to 20%, as previously established in Section 7.3.2. We chose this data set since we can have a better understanding of how all

steps in the approach behave. However, if we consider the data set presented in Table 7.3(b), where was identified an absence of report or a wrong report in some steps, the null hypothesis is not rejected. As we can see in Table 7.3(b), the Planning (21.52%) and Textual Comparison (26.30%) steps exceed the established metric.

7.5.1.2 Progressive Scenario

In this scenario all subjects were asked to apply the regression approach after a modification in the architecture specification due to an evolution. In this section, all collected data were analyzed and the same previous realized data validation was performed. Table 7.4 shows the general(brute) data collected after the experiment execution.

Table 7.4 Approach execution effort considering progressive scenario.

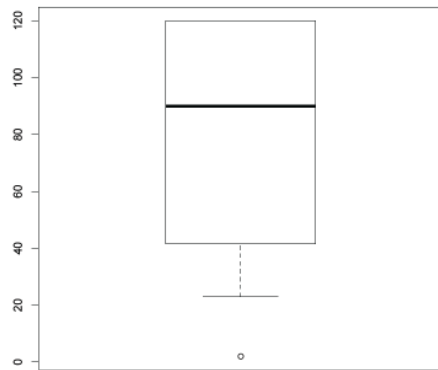
Subjects/Steps	ID1	ID2	ID3	ID4	ID5	ID6	ID7	ID8	Total
Planning	60	23	120	2	90	90	120	120	625
Analyzes	30	10	60	5	150	30	30	60	375
Specification Comparison	30	22	53	20	80	60	90	30	385
Test Design and Selection	NC	65	NC	15	270	120	120	60	650
Instrumentation	30	25	98	5	30	60	60	30	338
Test Suite Composition	10	17	20	5	30	130	60	20	292
Test Case Prioritization	10	10	34	5	10	10	15	10	104
Execution	10	25	34	5	60	10	8	60	212
Reporting	50	25	98	NR	30	60	60	60	383

Some subjects were not considered during this evaluation. For example, subjects (ID 1 and 3) were excluded since they did not report correctly the output of Test Design and Selection step, as well as subject (ID 4) was removed from Reporting step since he did not report. This information was collected after an interview, where they (subjects) explain their questionnaire answers, as well as, during the experiment data analysis.

After this previous analysis, the collected data was submitted to the same data validation performed in corrective scenario, this second validation aims to identify possible outliers. The same method was used here, and to summarize, only the steps with outliers are presented. Figures 7.6(a), 7.6(b), 7.6(c), 7.6(d), 7.6(e), 7.6(f), 7.7 show the steps and its respective outliers.

As the previous scenario analysis, we chose to leave all subjects pointed as outliers since the limited number of subjects. Tables 7.5(a) and 7.5(b) show the effort to apply the steps, whereas the first shows the effort to apply each step, the second shows the effort to apply the progressive scenario considering some steps.

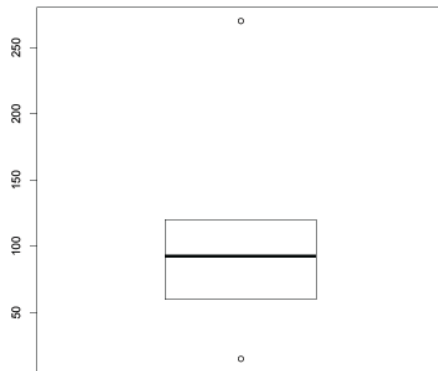
The same was observed for progressive scenario, if we do not consider the wrongly reported or incomplete steps, the null hypothesis is not rejected. As mentioned before,



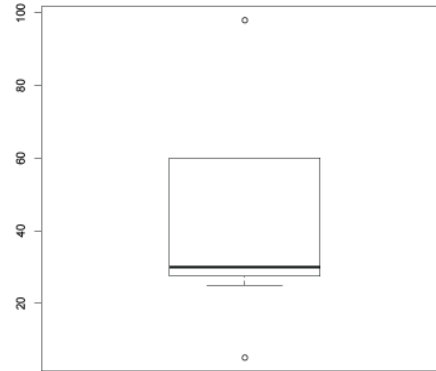
(a) Outliers (IDs 4) from Planning step.



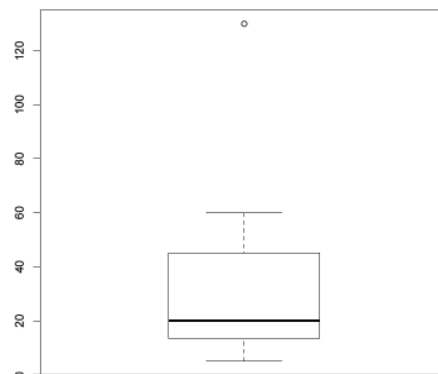
(b) Outliers (IDs 5) from Analyzes step.



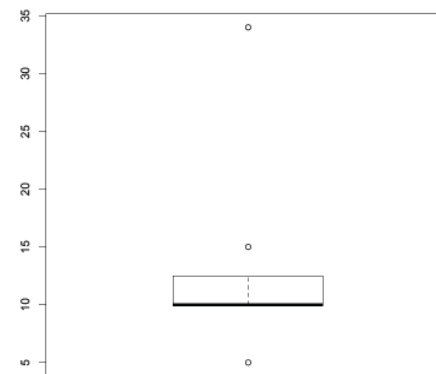
(c) Outlier (ID 5) from Test Design and Selection step.



(d) Outlier (ID 3 and 4) from Instrumentation step.



(e) Outlier (ID 6) from Test Suite Composition step.



(f) Outlier (ID 3, 4 and 7) from Test Case Prioritization step.

Figure 7.6 Outliers Analysis.

in this dissertation, we will consider all steps, so in this context, the null hypotheses is **rejected** since the most costly (in time) step did not exceed 20% ($H_0 : \mu_{EAA} \geq 20\%$).

We observed that the time spent to perform the progressive scenario was less than the

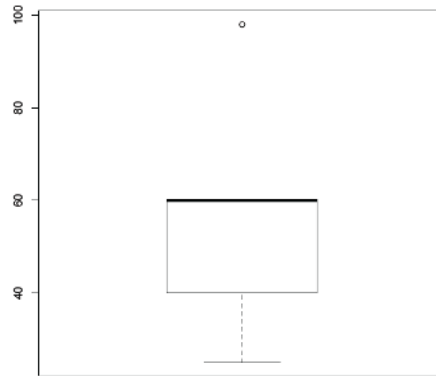


Figure 7.7 Outlier (ID 3) from Reporting step.

Table 7.5 Effort to apply the approach.

(a) Effort to apply the progressive scenario considering all steps.

Steps	EAA (%)
Planning	18.57
Analyzes	11.14
Specification Comparison	11.44
Test Design and Selection	19.32
Instrumentation	10.04
Test Suite Composition	8.68
Test Case Prioritization	3.09
Test Execution	6.30
Reporting	11.38

(b) Effort to apply the progressive scenario considering some steps.

Steps	EAA (%)
Planning	26.81
Analyzes	16.08
Specification Comparison	16.51
Instrumentation	14.50
Test Suite Composition	12.52
Test Case Prioritization	4.46
Test Execution	9.09

previous one. It can be explained since all subjects applied the corrective scenario first, thus when performing the progressive scenario they already had some expertise in the domain and code.

By analyzing the datasets, we can noticed that the time spent to perform the Test Design and Selection step in progressive scenario was less than in corrective scenario, it could be caused by the acquired experience or as reported by some subjects the fact that during the progressive scenario the number of retestable test cases were less than in corrective scenario.

7.5.2 Approach Understanding and Application Difficulties

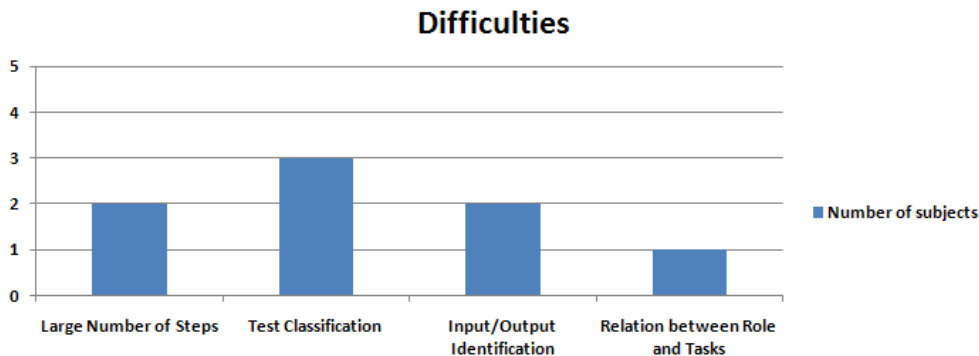
Analyzing subject's answers regarding to difficulties during approach execution, it was identified that 62.5% subjects had any kind of difficulty to understand the approach. Because of the understanding problem, all of them had also problems to apply the approach. The difficulties are summarized in Table 7.6.

Table 7.6 Difficulties to use the approach.

Difficulty	Number of subjects
Large number of steps	2
Test Classification	3
Input/Output Identification for each step	2
Relation between Role and Task	1

Two subjects (ID 1,5) claimed that the main problem to the understandability of the approach was the number of steps and tasks, that were too many and very complex, requiring a certain knowledge in both, development and testing areas. Another three subjects (ID 3,4,5) reported that one of the understandability issues was the lack of examples during test design and selection, more specifically to help the test classification task. The subjects (ID 3,6) stated that the input and output of each step were not clearly presented in the approach. At last, subject (ID 1) report his difficulty in understanding the relation between role and tasks, which tasks each role should perform.

Figure 7.8 shows the histogram with the distribution density of the found difficulties.

**Figure 7.8** Difficulties during approach execution.

The null hypothesis related to the percentage of subjects with any kind of difficulty in the process defines a percentage of more than 40% ($H_0: \mu_{AUAD} \geq 40\%$). Since we had 62.5% of the subjects with at least one difficulty, this null hypothesis was **not rejected**. However, in the same way as the previous hypothesis, this value for the null hypothesis was defined without any previous data.

7.5.2.1 Correlation Analysis

As we can observe, there is no correlations among the characteristics of subjects profile and the difficulties to understand the approach. Although subjects (ID 3,4,5) which have

no experience in applying test selection techniques, reported the absence of examples to help the test classification, other subjects with no experience did not indicate this problem.

7.5.3 Activities, Roles and Artifacts Missing

The goal with this question was collect more information about the regression testing approach missing steps. In this sense, the subjects were asked if there was any missing step, activities, roles and artifacts.

Analyzing the data, we could notice that none of the subjects identified any missing activities, roles, artifacts or steps. Since we had 0 (zero) identified as missing, the ($H_{03} : \mu_{ARAM} < 3$) null hypothesis is **rejected**.

7.5.3.1 Correlation Analysis

As we can see, all subjects have at least 1 year of experience in software testing, have some kind of course in the testing area and some of them have been worked with regression testing. It can serve as clue to indicate that the approach is complete and well structured.

7.5.4 Number of Defects

By analyzing the faults found during the approach application, the following dataset (Table 7.7) was structured. As it can be seen, all injected faults were identified.

Table 7.7 Defects per subjects.

Subject ID	Fault1	Fault2	Fault3	Fault4	Fault5	Fault6	Fault7
ID1	x	x					x
ID2	x	x	x	x	x	x	
ID3	x	x					
ID4	x			x	x		
ID5	x	x		x	x		
ID6	x	x	x				x
ID7	x	x	x				x
ID8	x	x	x				x

During the analysis, we could noticed that the subjects did not report only the root cause of the issue, they report faults in different architecture layers. For example, a fault was injected in a lower layer and it is propagated to layers above, nevertheless the subjects report the faults in all layers. In additional, not purposely injected faults and identification errors were identified. These aspects should be considered in further experiments and something should be done to avoid it.

It is important to highlight that only the root cause was considered to evaluate this aspect, as well as, the not purposely injected faults and identification faults were not considered in this evaluation. Faults wrongly reported in the questionnaire were also not considered. All of them will serve as lessons learned to avoid in future experiments.

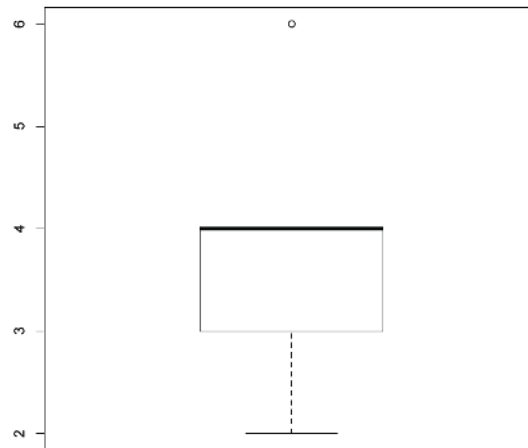


Figure 7.9 Boxplot Analysis.

In order to identify any outlier during this aspect evaluation, the same method applied in Section 7.5.1 was used here. By analyzing Figure 7.9, we could see that subject (ID 2) was pointed as outlier. He was not removed from the analysis, since the limited amount of subjects. An interpretation of this performance can be viewed in correlation analysis Section (7.5.4.1).

Figure 7.10 shows the number of subjects per fault found.

Considering these data, all injected faults were found by at least one subject, since we had 0 faults no identified, the $H_{11} : \mu_{ND} \geq 20\%$ null hypothesis is **rejected**.

7.5.4.1 Correlation Analysis

By observing the subject with the best results in this aspect, we could see that all of them have more than 2 years of experience in software development. It can indicate that a high experience in development is required by the person which will apply the approach. The subject (ID 2) had the best results in fault detection, the unique factor that was observed and could justify its success is the fact that he was the first one to deliver the experiment results. It could influence, since the experiment was performed with less gaps (stops).

Regarding to the number of subjects that found a specific fault (see Figure 7.10), we can notice that the faults (1,2 and 7) were the most found during this experiment, it could

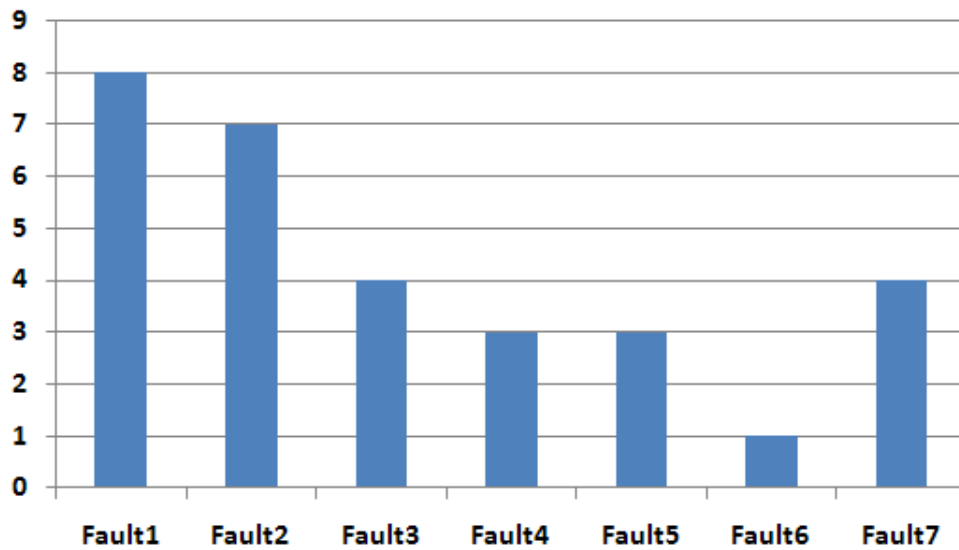


Figure 7.10 Number of Subjects vs Faults.

be explained by the fact that the CRs provided by the experimenter described these faults. It can indicate that the CRs help the approach execution. In additional, no correlation was found regarding to the type of fault.

Although the rejection of the hypothesis and the approach has been proven efficient in fault detection, it can not be considered as absolute truth. A more complex experiment, with real SPL architectures, faults and tests, needs to be performed.

7.5.5 Number of Tests Correctly Classified

The goal of this aspect was to evaluate how the approach is aiding the subjects during the test classification step. In this direction, the subjects were asked to classify the test cases in five categories (obsolete, reusable, retestable, new-structural and new-specification). Unfortunately, some subjects (ID 1, 3) were excluded of this evaluation since they did not report anything or wrongly report the results.

Table 7.8 summarizes the number of test correctly classified by subjects. Where “not reported (NR)” means that the subjects report some test cases but not correctly, and “none” indicates that no test cases were reported.

A set of 58 test cases were provided to the subjects, among these tests, 3 obsolete, 14 retestable, 41 reusable, 2 new-structural and 14 new-specification. Most of the subjects complained about the absence of examples regarding each kind of test. It can explain the bad results in this aspect.

Table 7.8 Number of tests correctly classified.

Subject ID	Obsolete	Reusable	Retestable	New-Structural	New-Specification	NTCC (%)
ID2	NR	13	3	NR	14	40.54
ID4	1	30	6	NR	NR	50
ID5	none	2	9	none	3	18.91
ID6	1	3	5	none	14	31.08
ID7	1	3	5	none	14	31.08
ID8	NR	7	NR	NR	NR	9.45

By analyzing the results, we noticed that the test classification description should be improved. Nevertheless, two subjects achieve more than 40% of correctly classified test cases. For this reason, we consider the null hypotheses ($H_{21} : \mu_{NTCC} \geq 40\%$) was **rejected**.

To improve the test classification task, guidelines describing how to identify each category of test, should be provided.

7.5.5.1 Correlation Analysis

We notice that subjects with high experience in software development, had better results. It can indicate that experience in development can help the process of test classification.

7.6 Lessons Learned

After concluding the experimental study, some aspects should be considered in order to repeat the experiment, since there were limitations in this first execution.

Project Context. The architecture (code) context was the main issue regarding the regression approach experiment execution. One of the problems was that the architecture was not a real SPL architecture composed by components and modules. In addition, the faults were artificially injected in the code. Thus, the approach needs to be evaluated in a more real context.

Training. The regression approach presentation was satisfactory, but for the subjects, it seems to be too much information at once. Therefore, it is interesting to have some support materials for the experiment, with examples on how to use the approach in different scenarios (corrective and progressive).

Motivation. As the project was not short and most of the subjects complained about boredom, it was difficult to keep the subject's motivation during all the execution. Thus, this aspect should be analyzed in order to try to control it. A possible solution can be to define some checkpoints during the approach application or split in two individual

experiments (corrective and progressive).

Data Collection. Some subjects were influenced due to the order of some questions in the questionnaire. For example, some of them were confused because the Q7 (faults question) was before the Q8 (test classification). Regarding to the background questionnaire, some questions (Q4 and Q13) could be better formulated to avoid subjective answers.

7.7 Chapter Summary

This Chapter presented the definition, planning, operation, analysis and interpretation of the experimental study that evaluated the viability of the Regression testing approach. The experiment, analyzed, the process understandability, effectiveness and completeness, how it helps the subjects during test classification and defects search tasks, as well as, try to identify any miss activity, step or role. Even with the reduced number of subjects (8), and a not very appropriate context, we could identify some directions for improvements, specially regarding understandability, based on the subjects difficulties. However, two aspects should be considered: the study's repetition in different contexts and studies based on observation in order to identify other problems and points for improvements.

The next chapter will present the conclusions of this work, its main contribution and directions for future works.

“The art of living consists in turning life into an artwork.”

Mahatma Gandhi



Conclusion

The software industry is constantly searching for new ways to achieve productivity gains, reduce development costs, improve time-to-market, increase software quality (Linden *et al.*, 2007) and turn software development less handcrafted. Organizations adopt software product lines approaches aiming to accomplish these goals with the purpose of maintaining competitive in the current business environment.

In this context, as in single-system development, testing is essential (Kauppinen, 2003) to uncover defects (Pohl and Metzger, 2006; Reuys *et al.*, 2006). From an industry point of view, with the growing SPL adoption by companies, more efficient and effective testing methods and techniques for SPL are needed, since the currently available techniques, strategies and methods make testing a very challenging process (Kolb and Muthig, 2003).

Testing in SPL aims to examine core assets - i.e. the architecture of the product line, which comprises the common parts of the products - individual products - derived from the common “architecture” prior established - and the interaction among them (McGregor, 2001b). As software architectures are becoming the central part during the development of quality systems, being the first model and base to guide the implementation (Muccini *et al.*, 2006) and provide a promising way to deal with large systems (Harrold, 2000), it is extremely important to consider testing since design level. For this reason, a software product line regression testing approach was developed (Chapter 6), as being a technique applied during integration level (Chapter 5), in order to be confident that SPL architectures still working after a evolution or corrective modification. The integration test level was chosen, due to the importance of the SPL architecture, considered the SPL main asset.

In addition, it can be useful to make confidence in the correctness of the software, increasing its reliability Wahl (1999), as well as, identifying errors that were missed, after applying traditional code-level testing Muccini *et al.* (2006). Since it is a worthwhile ongoing investment, every company should be using regression testing to produce quality

products Long (1993).

In this Chapter, the conclusion of this work is presented. The Chapter remainder is organized as follows. The research contributions are highlighted in Section 8.1. The related work to regression testing approach is described in Section 8.2 and future work concerning to the defined approach are listed in Section 8.3. Academic contributions are listed in Section 8.4 and the concluding remarks of this dissertation are described in Section 8.5.

8.1 Research Contributions

The main contributions of this work can be split into the following aspects: *(i)* the definition of a mapping study in order to understand and characterize the state-of-the-art regarding to SPL and testing; *(ii)* the definition of an approach to integration test the modules which composes SPL architectures, in order to check the conformance of the architectures with its specification; *(iii)* the definition of an SPL architecture regression testing approach; *(iv)* the execution of an experimental study which evaluated the regression testing approach. These contributions are further described next.

- **Mapping Study on Software Product Line Testing.** Through this mapping study forty-five studies were selected and analyzed according to the aspects related to the mapping studie's question: *how the existing testing approaches deal in the context of SPL?*. This question was split in nine sub-questions each of them related with SPL and testing concepts. The analysis results could identify gaps and points for future research in testing area, as how the variability testing should be handled over the SPL life-cycle, the metrics that should be used in SPL testing processes and the theme proposed by this dissertation.
- **Integration Testing Approach.** Given a software architecture (SA) description, conformance testing has been used to detect conformance errors between the SA specification and its implementation. The SA specification has been used as a reference model to which the source code should conform (Muccini *et al.*, 2004). In order to check this conformance, an integration testing approach was developed.
- **SPL Architecture Regression Testing Approach.** After the occurrence of a modification due to an evolution or correction, the SPL architectures need to be retested to ensure that no new errors were inserted and that the new architecture version still working as expected. For this reason, a SPL architecture regression testing

approach was developed, including its artifacts, roles, steps and activities. It was applied during integration testing level.

- **Experimental Study.** An experimental study was performed in an academic environment in order to evaluate the proposed approach. This initial validation of the regression testing approach helped in the improvement of the approach, since the findings and observed points suggested some modifications in the activities, steps to improve the understandability and facilitate further applications.

8.2 Related Work

Some SPL testing techniques and methods were identified through the mapping study, described in Chapter 4. However, among the approach included in this study, none of them presented any kind of systematic and formal approach definition for regression testing SPL architectures, and no study gives evidence on regression testing practices applied to SPL, only few comments.

Several researchers highlighted the importance of regression testing in the context of SPL, but not systematically. McGregor in (McGregor, 2001b) reports that during products derivation, the assets are often modified to fit the products needs. The modified portion of those assets are tested using regression testing and they are exercised using: (i) existing functional tests, if the specification is not changed; (ii) new created functional tests, if the specification is changed; and (iii) structural tests created to reflect the new code. He also highlights the importance of test selection techniques and test automation.

According to Kolb (Kolb, 2003), the major problems in testing product lines are the large number of variations, redundant work, the interplay between generic components and product-specific components, and regression testing. In (Jin-hua *et al.*, 2008) reports that during the application test, regression test are needed to be performed on the realized design patterns with test data from the domain test, in addition to the concrete classes. Besides, when components or related component cluster are changed, regression test is obliged to perform on the application architecture, in order to ensure the application architecture in conformance with its specification.

While the (McGregor, 2001b) and (Jin-hua *et al.*, 2008) are describing regression testing in system test level, the next paragraphs describes some studies that comprises the regression testing applied in integration testing level, which is the same level reported by this dissertation.

In Muccini *et al.* (2004), the authors emphasize that with the advent and use of

software specifications, source code no longer has to be the single source for selecting test cases. Their particular interest has been devoted to specification-based conformance testing. The main goal of their work is to review and extend their previous work on Software Architecture (SA)-based conformance testing, to provide a systematic way to use an SA for code testing. They present a conformance testing approach, establishing a set of steps in order to test a C2 style architecture. This work also presents a case study, where the approach is applied in the elevator system's architecture.

In Muccini *et al.* (2006), the authors explore how regression testing can be systematically applied at the software architecture level in order to reduce the cost of retesting modified systems, and also to assess the regression testability of the evolved system. This approach addresses two goals: (i) Test conformance of a modified implementation P' to the initial SA and (ii) test conformance of an evolved software architecture. To achieve these goals, a set of steps and tools were used.

In our work we did not implement or restricted us to any test selection technique to select test cases. Instead, we studied some approaches and their characteristics in order to figure out how systematically perform regression testing in SPL architectures taking advantage of their similarities. The main difference between our work and Henry's work Muccini *et al.* (2004) is that we are considering conformance testing in SPL context, taking into consideration the existing variability (the test cases are capable to represent the variation points and its variants) and not being restricted to any architectural style. Regarding the second study, besides he is not considering SPL context, he did not treat test prioritization on his work. In addition, our approach defines a systematic way to perform regression testing describing some artifacts, roles, activities and steps.

8.3 Future Work

Due to the time constraints imposed on the master degree, this work can be seen as an initial climbing towards a process for testing product lines, and interesting directions remain to improve what was started here and new routes can be explored in the future. Thus, the following issues should be investigated as future work:

- **Metrics.** This dissertation proposed some metrics to evaluate regression testing approach use in the experimental study, however these metrics were never used before, this way, they need to be refined and reproduced. This metric set could be also increased by several other metrics to measure the approach application in SPL context.

- **Integration Testing Approach Industrial Evaluation.** Since a integration testing approach was also proposed and only an simple application(example) was performed to evaluate this approach, it is necessary to perform a more elaborated experimental study, applying it in industrial projects.
- **Application of the Regression Approach in an industrial context** As the regression testing approach was evaluated in academic conditions, it is necessary to evaluate it in a more elaborated context with real SPL architectures.
- **Test Classification Guideline.** A very nice improvement for Regression approach would be to have guidelines on how to classify the test cases. Those guidelines would have examples explaining how to identify all types of tests.
- **Tool Support.** The need of a tool that based on the code and change analysis, can identify and select the test cases that need to be executed in the architecture new version.
- **Test Prioritization.** A nice improvement would be to study the best way to prioritize test cases based on the most common issues found in SPL projects.

8.4 Academic Contributions

- Software Product Lines Testing: A Systematic Mapping Study. Under evaluation (2nd round) in Information and Software Technology Journal.
- A Regression Testing Approach for Software Product Lines Architectures. Under evaluation in 4th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2010)

8.5 Concluding Remarks

Software reuse is a key aspect for organizations interested in achieving improvements in productivity, quality and costs reduction. Software product lines, as a software reuse approach, has proven its benefits in different industrial environments. Academic research in the software product line is also very rich and a diversity of studies are being conducted in different topics.

In this context, this work presented the SPL architecture regression testing approach, which aims to verify if the new version of the architecture still working properly, after

an evolution or correction modifications. It is applied at integration testing level (for both CAD and PD) focusing on increasing reuse, productivity and reducing testing cost. This approach can be seen as a systematic way to perform regression testing after a modification in SPL architectures, through a well-defined sequence of activities, steps, inputs, outputs, and guidelines.

Additionally, the approach was evaluated in a academic context, through an experimental study on the Bank system domain. According to the data collected and analyzed in the experimental study, the approach presents indications of its viability. We believe this dissertation is one more step to the maturation of the regression testing approach in software product line architectures.

References

- (2009). Clover - Code Coverage Analysis. <http://www.atlassian.com/software/clover/>. 5.6.1
- (2009). Coverlipse. <http://coverlipse.sourceforge.net/>. 5.6.1
- (2009). EclEmma - Java Code Coverage for Eclipse. <http://www.eclEmma.org/>. 5.6.1, 5.7
- (2009). Emma - a free Java code coverage tool. <http://emma.sourceforge.net/>. 5.6.1
- (2009). JUnit Framework. <http://junit.sourceforge.net/>. 5.7
- Abran, A., Bourque, P., Dupuis, R., Moore, J. W., and Tripp, L. L. (2004). *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, Piscataway, NJ, USA, 2004 version edition. 3.3, 3.4.2, 3.4.4, 3.5, 6.3.1
- Afzal, W., Torkar, R., and Feldt, R. (2008). A systematic mapping study on non-functional search-based software testing. In *SEKE'08: Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, pages 488–493, Redwood City, California, USA. 4.2
- Afzal, W., Torkar, R., and Feldt, R. (2009). A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, **51**(6), 957–976. 4.2
- Al-Dallal, J. and Sorenson, P. G. (2008). Testing software assets of framework-based product families during application engineering stage. *Journal of Software*, **3**(5), 11–25. 4.5.2.6, 4.4, C
- Almeida, E. S., Alvaro, A., Lucrédio, D., Garcia, V. C., and Meira, S. R. L. (2004). Rise project: Towards a robust framework for software reuse. In *IRI'04: International Conference on Information Reuse and Integration*, pages 48–53, Las Vegas, NV, USA. 1.3.1
- Almeida, E. S., Alvaro, A., Lucrédio, D., Garcia, V. C., and Meira, S. R. L. (2005). A survey on software reuse processes. In *IRI'05: International Conference on Information Reuse and Integration*, pages 66–71, Las Vegas, USA. 1.3.1
- Almeida, E. S. D. (2007). *RiDE: The RiSE Process for Domain Engineering*. Ph.d thesis, Universidade Federal de Pernambuco, Recife, Pernambuco, Brazil. 2.1, 7.5.1.1

- Alvaro, A. (2009). *Software Component Certification: A Component Quality Model*. Ph.D. thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 1.3.1
- Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press, 1st edition. 3.2, 3.3, 3.4.1, 3.4.2, 3.5, 3.6, 3.7, 4.3.3
- Apiwattanapong, T., Orso, A., and Harrold, M. J. (2007). JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, **14**(1), 3–36. (document), 6.6.1.3, 6.10, 6.11
- Bailey, J., Budgen, D., Turner, M., Kitchenham, B., Brereton, P., and Linkman, S. (2007). Evidence relating to object-oriented software design: A survey. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 482–484, Washington, DC, USA. 4.2
- Balbino, M; Almeida, E. S. M. S. R. L. (2009). A software component quality model: A preliminary evaluation. In *ESELAW'09: Proceedings of the VI Experimental Software Engineering Latin American Workshop*, Sao Carlos, Brazil. 1.3.1
- Baresi, L. and Pezzè, M. (2006). An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, **148**(1), 89–111. 3.1
- Basili, V., Caldiera, G., and Rombach, H. (1994). The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, **1**, 528–532. 7.2
- Basili, V. R., Selby, R., and Hutchens, D. (1986). Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, **12**(7), 733–743. 7.2
- Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J.-M. (1999). Pulse: a methodology to develop software product lines. In *SSR '99: Proceedings of the 1999 symposium on Software reusability*, pages 122–131, New York, NY, USA. ACM. 6.4
- Beatriz Pérez Lamanha, Macario Polo Usaola, M. P. (2009). Towards an automated testing framework to manage variability using the uml testing profile. In *AST'09: Proceedings of the Workshop on Automation of Software Test (ICSE)*, pages 10–17, Vancouver, Canada. 4.4, C
- Beizer, B. (1990). *Software testing techniques*. International Thomson Computer Press, London, 2. ed edition. 5.1, 5.2
-

-
- Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *FOSE '07: Future of Software Engineering*, pages 85–103, Washington, DC, USA. 1.1, 4.4.2, 6.2
- Bertolino, A. and Gnesi, S. (2003a). PLUTO: A Test Methodology for Product Families. In *Software Product-Family Engineering, 5th International Workshop, PFE*, pages 181–197, Siena, Italy. 4.5.2.6, 4.4, C
- Bertolino, A. and Gnesi, S. (2003b). Use case-based testing of product lines. *ACM SIGSOFT Software Engineering Notes*, **28**(5), 355–358. 4.4, C
- Bezerra, Y. M., Pereira, T. A. B., and da Silveira, G. E. (2009). A systematic review of software product lines applied to mobile middleware. In *ITNG '09: Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, pages 1024–1029, Washington, DC, USA. 4.2
- 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. 4.4.2
- Briand, L. C., Labiche, Y., and He, S. (2009). Automating regression test selection based on uml designs. *Information and Software Technology*, **51**(1), 16–30. 6.3.3
- Brito, K. S. (2007). *LIFT: A Legacy InFormation retrieval Tool*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 1.3.1
- Budgen, D., Turner, M., Brereton, P., and Kitchenham, B. (2008). Using Mapping Studies in Software Engineering. In *Proceedings of PPIG Psychology of Programming Interest Group 2008*, pages 195–204, Lancaster University, UK. 4.2, 4.7, 4.8
- Buregio, V. A., Almeida, E. S., Lucredio, D., and Meira, S. L. (2007). Specification, design and implementation of a reuse repository. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 579–582, Washington, DC, USA. 1.3.1
- Burnstein (2003). *Practical Software Testing: A Process-oriented Approach*. Springer. (document), 3.2, 3.2, 3.3, 3.4.1, 3.4.2, 3.4, 3.4.4, 3.5, 3.6, 3.5, 3.6, 3.6.1, 3.6.2, 5.5, 5.5, 6.4
- Cavalcanti, Y. C. (2009). *A Bug Report Analysis and Search Tool*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 1.3.1
-

- Cavalcanti, Y. C., Martins, A. C., Almeida, E. S., and Meira, S. R. L. (2008). Avoiding duplicate cr reports in open source software projects. In *The 9th International Free Software Forum (IFSF)*, Porto Alegre, Brazil. 1.3.1
- Chen, L., Babar, M. A., and Ali, N. (2009). Variability management in software product lines: A systematic review. In *SPLC'09: Proceedings of 13th Software Product Line Conference*, San Francisco, CA, USA. 4.2
- Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA. 2.1, 2.2.1, 3.7, 5.1
- Cohen, M. B., Dwyer, M. B., and Shi, J. (2006). Coverage and adequacy in software product line testing. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63, New York, NY, USA. ACM. 4.5.2.6, 4.5.2.9, 4.4, C
- Condori-Fernandez, N., Daneva, M., Sikkel, K., Wieringa, R., Dieste, O., and Pastor, O. (2009). A systematic mapping study on empirical evaluation of software requirements specifications techniques. In *ESEM '09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 502–505, Washington, DC, USA. 4.2, 4.2
- Condrón, C. (2004). A domain approach to test automation of product lines. In *SPLiT - In International Workshop on Software Product Line Testing (2004)*, pages 27–35, Boston, MA, USA. 4.5.2.8, 4.4, 4.5.3.1, C
- Cook, T. and Campbell, D. (1979). *Quasi-experimentation: design and analysis issues for field settings*. Chicago: Rand McNally. 7.3.7
- Cunha, C. E., Cavalcanti, Y. C., da Mota Silveira Neto, P. A., Almeida, E. S., and Meira, S. R. L. (2010). A visual bug report analysis and search tool. In *22nd International Conference on Software Engineering and Knowledge Engineering (SEKE)*, San Francisco, U.S. 1.3.1
- da Cunha, C. E. A. (2009). *A Visual Bug Report Analysis and Search Tool*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 1.3.1
- de Oliveira, Junior, E. A., Gimenes, I. M. S., Huzita, E. H. M., and Maldonado, J. C. (2005). A variability management process for software product lines. In *Proceed-*

-
- ings of the conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 225–241. 2.3
- Denger, C. and Kolb, R. (2006). Testing and inspecting reusable product line components: first empirical results. In *International Symposium on Empirical Software Engineering (ISESE)*, pages 184–193, New York, NY, USA. 4.1, 4.4, C
- Durao, F. A. (2008). *Semantic Layer Applied to a Source Code Search Engine*. Master’s thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 1.3.1
- Duszynski, S., Knodel, J., and Lindvall, M. (2009). Save: Software architecture visualization and evaluation. In *Conference on Software Maintenance and Reengineering (CSMR)*, pages 323–324. 6.2
- Dybå, T. and Dingsøy, T. (2008a). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, **50**(9-10), 833–859. 4.4.4
- Dybå, T. and Dingsøy, T. (2008b). Strength of evidence in systematic reviews in software engineering. In *ESEM ’08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 178–187, New York, NY, USA. ACM. 4.4.4
- Eduardo, Frederico, Martins, A. C., Mendes, R., Melo, C., Garcia, V. C., Almeida, E. S., and Silvio (2006). Towards an effective context-aware proactive asset search and retrieval tool. In *6th Workshop on Component-Based Development (WDBC)*, pages 105–112, Recife, Pernambuco, Brazil. 1.3.1
- Edwin, O. O. (2007). *Testing in Software Product Lines*. Master’s thesis, Department of Software Engineering and Computer Science Blekinge Institute of Technology, Sweden. 1
- Engström, E., Skoglund, M., and Runeson, P. (2008). Empirical evaluations of regression test selection techniques: a systematic review. In *ESEM ’08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 22–31, New York, NY, USA. 4.5.2.4
- Feng, Y., Liu, X., and Kerridge, J. (2007). A product line based aspect-oriented generative unit testing approach to building quality components. In *COMPSAC - Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 403–408, Washington, DC, USA. 4.5.2.5, 4.4, 4.5.3.1, C
-

- Fenton, N. E. and Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA. 7.5.1.1
- Ganesan, D., Maurer, U., Ochs, M., Snoek, B., and Verlage, M. (2005). Towards testing response time of instances of a web-based product line. In *International Workshop on Software Product Line Testing (SPLIT)*, Rennes, France. 4.5.2.5, 4.4, C
- Garcia, V. C. (2010). *A Reference Model for Software Reuse Adoption in Companies*. Ph.D. thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 1.3.1
- Garcia, V. C., Lisboa, L. B., Frederico, Almeida, E. S., and Silvio (2008). A lightweight technology change management approach to facilitating reuse adoption. In *2nd Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, Porto Alegre, Brazil. 1.3.1
- Geppert, B., Li, J. J., Rößler, F., and Weiss, D. M. (2004). Towards generating acceptance tests for product lines. In *ICSR - Proceedings of 8th International Conference on Software Reuse*, pages 35–48. 4.5.2.3, 4.4, 5.1, C
- Goldsmith, R. F. and Graham, D. (2002). The forgotten phase. In *Software Development Magazine*, pages 45 – 47. 4.5.2.2
- 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. 4.5.2.1, 4.5.2.4
- H K N Leung, L. J. W. (1991). A cost model to compare regression test strategies. In *In Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 201–208, Sorrento, Italy. 6.3.4
- Harrold, M. J. (1998). Architecture-based regression testing of evolving systems. In *International Workshop on Role of Architecture in Testing and Analysis (ROSATEA 1998)*, pages 73–77, Marsala, Sicily, Italy. 4.5.2.4, 4.4, 6.3.4, C
- Harrold, M. J. (2000). Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA. 6.1, 6.7, 8
- Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. (2001). Regression test selection for java software. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented*
-

-
- programming, systems, languages, and applications*, pages 312–326, New York, NY, USA. ACM. 6.1, 6.6.1.3
- Hartmann, J., Vieira, M., and Ruder, A. (2004). A UML-based approach for validating product lines. In *International Workshop on Software Product Line Testing (SPLiT 2004)*, pages 58–65, Boston, MA. 4.5.2.6, 4.5.2.8, C
- Hatton, L. (2007). How accurately do engineers predict software maintenance tasks? *Computer*, **40**(2), 64–69. (document), 6.3.1, 6.1
- IEEE (1998). Ieee 829-1998 – ieee standard for software test documentation. Standard. 6.6.1.1
- Iso (2006). International standard - ISO/IEC 14764 IEEE Std 14764-2006. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998*, pages 1–46. 6.3.1
- Jaring, M., Krikhaar, R. L., and Bosch, J. (2008). Modeling variability and testability interaction in software product line engineering. In *ICCBSS - 7th International Conference on Composition-Based Software Systems*, pages 120–129. 4.3.3, 4.5.2.2, 4.5.2.3, 4.5.2.7, 4.4, 4.5.3.1, C
- Jin-hua, L., Qiong, L., and Jing, L. (2008). The w-model for testing software product lines. In *ISCST '08: Proceedings of the International Symposium on Computer Science and Computational Technology*, pages 690–693, Los Alamitos, CA, USA. 1.1, 4.5.2.1, 4.5.2.4, 4.4, 4.5.3.1, 5.6.1, 5.6.1, 6.6, 8.2, C
- Juristo, N., Moreno, A. M., and Vegas, S. (2004). Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, **9**(1-2), 7–44. 4.4.2
- Juristo, N., Moreno, A. M., and Strigel, W. (2006a). Guest editors' introduction: Software testing practices in industry. *IEEE Software*, **23**(4), 19–21. 4.1
- Juristo, N., Moreno, A. M., Vegas, S., and Solari, M. (2006b). In search of what we experimentally know about unit testing. *IEEE Software*, **23**(6), 72–80. 4.2
- Kamsties, E., Pohl, K., Reis, S., and Reuys, A. (2003). Testing variabilities in use case models. In *PFE'03: Proceedings of 5th International Workshop Software Product-Family Engineering*, pages 6–18, Siena, Italy. 4.5.2.3, 4.4, C
-

-
- Kang, S., Lee, J., Kim, M., and Lee, W. (2007). Towards a formal framework for product line test development. In *CIT '07: Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, pages 921–926, Washington, DC, USA. 4.5.2.6, 4.4, 5.1, C
- Kauppinen, R. (2003). *Testing framework-based software product lines*. Master's thesis, University of Helsinki Department of Computer Science. 4.1, 4.3.3, 4.4, 4.5.3.1, 8, C
- Kauppinen, R. and Taina, J. (2003). Rita environment for testing framework-based software product lines. In *SPLST'03: Proceedings of the 8th Symposium on Programming Languages and Software Tools*, pages 58–69, Kuopio, Finland. 4.1, 4.5.2.4, 4.5.2.8, 4.4, C
- Kauppinen, R., Taina, J., and Tevanlinna, A. (2004). Hook and template coverage criteria for testing framework-based software product families. In *SPLIT '04: Proceedings of the International Workshop on Software Product Line Testing*, pages 7–12, Boston, MA, USA. 4.5.2.9, 4.4, C
- Kim, K., Kim, H., Ahn, M., Seo, M., Chang, Y., and Kang, K. C. (2006). ASADAL: a tool system for co-development of software and test environment based on product line engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 783–786, New York, NY, USA. 5.1
- Kishi, T. and Noda, N. (2006). Formal verification and software product lines. *Communications of the ACM*, **49**(12), 73–77. 4.5.2.1, 4.5.2.2, 4.4, 4.5.3.1, C
- Kitchenham, B. (2010). What's up with software metrics? - a preliminary mapping study. *Journal of Systems and Software*, **83**(1), 37–51. 4.2
- 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. 4.2, 4.2, 4.4.4
- Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., Emam, K. E., and Rosenberg, J. (2002). Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, **28**(8), 721–734. 4.4.4
-

- Kitchenham, B. A., Dyba, T., and Jorgensen, M. (2004). Evidence-based software engineering. In *ICSE'04 : Proceedings of the 26th International Conference on Software Engineering*, pages 273–281, Washington, DC, USA. 4.2
- Kitchenham, B. A., Mendes, E., and Travassos, G. H. (2007). Cross versus within-company cost estimation studies: A systematic review. *IEEE Transactions on Software Engineering*, **33**(5), 316–329. 4.4.4
- Knauber, P. and Hetrick, W. (2005). Product line testing and product line development – variations on a common theme. In *SPLIT '05: Proceeding of the International Workshop on Software Product Line Testing*, Rennes, France. 5.2, 5.6.2
- Knauber, P., Muthig, D., Schmid, K., and Widen, T. (2000). Applying product line concepts in small and medium-sized companies. *IEEE Software*, **17**(5). 2.5
- Kolb, R. (2003). A risk-driven approach for efficiently testing software product lines. In *IESEF'03 - Fraunhofer Institute for Experimental Software Engineering*. 4.5.2.4, 4.5.2.8, 4.4, 4.5.3.1, 6.1, 8.2, C
- Kolb, R. and Muthig, D. (2003). Challenges in testing software product lines. In *CONQUEST'03 - Proceedings of 7th Conference on Quality Engineering in Software Technology*, pages 81–95, Nuremberg, Germany. 1, 4.1, 4.3.3, 4.5.2.4, 4.4, 8, C
- Kolb, R. and Muthig, D. (2006). Making testing product lines more efficient by improving the testability of product line architectures. In *ROSATEA '06: Proceedings of the ISSA workshop on Role of software architecture for testing and analysis*, pages 22–27, New York, NY, USA. 4.5.2.2, 4.5.2.3, 4.5.2.6, 4.5.2.8, 4.4, 5.6, 6.1, 6.2, C
- Krueger, C. W. (2002). Easing the transition to software mass customization. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 282–293, London, UK. 2.4
- Lamancha, B. P., Usaola, M. P., and Velthius, M. P. (2009). Software product line testing - a systematic review. In *ICSOFT International Conference on Software and Data Technologies*, pages 23–30. INSTICC Press. 4.7, 4.8
- Lee, K. and Kang, K. C. (2004). Feature dependency analysis for product line component design. In *ICSR '04: Proceedings of the International Conference on Software Reuse*, pages 69–85, Madrid, Spain. 5.6.1

- Leung, H. K. N. and White, L. (1989). Insights into regression testing. In *ICSM '89: Proceedings of the International Conference on Software Maintenance*, pages 60–69. (document), 6.3.1, 6.3.2, 6.3.3, 6.1, 6.3.3
- Li, J. J., Weiss, D. M., and Slye, J. H. (2007a). Automatic system test generation from unit tests of exvantage product family. In *SPLIT '07: Proceedings of the International Workshop on Software Product Line Testing*, pages 73–80, Kyoto, Japan. 4.5.2.3, 4.5.2.8, 4.4, 4.5.3.1, 5.2, C
- Li, J. J., Geppert, B., Roessler, F., and Weiss, D. (2007b). Reuse execution traces to reduce testing of product lines. In *SPLIT '07: Proceedings of the International Workshop on Software Product Line Testing*, Kyoto, Japan. 4.5.2.8, 4.4, C
- Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 6.3.1
- Linden, F. J. v. d., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. (document), 2.2, 2.2.1, 2.1, 5.1, 8
- Lisboa, L. B. (2008). *ToolDAy - A Tool for Domain Analysis*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 1.3.1
- Lisboa, L. B., Garcia, V. C., Lucrédio, D., de Almeida, E. S., de Lemos Meira, S. R., and de Mattos Fortes, R. P. (2010). A systematic review of domain analysis tools. *Information and Software Technology*, **52**(1), 1–13. 4.2
- Long, M. A. (1993). Software regression testing success story. In *ITC'93 : Proceedings IEEE International Test Conference, Designing, Testing, and Diagnostics*, pages 271–272, Baltimore, MD, USA. 8
- Markus Gälli, O. G. and Nierstrasz, O. (2005). Composing unit tests. In *SPLIT '05: Proceedings of the International Workshop on Software Product Line Testing*, pages 16–22, Rennes, France. 5.3
- Martins, A. C., Garcia, V. C., Almeida, E. S., and Silvio (2008). Enhancing components search in a reuse environment using discovered knowledge techniques. In *SBCARS'08: Proceedings of the 2nd Brazilian Symposium on Software Components, Architectures, and Reuse*, Porto Alegre, Brazil. 1.3.1
-

- Mascena, J. C. C. P., Meira, S. R. d. L., de Almeida, E. S., and Garcia, V. C. (2006). Towards an effective integrated reuse environment. In *GPCE'06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 95–100, New York, NY, USA. 1.3.1
- McGregor, J. (2009). Variation verification. *Journal of Object Technology*, **8**(2), 7–14. 6.6.1.3
- McGregor, J., Sodhani, P., and Madhavapeddi, S. (2004a). Testing Variability in a Software Product Line. In *SPLIT '04: Proceedings of the International Workshop on Software Product Line Testing*, page 45, Boston, Massachusetts, USA. 1, 2.3
- McGregor, J. D. (2001a). Structuring test assets in a product line effort. In *ICSE'01: In Proceedings of the 2nd International Workshop on Software Product Lines: Economics, Architectures, and Implications*, pages 89–92, Toronto, Ontario, Canada. 4.5.2.8, C
- McGregor, J. D. (2001b). Testing a software product line. Technical Report CMU/SEI-2001-TR-022. 3.7, 4.1, 4.3.3, 4.5.2.2, 4.5.2.4, 4.4, 4.5.3.1, 5.1, 5.2, 5.3, 5.6.1, 6.4, 8, 8.2, C
- McGregor, J. D. (2002). Building reusable test assets for a product line. In *ICSR'02: Proceedings of 7th International Conference on Software Reuse*, pages 345–346, Austin, Texas, USA. 4.5.2.3, 4.5.2.8, 4.4, 4.5.3.1, C
- McGregor, J. D. (2008). Toward a fault model for software product lines. In *SPLC '08: Proceedings of the Software Product Line Conference*, pages 157–162, Limerick, Ireland. 7.3.6
- McGregor, J. D., Sodhani, P., and Madhavapeddi., S. (2004b). Testing variability in a software product line. In *SPLIT '04: Proceedings of the International Workshop on Software Product Line Testing*, pages 45–50, Boston, MA. 4.5.2.6, 4.5.2.7, 4.4, C
- McIlroy, D. (1968). Mass-produced software components. In *ICSE '68: Proceedings of the 1st International Conference on Software Engineering*, pages 88–98, Garmisch Pattenkirchen, Germany. 2.1
- Medeiros, F. M., de Almeida, E. S., and de Lemos Meira, S. R. (2009). Towards an approach for service-oriented product line architecture. In *SPLC'09: Proceedings of the Software Product Line Conference*, San Francisco, CA, USA. 1.3.1

- Mendes, R. C. (2008). *Search and Retrieval of Reusable Source Code using Faceted Classification Approach*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 1.3.1
- Moraes, M. B. S., Almeida, E. S., and de Lemos Meira, S. R. (2009). A systematic review on software product lines scoping. In *ESELAW09: 'Proceedings of the VI Experimental Software Engineering Latin American Workshop*, Sao Carlos-SP, Brazil. 4.2
- Muccini, H. and van der Hoek, A. (2003). Towards testing product line architectures. *Electronic Notes in Theoretical Computer Science*, **82**(6). 3.4.2, 4.5.2.4, 4.4, 5.5, 5.6.2, 6.2, C
- Muccini, H., Dias, M. S., and Richardson, D. J. (2004). Systematic testing of software architectures in the c2 style. In *FASE'04: Proceedings International Conference on Fundamental Approaches to Software Engineering*, pages 295–309, Barcelona, Spain. 8.1, 8.2
- Muccini, H., Dias, M. S., and Richardson, D. J. (2005). Towards software architecture-based regression testing. In *WADS '05: Proceedings of the Workshop on Architecting Dependable Systems*, pages 1–7, New York, NY, USA. 4.5.2.4
- Muccini, H., Dias, M., and Richardson, D. J. (2006). Software architecture-based regression testing. *Journal of Systems and Software*, **79**(10), 1379–1396. 4.5.2.4, 5.6.1, 6.1, 6.7, 8, 8.2
- Myers, G. J. (2004). *The Art of Software Testing*. Wiley, 2 edition. 5.5, 5.5
- Nascimento, L. M. (2008). *Core Assets Development in SPL - Towards a Practical Approach for the Mobile Game Domain*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 1.3.1
- Nebut, C., Fleurey, F., Traon, Y. L., and Jézéquel, J.-M. (2003). A requirement-based approach to test product families. In *PFE'03: Proceedings of 5th International Workshop Software Product-Family Engineering*, pages 198–210, Siena, Italy. 4.5.2.6, 4.5.2.8, 4.4, 4.5.3.1, C
- Nebut, C., Traon, Y. L., and Jézéquel, J.-M. (2006). System testing of product lines: From requirements to test cases. In *Software Product Lines*, page 447. 4.5.2.3, 4.5.2.8, 4.4, C

- Needham, D. and Jones, S. (2006). A software fault tree metric. In *ICSM'06: Proceedings of the International Conference on Software Maintenance*, pages 401–410, Philadelphia, Pennsylvania, USA. C
- Neiva, D. F. S. (2008). *RiPLE-RE: A Requirements Engineering Process for Software Product Lines*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil. 5.6.1
- Neiva, D. F. S.; Almeida, E. S. M. S. R. L. (2009). An experimental study on requirements engineering for software product lines. In *35th IEEE EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Service and Component Based Software Engineering (SCBSE) Track*, Patras, Greece. 1.3.1
- Northrop, L. M. (2002). Sei's software product line tenets. *IEEE Software*, **19**(4), 32–40. (document), 2.1, 2.1, 2.2.1, 2.2, 2.3
- Northrop, L. M. and Clements, P. C. (2007). A framework for software product line practice, version 5.0. Technical report, Software Engineering Institute. 4.1, 4.5.2.8
- Odia, O. E. (2007). *Testing in Software Product Lines*. Master's thesis, School of Engineering at Blekinge Institute of Technology. 4.3.3, 4.4, 4.5.3.1, 4.7, C
- Olimpiew, E. and Gomaa, H. (2005a). Reusable system tests for applications derived from software product lines. In *SPLIT '05: Proceedings of the International Workshop on Software Product Line Testing*, Rennes, France. 4.5.2.3, 4.5.2.6, 4.4, C
- Olimpiew, E. M. and Gomaa, H. (2005b). Model-based testing for applications derived from software product lines. In *A-MOST'05: Proceedings of the 1st International Workshop on Advances in model-based testing*, pages 1–7, New York, NY, USA. 4.5.2.8, C
- Olimpiew, E. M. and Gomaa, H. (2009). Reusable model-based testing. In *ICSR '09: Proceedings of the 11th International Conference on Software Reuse*, pages 76–85, Berlin, Heidelberg. 4.5.2.6, 4.5.2.8, 4.4, C
- Orso, A., Shi, N., and Harrold, M. J. (2004). Scaling regression testing to large software systems. In *SIGSOFT'04: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 241–251, New York, NY, USA. 6.1, 6.3.4, 6.3.4
-

-
- Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). Systematic mapping studies in software engineering. In *EASE '08: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, University of Bari, Italy. (document), 4, 4.1, 4.2, 4.1, 4.2, 4.4.4, 4.5.1, 4.5.3, 4.5.3
- Pohl, K. and Metzger, A. (2006). Software product line testing. *Communications of the ACM*, **49**(12), 78–81. 4.1, 4.3.3, 4.4, 8, C
- Pohl, K. and Sikora, E. (2005). Documenting variability in test artefacts. In *Software Product Lines*, pages 149–158. Springer. 4.4, C
- Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005a). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Secaucus, NJ, USA. (document), 1.1, 2.1, 2.2.1, 2.2.1, 2.3, 2.4, 2.1, 6.1
- Pohl, K., Böckle, G., and van der Linden, F. J. (2005b). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer. 4.5.2.3, 4.5.2.6
- Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, fifth edition. 6.3.1
- Pretorius, R. and Budgen, D. (2008). A mapping study on empirical evidence related to the models and forms used in the uml. In *ESEM'08: Proceedings of Empirical Software Engineering and Measurement*, pages 342–344, Kaiserslautern, Germany. 4.2
- Reis, S.; Metzger, A. P. K. (2006). A reuse technique for performance testing of software product lines. In *SPLIT '05: Proceedings of the International Workshop on Software Product Line Testing*, Baltimore, Maryland, USA. 4.5.2.5, 4.4, 4.5.3.1, C
- Reis, S., Metzger, A., and Pohl, K. (2007a). Integration testing in software product line engineering: A model-based technique. In *FASE'07: Proceedings of the Fundamental Approaches to Software Engineering*, pages 321–335, Braga, Portugal. 4.4
- Reis, S., Metzger, A., and Pohl, K. (2007b). Integration testing in software product line engineering: A model-based technique. In *FASE'07: Proceedings of the Fundamental Approaches to Software Engineering*, pages 321–335, Braga, Portugal. 5.2, C
- Reuys, A., Kamsties, E., Pohl, K., and Reis, S. (2005). Model-based system testing of software product families. In *CAiSE'05: Proceedings of International Conference on Advanced Information Systems Engineering*, pages 519–534. 4.5.2.6, 4.4, C
-

- Reuys, A., Reis, S., Kamsties, E., and Pohl, K. (2006). The scented method for testing software product lines. In *Software Product Lines*, pages 479–520. 4.1, 4.5.2.1, 4.5.2.3, 4.5.2.6, 4.5.2.7, 4.5.2.8, 4.4, 5.2, 8, C
- Rook, P. (1986). Controlling software projects. *Software Engineering Journal*, **1**(1), 7–16. 3.4
- Rosik, J., Le Gear, A., Buckley, J., and Ali Babar, M. (2008). An industrial case study of architecture conformance. In *ESEM'08: Proceedings of International symposium on Empirical software engineering and measurement*, pages 80–89, New York, NY, USA. 6.6.1.2
- Rothermel, G. and Harrold, M. J. (1994). A framework for evaluating regression test selection techniques. In *ICSE'94: Proceedings of the International Conference on Software Engineering*, pages 201–210, Sorrento, Italy. 6.3.4
- Rothermel, G. and Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, **22**(8), 529–551. 4.3.3, 4.5.2.1, 4.5.2.4, 5.1, 6.1, 6.3.3, 6.3.4, 6.3.4
- 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. 6.3.4
- Rothermel, G., Untch, R. J., and Chu, C. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, **27**(10), 929–948. 6.6.1.3
- Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education. 4.5.2.6
- Sellier, D., Elguezabal, G. B., and Urchegui, G. (2007). Introducing software product line engineering for metal processing lines in a small to medium enterprise. In *SPLC'07: Proceedings of the Software Product Line Conference*, pages 54–62, Kyoto, Japan. 2.5
- Shaw, M. and Clements, P. (2006). The golden age of software architecture. *IEEE Software*, **23**(2), 31–39. 6.1
- Souza Filho, E. D., Oliveira Cavalcanti, R., Neiva, D. F., Oliveira, T. H., Lisboa, L. B., Almeida, E. S., and Lemos Meira, S. R. (2008). Evaluating domain design approaches using systematic review. In *ECSSA '08: Proceedings of the 2nd European conference on Software Architecture*, pages 50–65, Berlin, Heidelberg. 1.3.1

- SPEM, O. (2008). Software Process Engineering Metamodel (SPEM). Technical report, Technical report, Object Management Group. 5.4.1
- Staff, I. M. (1992). On the edge: Regression testability. *IEEE Micro*, **12**(2), 81–84. 6.1, 6.3.2
- Svahnberg, M. and Bosch, J. (1999). Evolution in software product lines: Two cases. *Journal of Software Maintenance*, **11**(6), 391–422. 6.1
- Svahnberg, M., van Gorp, J., and Bosch, J. (2005). A taxonomy of variability realization techniques: Research articles. *Software Practice and Experience*, **35**(8), 705–754. 2.3
- Tevanlinna, A., Taina, J., and Kauppinen, R. (2004). Product family testing: a survey. *ACM SIGSOFT Software Engineering Notes*, **29**(2), 12. 4.1, 4.3.3, 4.5.2.1, 4.5.2.2, 4.5.2.6, 4.5.2.9, 4.5.3.1, 4.7, 5.1, 5.6.1, 6.5
- Todd Graves, Mary Jean Harrold, J.-M. K. A. P. G. R. (1998). An empirical study of regression test selection techniques. In *ICSE'98: Proceedings of the International Conference on Software Engineering*, pages 188–197, Kyoto, Japan. 6.3.4
- Tracz, W. (1988). Software reuse myths. *ACM SIGSOFT Software Engineering Notes*, **13**(1), 17–21. 2.1
- Vanderlei, T. A., Dur ao, F. A., Martins, A. C., Garcia, V. C., Almeida, E. S., and de L. Meira, S. R. (2007). A cooperative classification mechanism for search and retrieval software components. In *SAC'07: Proceedings of the ACM symposium on Applied computing*, pages 866–871, New York, NY, USA. 1.3.1
- Šmite, D., Wohlin, C., Gorschek, T., and Feldt, R. (2010). Empirical evidence in global software engineering: a systematic review. *Empirical Software Engineering*, **15**(1), 91–118. 4.5.3
- Wahl, N. J. (1999). An overview of regression testing. *ACM SIGSOFT Software Engineering Notes*, **24**(1), 69–73. 6.1, 6.3.1, 6.3.4, 8
- Weiss, D. M., C. P. C.-K. K. and Krueger, C. (2006). Software product line hall of fame. page 237, Washington, DC, USA. IEEE Computer Society. 2.6
- Weiss, D. M. (2008). The product line hall of fame. In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, page 395, Washington, DC, USA. IEEE Computer Society. 4.1
-

- Wieringa, R., Maiden, N. A. M., Mead, N. R., and Rolland, C. (2006). Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements Engineering*, **11**(1), 102–107. 4.5.1
- Winbladh, K., Alspaugh, T. A., Ziv, H., and Richardson, D. (2006). Architecture-based testing using goals and plans. In *ROSATEA'06: Proceedings of the Workshop on Role of software architecture for testing and analysis*, pages 64–68, New York, NY, USA. 6.2
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA. (document), 7.2.1, 7.1, 7.3.1, 7.3.2, 7.3.4, 7.3.5, 7.3.7, 7.4.3, 7.5.1.1
- Wübbecke, A. (2008). Towards an efficient reuse of test cases for software product lines. In *SPLC'08: Proceedings of Software Product Line Conference*, pages 361–368. 4.5.2.3, 4.5.2.6, 4.4, C
- Zeng, H., Zhang, W., and Rine, D. (2004). Analysis of testing effort by using core assets in software product line testing. In *SPLIT '04: Proceedings of the International Workshop on Software Product Line Testing*, pages 1–6, Boston, MA. 4.5.2.1, 4.5.2.8, 4.4, C

Appendices



Experimental Study Questionnaires

A.1 Background Questionnaire

Name:

Date:

Amount of years since graduation:

1. - Did you conclude any post-graduation course?
 Specialization M.Sc. Ph.D. None
2. - Are you doing any post-graduation course?
 Specialization Student M.Sc. Student Ph.D. Student None
3. - Did you take any testing course during your studies?
 None Graduation Specialization M.Sc. Ph.D. Other (e.g. Quality)
Number of hours (Total):
4. - How you classify your experience in Java?
 High Medium Low None
5. - Which is your experience in software development? And how many years?
 Never develop.
 I have experience developing software alone. Years:
 I have experience developing software during a course. Years:
 I have experience developing software in company project. Years:

A.1. BACKGROUND QUESTIONNAIRE

6. - What is your experience in software testing? And how many years?

- Never test.
- I have experience testing software alone. Years:
- I have experience testing software during a course. Years:
- I have experience testing software in company project. Years:

7. - Describe your experience regarding the JUnit framework. And how many years?

Area	None	Low	Medium	High	Years
Academic					
Commercial					

8. - Describe your experience regarding the EclEmma tool. And how many years?

Area	None	Low	Medium	High	Years
Academic					
Commercial					

9. - Describe your experience regarding to regression testing tools. (Ex.: Jdiff, DejaVoo) And how many years?

Area	None	Low	Medium	High	Years
Academic					
Commercial					

10. - How many commercial software projects did you participate after graduation? Which role? (Tester, Developer or other).

Category	Quantity	Role(s)
Low Complexity		
Medium Complexity		
High Complexity		

11. - Regarding your personal experience in Software Testing (Integration Testing) (mark x):

Area	None	Low	Medium	High	Years
Academic					
Commercial					

A.1. BACKGROUND QUESTIONNAIRE

Area	None	Low	Medium	High	Years
Academic					
Commercial					

12. - Regarding your personal experience in Software Testing (Regression Testing) (mark x):

13. - How do you describe your experience in control flow graphs (CFG) analysis? ()
None () Low () Medium () High

14. - Have you ever performed any analysis of regression testing (test selection) before?
If yes, which technique was used?

15. - Regarding your personal experience in test case design (mark x): Using White-box Techniques:

Area	None	Low	Medium	High	Years
Academic					
Commercial					

Using Black-box Techniques:

Area	None	Low	Medium	High	Years
Academic					
Commercial					

16. - Regarding your personal experience in test case prioritization (mark x):

Area	None	Low	Medium	High	Years
Academic					
Commercial					

17. - Regarding your personal experience in software product lines (SPL) development?
And how many years? () Never develop.

() I have experience developing software alone. Years:

() I have experience developing software during a course. Years:

() I have experience developing software in company project. Years:

A.2 Regression Testing Approach Analysis Questionnaire

Name:

Date:

Regarding the regression testing approach, please answer the following questions:

1. - Did you have any difficulties in understanding or applying the regression testing approach? Which one(s)?
2. - In your opinion, what are the strengths of the regression testing approach?
3. - In your opinion, what are the weak points of the regression testing approach?
4. - Is there any missing activity, roles or artifact in the regression testing approach?
Why?
5. - Which improvements would you suggest for the regression testing approach?
(Ex.: The graph comparison step can be replaced by textual comparison)
6. How many faults did you find using the regression testing approach? And where the fault was found?(Ex.: The fault X was identified on class Y in the Z method)
7. How did you classify the test cases (old and new test cases)? Answer the question adding the list of test cases identified by its ID.

Obsolete:

Retestable:

Reusable:

Unclassified:

- New-Structural Tests:

- New-Specification tests:

Number of updated test cases:

8. How much time did you spend in each step of the approach (Corrective Regression)?
 - 1) Planning:
 - 2) Analyzes:
 - 3) Graph Generation:

A.2. REGRESSION TESTING APPROACH ANALYSIS QUESTIONNAIRE

- 3a) Graph Comparison:
- 3b) Textual Comparison:
- 4) Test Design and Selection:
- 5) Instrumentation:
- 6) Test Suite Composition:
- 7) Test Case Prioritization:
- 8) Execution:
- 9) Reporting:

9. How much time did you spend in each step of the approach (Progressive Regression)?

- 1) Planning:
- 2) Analyzes:
- 3) Specification Comparison
- 4) Test Design and Selection:
- 5) Instrumentation:
- 6) Test Suite Composition:
- 7) Test Case Prioritization:
- 8) Execution:
- 9) Reporting:

B

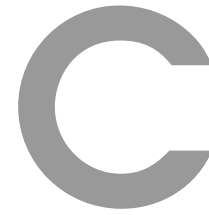
Mapping Study Sources

B.1 List of Conferences

Acronym	Conference Name
AOSD	International Conference on Aspect-Oriented Software Development
APSEC	Asia Pacific Software Engineering Conference
ASE	International Conference on Automated Software Engineering
CAiSE	International Conference on Advanced Information Systems Engineering
CBSE	International Symposium on Component-based Software Engineering
COMPSAC	International Computer Software and Applications Conference
CSMR	European Conference on Software Maintenance and Reengineering
ECBS	International Conference and Workshop on the Engineering of Computer Based Systems
ECOWS	European Conference on Web Services
ECSA	European Conference on Software Architecture
ESEC	European Software Engineering Conference
ESEM	Empirical Software Engineering and Measurement
WICSA	Working IEEE/IFIP Conference on Software Architecture
FASE	Fundamental Approaches to Software Engineering
GPCE	International Conference on Generative Programming and Component Engineering
ICCBSS	International Conference on Composition-Based Software Systems
ICSE	International Conference on Software Engineering
ICSM	International Conference on Software Maintenance
ICSR	International Conference on Software Reuse
ICST	International Conference on Software Testing, Verification and Validation
ICWS	International Conference on Web Services
IRI	International Conference on Information Reuse and Integration
ISSRE	International Symposium on Software Reliability Engineering
MODELS	International Conference on Model Driven Engineering Languages and Systems
PROFES	International Conference on Product Focused Software Development and Process Improvement
QoSA	International Conference on the Quality of Software Architectures
QSIC	International Conference on Quality Software
ROSATEA	International Workshop on The Role of Software Architecture in Testing and Analysis
SAC	Annual ACM Symposium on Applied Computing
SEAA	Euromicro Conference on Software Engineering and Advanced Applications
SEKE	International Conference on Software Engineering and Knowledge Engineering
SERVICES	Congress on Services
SPLC	Software Product Line Conference
SPLiT	Software Product Line Testing Workshop
TAIC PART	Testing - Academic & Industrial Conference
TEST	International Workshop on Testing Emerging Software Technology

B.2 List of Journals

Journals
ACM Transactions on Software Engineering and Methodology (TOSEM)
Communications of the ACM (CACM)
ELSEVIER Information and Software Technology (IST)
ELSEVIER Journal of Systems and Software (JSS)
IEEE Software
IEEE Computer
IEEE Transactions on Software Engineering
Journal of Software Maintenance Research and Practice
Software Practice and Experience Journal
Software Quality Journal
Software Testing, Verification and Reliability



Quality Studies Scores

Id	REF	Study Title	Year	A	B	C
1	Condron (2004)	A Domain Approach to Test Automation of Product Lines	2004	2	0	2
2	Feng <i>et al.</i> (2007)	A product line based aspect-oriented generative unit testing approach to building quality components	2007	1.5	0	2.5
3	Nebut <i>et al.</i> (2003)	A Requirement-Based Approach to Test Product Families	2003	2.5	1	1.5
4	Reis (2006)	A Reuse Technique for Performance Testing of Software Product Lines	2006	1.5	2	3
5	Kolb (2003)	A Risk-Driven Approach for Efficiently Testing Software Product Lines	2003	2	1	2.5
6	Needham and Jones (2006)	A Software Fault Tree Metric	2006	0	0	1
7	Hartmann <i>et al.</i> (2004)	A UML-Based approach for Validating Product Lines	2004	1	2	0.5
8	Zeng <i>et al.</i> (2004)	Analysis of Testing Effort by Using Core Assets in Software Product Line Testing	2004	1	1.5	2.5
9	Harrold (1998)	Architecture-Based Regression Testing of Evolving Systems	1998	0	0.5	2
10	Li <i>et al.</i> (2007a)	Automatic Integration Test Generation from Unit Tests of eXVantage Product Family	2007	1	1	2
11	McGregor (2002)	Building reusable test assets for a product line	2002	2	2	0.5
12	Kolb and Muthig (2003)	Challenges in testing software product lines	2003	0	3	1.5
13	Cohen <i>et al.</i> (2006)	Coverage and adequacy in software product line testing	2006	1	1.5	2
14	Pohl and Sikora (2005)	Documenting Variability in Test Artefacts	2005	1	0	1
15	Kishi and Noda (2006)	Formal verification and software product lines	2006	2	1.5	2
16	Kauppinen <i>et al.</i> (2004)	Hook and Template Coverage Criteria for Testing Framework-based Software Product Families	2004	0.5	0.5	3
17	Reis <i>et al.</i> (2007b)	Integration Testing in Software Product Line Engineering: A Model-Based Technique	2007	1	0	3
18	Kolb and Muthig (2006)	Making testing product lines more efficient by improving the testability of product line architectures	2006	1	1.5	1.5
19	Reuys <i>et al.</i> (2005)	Model-Based System Testing of Software Product Families	2005	2	1	3.5
20	Olimpiew and Gomaa (2005b)	Model-based Testing For Applications Derived from Software Product Lines	2005	0	1	1

* The shaded lines represent the most relevant studies according to the grades.

Id	REF	Study Title	Year	A	B	C
21	Jaring <i>et al.</i> (2008)	Modeling Variability and Testability Interaction in Software Product Line Engineering	2008	2.5	6	3.5
22	Bertolino and Gnesi (2003a)	PLUTO: A Test Methodology for Product Families	2003	0.5	1	3
23	Olimpiew and Gomaa (2009)	Reusable Model-Based Testing	2009	3	0.5	3.5
24	Olimpiew and Gomaa (2005a)	Reusable System Tests for Applications Derived from Software Product Lines	2005	2.5	1	1
25	Li <i>et al.</i> (2007b)	Reuse Execution Traces to Reduce Testing of Product Lines	2007	0	0.5	2
26	Kauppinen and Taina (2003)	RITA environment for testing framework-based software product lines	2003	0	0	0.5
27	Pohl and Metzger (2006)	Software Product Line Testing Exploring principles and potential solutions	2006	0.5	0	2.5
28	McGregor (2001a)	Structuring Test Assets in a Product Line Effort	2001	1.5	1	0.5
29	Nebut <i>et al.</i> (2006)	System Testing of Product Lines From Requirements to Test Cases	2006	0	2	2
30	McGregor (2001b)	Testing a Software Product Line	2001	4	1.5	2
31	Denger and Kolb (2006)	Testing and inspecting reusable product line components: first empirical results	2006	0	1	0.5
32	Kauppinen (2003)	Testing Framework-Based Software Product Lines	2003	0.5	0.5	2
33	Odia (2007)	Testing in Software Product Line	2007	2	2.5	2
34	Al-Dallal and Sorenson (2008)	Testing Software Assets of Framework-Based Product Families during Application Engineering Stage	2008	3	1	4
35	Kamsties <i>et al.</i> (2003)	Testing variabilities in use case models	2003	0.5	1.5	1.5
36	McGregor <i>et al.</i> (2004b)	Testing Variability in a Software Product Line	2004	0	1	2.5
37	Reuys <i>et al.</i> (2006)	The ScENTED Method for Testing Software Product Lines	2006	3	1	4.5
38	Jin-hua <i>et al.</i> (2008)	The W-Model for Testing Software Product Lines	2008	1	3	1.5
39	Kang <i>et al.</i> (2007)	Towards a Formal Framework for Product Line Test Development	2007	2	2	1
40	Beatriz Pérez Lamancha (2009)	Towards an automated testing framework to manage variability using the UML Testing Profile	2009	0	0	1
41	Wübbecke (2008)	Towards an Efficient Reuse of Test Cases for Software Product Lines	2008	0	0	2
42	Geppert <i>et al.</i> (2004)	Towards Generating Acceptance Tests for Product Lines	2004	0.5	1.5	2
43	Muccini and van der Hoek (2003)	Towards Testing Product Line Architectures	2003	0	2.5	1
44	Ganesan <i>et al.</i> (2005)	Towards Testing Response Time of Instances of a web-based Product Line	2005	1	1.5	1
45	Bertolino and Gnesi (2003b)	Use Case-based Testing of Product Lines	2003	1	1	2.5