AN ACTION RESEARCH OF ACHIEVEMENTS IN A SOFTWARE PRODUCT LINE IMPLEMENTATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

MUHİTTİN ERDEM ERGÜL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JANUARY 2014

Approval of the thesis:

AN ACTION RESEARCH OF ACHIEVEMENTS IN A SOFTWARE PRODUCT LINE IMPLEMENTATION

submitted by **MUHİTTİN ERDEM ERGÜL** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Eng.** _____

Prof. Dr. Semih Bilgen
Supervisor, **Electrical and Electronics Eng. Dept., METU** _____

Examining Committee Members:

Assoc. Prof. Dr. Cüneyt Bazlamaçcı
Electrical and Electronics Engineering Dept., METU _____

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Altan Koçyiğit
Information Systems Dept., METU _____

Assoc. Prof. Dr. Şenan Ece Güran Schmidt
Electrical and Electronics Engineering Dept., METU _____

Cumhur ÜNLÜ, M.Sc.
Test Engineering Department, ASELSAN A.Ş. _____

Date: 21.01.2014

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: MUHİTTİN ERDEM ERGÜL

Signature :

ABSTRACT

AN ACTION RESEARCH OF ACHIEVEMENTS IN A SOFTWARE PRODUCT LINE IMPLEMENTATION

Ergül, Muhittin Erdem
M.S., Department of Electrical and Electronics Engineering
Supervisor : Prof. Dr. Semih Bilgen

January 2014, 87 pages

Software product lines emphasize, with an innovative approach, the idea of predictive re-use. In this way, significant improvements are provided in cost, time-to-market and quality and market dominance is enabled in the target area. The majority of the academic studies in this area are case studies. In this study it is also intended to provide scientific data to the literature about the achievements brought about by software product lines. First, the challenges in the software development activities performed under ASELSAN software testing department are determined. In this respect, software testing simulators software product line is established in order to improve software development efforts and the achievements are discussed with the measurements and analysis. It is observed that software product line adoption led to significant effort reduction, improved product quality, homogenous look-and-feel and improved reusability in the software development efforts of software testing department of ASELSAN REWIS division.

Keywords: Software Product Line, SPL Case Studies, Software Development Improvement, Software Development Metrics, Action Research

ÖZ

BİR YAZILIM ÜRÜN HATTI UYGULAMASINDAKİ KAZANIMLAR ÜZERİNE EYLEM ARAŞTIRMASI

Ergül, Muhittin Erdem
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Semih Bilgen

Ocak 2014 , 87 sayfa

Yazılım ürün hatları yenilikçi bir yaklaşımla kestirimci yeniden kullanım fikrini öne çıkarır. Bu sayede maliyet, pazara çıkış zamanları ve kalitede önemli iyileştirmeler yakalanabilmekte ve hedef markette baskın konuma geçme sağlanabilmektedir. Bu alanda yapılan akademik çalışmaların çoğunluğunu vaka analizleri oluşturmaktadır. Bu çalışmada da literatüre yazılım ürün hatlarındaki kazanımlarla ilgili bilimsel veri sağlamak amaçlanmıştır. Öncelikle ASELSAN yazılım test bölümünde yapılan yazılım geliştirme faaliyetlerindeki zorluklar belirlenmiştir. Bu doğrultuda yazılım geliştirme çabalarını iyileştirmek adına yazılım test simülatörleri yazılım ürün hattı kurulmuştur ve alınan ölçümler ve yapılan analizler ile kazanımlar tartışılmıştır. Yazılım ürün hattına geçiş, ASELSAN REHIS bölümü yazılım test departmanının yazılım geliştirme çabalarında önemli ölçüde efor azaltma, ürün kalitesinde iyileşme, homojen görünüm ve yeniden kullanımda iyileşmeye yol açmıştır.

Anahtar Kelimeler: Yazılım Ürün Hattı, YÜH Vaka İncelemeleri, Yazılım Geliştirme İyileştirmesi, Yazılım Geliştirme Ölçütleri, Eylem Araştırması

To my family

ACKNOWLEDGMENTS

I would like to express my appreciation to my supervisor Professor Semih Bilgen for his constant support, guidance and encouragement throughout the thesis.

My family also provided very precious support for this work. I am grateful for all the love and support by Hacer-Ahmet-Elif-Nuri-Sultan Ergül.

Finally, I would like to thank my colleagues in ASELSAN Inc. for their support and assistance. Special thanks to Uğur Zöngür and Tuncer Erdoğan for their contributions. They have big role in the development of the legacy infrastructure that this thesis work adopts and improves. Thanks to Hakan Bosnalı, Önder Cezayirli, Ufuk Şeker, Yeşim Okşar, Ömer Faruk Moralıoğlu and Murat Yılmaz for their collaboration in taking the measurements which is a crucial part of the study.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ATCS	Application Test Case Scenario
BAPO	Business, Architecture, Process, Organisation
CBO	Coupling Between Object Classes
CC	Cyclomatic Complexity
CMMI	Capability Maturity Model Integration
COBALT	Communication Basics Library for Testing
CSCI	Computer Software Configuration Item
DIT	Depth of Inheritance Tree
DTCS	Domain Test Case Scenario
DTR	Data Tree Representation
FEF	Family Evaluation Framework
GUI	Graphical User Interface
HWCI	Hardware Configuration Item
LCOM	Lack of Cohesion of Methods
MVC	Model View Controller
NOC	Number of Children
REWIS	Radar, Electronic Warfare and Intelligence Systems
RFC	Response for a Class
SSD	Single System Development
SRS	Software Requirements Specification
STS	Software Testing Simulators
SUS	System Usability Scale
SPL	Software Product Line
TBD	To Be Defined
VP	Variation Point
WMC	Weighted Methods per Class

CHAPTER 1

INTRODUCTION

Software products are gaining importance day by day, and being used in a broader range. However, the competition in software markets is also increasing rapidly, cost and time-to-market become of utmost importance. Re-use methods provide great support to software development at this point.

The idea of program families was suggested in the 1970s. Parnas defined program families in [1] as sets of programs sharing extensive common properties. When program families are the subject, studying the common properties is advantageous before analyzing the individual members. The product line concept was totally introduced in the early 1990s. Unlike single system development, software product line (SPL) approach enables large-scale reuse of common assets. This leads to a reduction in the total amount of development effort. Other improvements of the approach are quality assurance, higher reliability, security and usability of the final product [2].

Besides its advantages, software product line adoption is a tricky effort for an organization which may fail after some struggle. In [3] common obstacles and their remedies are described and an incremental progress is recommended instead of a revolutionary approach. This is the first decision to be taken at the beginning of the adoption process. An evolutionary approach minimizes the risks [4]. Furthermore, in order organizations to more systematically adopt software product lines, it is helpful to make an evaluation like Families Evaluation Framework described in [2]. An appropriate evaluation guides to see the current status of

the organization and to determine a realistic roadmap.

Case studies provide important data to the literature about the challenges and the achievements of SPL adoption in practice. Systematic large-scale reuse leads companies to achieve cost, time-to-market and quality improvements. High customizability and flexibility can be achieved by variability management in a SPL. In addition, SPL adoption improves maintenance and modifiability of the software products.

Test Engineering Department of ASELSAN REWIS division performs black-box testing to verify specified behavior of the software. Software requirements of the REWIS products usually comprise electrical messaging scenarios which is only visible when simulators of other units are used. Thus, software test engineers need to develop this kind of software to use for their testing activities. In general, simulator development constitutes a significant part of the overall testing effort. Formerly, traditional single system development approach was used in software development. However, some problems were seen in the context of simulator development efforts. These are effort overload, demand for higher product quality, heterogeneous look-and-feel and low reusability. Then SPL adoption is suggested to overcome these challenges of software development. Software testing simulators software product line (STS-SPL) is constructed by extending the legacy system COBALT (Communication Basics Library for Testing). COBALT provided the reusable software assets of the STS-SPL. STS-SPL is created by performing a few key activities. First, short and long term strategy of the business and SPL product portfolio are defined. Then, STS-SPL requirements specification is generated. In order to represent variability in requirements, a labeling system is used. COBALT framework is an infrastructure that has been locally developed in ASELSAN and not formally documented. Thus, reference architecture and the variability model of the product line is determined by examining the COBALT infrastructure. Later, FEF evaluation is performed to determine the maturity of the SPL. Finally some improvements are performed over the STS-SPL.

In the present study, action research is carried out to assess and enhance, albeit in a restricted scope, the success of the SPL adoption effort described above. Success is evaluated by taking measurements and the results are discussed. It is observed that software product line adoption led to significant effort reduction, improved product quality, homogenous look-and-feel and improved reusability in the software development efforts of software testing department of ASELSAN REWIS division. Quantitative assessment of the benefits of SPL adoption shall be carried out as much as possible and together with qualitative evaluations of the involved staff. It must be stated explicitly that this is in agreement with the inherently qualitative nature of the work done, as rather than establishing an all-encompassing theory and asserting general truths, the present aim is restricted to exemplifying the benefits of SPL as achieved in a specific project carried out in a specific organization. That is the fundamental aim of the present study is to provide concrete and elaborated evidence on the costs and benefits of implementing an SPL in a specific organization.

The outline of this thesis is as follows: Chapter 2 provides an overview of the literature on action research methodology, SPL case studies and quality measurement. In chapter 3, an overview of the industrial setting is presented. Chapter 4 describes the ASELSAN STS-SPL and the improvement efforts. Chapter 5 presents the measurements and their results. Chapter 6 is an overview of the reported study and includes a general conclusion and discussion.

CHAPTER 2

LITERATURE OVERVIEW

In this chapter, first, the methodology of the present research shall be presented, with reference to its sources in the literature and with justification of its appropriateness for the case under focus. Second, a literature overview on SPL case studies shall be given. Reported benefits of SPL's are categorized under six topics. Finally, an appropriate quality model for the evaluation of the present study and quality measurement metrics shall be presented.

2.1 Action Research Methodology

Use of qualitative research methods in evaluation of computer systems and information technology is contemporarily increasing [5, 6]. Action research is a qualitative research approach initially used in the social and medical sciences in the mid-twentieth century. In the 1990s it was adopted for use in scholarly investigations in information systems [7]. In [8] Peter Reason and Hilary Bradbury define action research as: "Action research is a participatory, democratic process concerned with developing practical knowledge in the pursuit of worthwhile human purposes, grounded in a participatory worldview which we believe is emerging at this historical moment. It seeks to bring together action and reflection, theory and practice, in participation with others, in the pursuit of practical solutions to issues of pressing concern to people, and more generally the flourishing of individual persons and their communities." Sometimes an action researcher is referred to as participant observer because the observer also needs to participate in the setting in order to take action and change the pro-

cesses. In fact this setting precisely describes the present study.

According to action researchers, complex social systems cannot be reduced for meaningful study but can only be understood as whole entities. Thus, they investigate complex social processes by introducing changes into these processes and observing the effects of these changes [7].

Due to its interpretive and idiographic, that is concrete rather than abstract and theoretical nature, typically qualitative data is collected in action research. Thus, it is reasonable initially studying the qualitative research fundamentals and the differences between qualitative and quantitative research methodologies.

2.1.1 Qualitative Research Fundamentals

Qualitative research is an empirical research strategy that is conducted in natural settings using non-quantitative data instead of numbers. Qualitative data are generally collected from observations, interviews, and documents by investigating the behavior and reaction of people in particular situations within which they act. In qualitative methods data is gathered in the form of words such as transcripts of open-ended interviews, written observational descriptions of activities and conversations, and documents and other artifacts of people's actions. The textual essences of such data are preserved throughout the analysis in order to understand a phenomenon from the participants' perspective in its own environment which is not possible when textual data are quantified and aggregated [5].

2.1.1.1 Qualitative vs. Quantitative Research

Although qualitative or quantitative data may be collected in an empirical study, there is a strict distinction among them. Quantitative data concerns numbers and is analyzed using statistics; however qualitative data concerns words, descriptions, pictures, diagrams etc. and is analyzed using categorization and sorting [6].

Qualitative research is more useful when the purpose of the study is to examine the dynamics of a process rather than its static characteristics or the issues studied are not easily partitioned into discrete entities [5].

2.1.2 Action Research Approach

In action research approach, first a client-system infrastructure or research environment is established. The client-system infrastructure is the agreement that specifies the authority, or sanctions, that researchers should regard in specifying actions. Later, five distinct phases are iterated as follows: Diagnosing, action planning, action taking, evaluating and specify learning. The details of each phase are given in [7].

Being a qualitative research methodology, action researchers may use three main sources for data and four basic techniques of data analysis as outlined in Table 2.1 [5].

Table2.1: Qualitative Data Collection and Analysis Techniques

Sources for Data	**Data Analysis Techniques**
Observations, Open-ended interviews and survey questions, Documents and texts	Coding, Analytical memos, Displays, Contextual and narrative analysis

2.1.3 Other Related Methodologies

Case study which is also based on qualitative data is closely related to action research with its definition and purpose. Key difference between two is that a case study is completely observational but action research requires involvement in the change process. Action research is suggested to be adopted in software

process improvement and technology transfer situations. Case studies are preferable when the effect of a change is studied in situations like pre- and post-event studies [6].

Consulting is another related definition residing in research literature. Although action research processes and typical organizational consulting processes are substantially similar; there are five key differences among them [7].

1. Action research is motivated by scientific expectations; however consulting is motivated by commercial benefits.

2. Action researcher is responsible for providing scientific knowledge both for research community and client whereas consultant only has responsibility against client.

3. Collaboration is essential in action research but in consulting, an objective perspective on the organizational problems is provided.

4. In action research solutions are suggested according to a theoretical framework, but consultants use their own experiences.

5. In action research, organizational understanding is enabled by iterative experimental changes in the organization. However, in consultation, organizational understanding is enabled through independent analysis of the problems.

Experiment is another similar definition which is an essential research methodology in science. Controlled experiments are described as "measuring the effects of manipulating one variable on another variable" [6].

2.2 SPL Case Studies Overview

2.2.1 Easier and Improved Reuse of Software Assets

Reuse is the basic and crucial action of the SPL approach. However distinctively from the other reuse methods it is performed in a prescribed way. It is also referred to as development for reuse. Development for reuse provides a basis for the development of the products and this basis includes a variety of assets regarding the requirements, architecture, implementation and testing. [2].

The amount of reusability improves the benefits of the SPL effort in the meantime a successful SPL improves and encourages the reuse of the core assets. Usually reusability level is related to the compatibility of the software architecture with the current and planned products. SPL approach requires a reference architecture to support this ability. In [9, 10], the importance of the reference architecture is emphasized as a leading factor to high level reuse. AKVASmart [2], Nokia Networks [2] and U.S. Army Technical Applications Program Office (TAPO) [11] are some of the companies who have explicitly referred to easier and higher reusability as improvements to their software development process.

To achieve high level of reusability, compatibility with the existing products is important. In order to incorporate them to the product line usually an evolutionary approach is recommended instead of a revolutionary approach [3].

2.2.2 Cost and Effort Reduction

Majority of the SPL case studies reports improvements in the reduction of costs and effort. In software development, effort composes the majority of the overall cost and so it is closely related to cost metric. Large-scale software reuse enabled by SPL adoption is the main reason of effort reduction.

An upfront investment is required to instantiate SPL but case studies show that the investments made are worthwhile. For example, the Wikon GmbH case study shows that the break-even is as close as 4th product [12]. More generally,

the break-even point of return on investment to SPL is expected to be between the 2nd to the 8th product [2].

Successful SPL representatives declare reduction in the demand to staff resources which leads to an important cost improvement. U. S. Naval Undersea Warfare Center and Hewlett Packard report that, with the introduction of SPL, 25% of the total personnel may be sufficient to develop new products [13, 14]. In FISCAN's case, less than one-third of the software developers and testers are sufficient to finish all the work [15].

Development efficiency can be raised by enlarging the product portfolio. Because each product reuse the core assets, when the number of the participants increase the coding and testing costs of shared components stay in place and the overall cost is reduced. Dialect Solutions increases its development efficiency and overall productivity by enabling core assets to not branch for each product [16].

Asset scoping is an economic decision activity which may be an important factor in SPL success. Nokia Mobile Browsers binds its SPL success to well management of core assets: "To minimize costs, we avoided developing core assets that we would not reuse." [17]

Majority of the SPL case studies report general cost improvements acquired with the systematic reuse of the software assets. SPL adoption leads Boeing and DNV Software to reduced life cycle costs [2, 18]. Deutsche Bank AG established approximately 4 million dollars of direct annual cost savings per year [19]. E-COM Technology Ltd. [20], Nokia Mobile Browsers [17], Raytheon; U. S. National Reconnaissance Office [21], Bosch Gasoline Systems [2], Danfoss Drives [10] and U. S. Naval Undersea Warfare Center [14] are other companies those successfully establish SPL and benefit reduced cost. Philips Medical Systems reports 2-4 times effort reduction [2].

Productivity of software engineers is a common improvement reported in the case studies which indicates another reduction in the development effort and cost. CelsiusTech Systems reports extremely high productivity and extremely low cost with the adoption of SPL [22]. Cummins [9], E-COM Technology Ltd.

[20], Lucent Technologies [23], Siemens Healthcare [24] and Toshiba [25] state high productivity in related cases studies.

Other improvements like time-to-market, maintainability, integrability, modifiability etc. have implicit effects on cost metric. However some case studies measures and explicitly reports economic effects of these improvements. Rockwell Collins - Common Army Avionics System (CAAS) benefits from the improvements in integration, maintenance, documentation, training and testing costs [11]. U.S. Army Technical Applications Program Office (TAPO) reports improvements in the development, maintenance, integration and documentation costs [11]. Siemens, ABB and Market Maker Software AG benefit from the reduction of cost of quality while having significant quality improvements [13, 2]. Argon Engineering experiences decrease in the development, upgrade and technology costs [26].

2.2.3 Better Scheduling and Time-to-Market

SPL approach enables companies to develop and take their software products into market faster still conserving the quality. Nortel reports that they were able to reduce cycle time by 45 percent, without any quality payback [27]. This improvement is achieved by enabling delta engineering which is the action of producing new products by only giving development effort for the variation points[28]. Besides, if the required variant is available in the product line infrastructure, it is reused.

It is visible that there is a close relation between the amount of reuse and time-to-market. Overwatch Textron Systems estimates the amount of reuse range from 40-70% and time to market by a factor of approximately 2.5 [29]. CelsiusTech measures the reused common parts of their software products as on average 70% to 80% and reports a huge schedule reduction [30].

Actually most of the case studies report great time-to-market improvements along with their SPL experiences. Hewlett Packard's case study is interesting with the ability to increase time to market 2X while product complexity and the

number of products increasing 10X. They explain this success with the advance within their architecture and process which meets their business demands [31]. Other companies reporting time-to-market and development time improvements are: Argon Engineering [26], Asea Brown Boveri (ABB) [13], Cummins [9], DNV Software [2], E-COM Technology Ltd. [20], Fiscan [15], General Motors Powertrain [32], LSI Logic - Engenio Storage Group [33], Mondragón Sistemas de Información (MSI) [34], Nokia Mobile Browsers [17], ORisk Consulting [35], Philips- PKI telecommunications switching system [36], Philips Medical Systems [2], Raytheon; U. S. National Reconnaissance Office [21], Salion, Inc. [37], Siemens [2], TomTom Automotive [38], U.S. Army Technical Applications Program Office (TAPO) [11], U.S. Army [39], U. S. Naval Undersea Warfare Center [14] and Rockwell Collins - Common Army Avionics System (CAAS) [11].

As well as bringing the software products to the market faster, better scheduling is enabled with the SPL approach. This is mostly due to the less complexity of the systems with the introduction of the variability management and the visibility of the similarity between different products. Testo reports that they could finish all projects on time [40]. However, Market Maker Software AG's scheduling work caused time lacks because of their SPL setting to be more complex than traditional development methods. Thus, although they could reduce time to market 2-4 times than before; resolving issues took longer than their expectations [2]. Another risk to achieve time improvements is reported by AKVASmart. Their case study shows that when an evolutionary strategy is chosen, compatibility issues may make the development time much longer than expected [2].

2.2.4 Improved Software Quality

Quality of the reusable software assets is a decisive factor in the achievement of other improvements like time-to-market and development effort reduction because the quality problems in common assets can easily propagate into all products in a SPL [41]. Thus, achieving high quality level of end products is only possible by ensuring high quality components. However, a product line

approach usually ensures the quality of components implicitly by making reuse of them again and again leading software defects to emerge somehow when a proper SPL testing approach is followed. Thus, a high level of commonality is a factor of improved quality.

Quality improvement is one of the important motivations for SPL initiation for most of the companies. It is generally measured by the defect density rate. Hewlett Packard [13], Philips Medical Systems [2], ORisk Consulting [35] and Ericsson [42] reports significantly lower defect density compared to previous products.

AKVASmart showed that an independent plug-in architecture helps software quality with the easy deployment of bug fixes [2]. Dialect Solutions reports rapid improvements to product quality by developing defect fixes within core assets only once [16]. EIMW SPL's automated SPL process leads to assure product quality by avoiding human mistakes [43].

Other companies reporting improved reliability and code quality are CelsiusTech [22], Asea Brown Boveri (ABB) [13], Cummins [9], Dialect Solutions [16], DNV Software [2], Fiscan [15], Nokia Mobile Browsers [17], Raytheon; U. S. National Reconnaissance Office [21], Siemens Healthcare [24], Testo [40] and Wikon GmbH [12].

2.2.5 Product Diversity and Competitiveness

Today, software companies compete with each other to gain market dominance and SPL approach helps it with time-to-market, cost and quality improvements. Usually product diversity is needed to address requirements of different customers. High customizability and flexibility can be achieved by variability management in a SPL. Philips pays attention to variability management and satisfies the variability needs of marketing enabling a single software product line for a huge product portfolio of mid-range and high-end televisions [2]. Cummins [9], Telvent [2] and Testo [40] achieve to enter effectively to new markets with their SPL capabilities. Salion, Inc. benefits from the scalability of product portfolio

to increase competitiveness [37].

Flexibility of the reference architecture may be a leading factor for opportunistic reuse of SPL. CelsiusTech could easily expand its product portfolio by the flexibility of its architecture and product line those were originally developed for a different domain [30].

Actually there are numerous occasions of SPLs for increased product diversity and competitiveness. Cost and effort reduction enables business organizations to compete better in a market segment. HomeAway's SPL adoption leads to effort reduction enabling targeting new market segments faster, more efficiently, and with greater profitability [44]. EIWM product line represents the advantage of ease of customizability by the help of automation in SPL development [43]. ORisk Consulting emphasize the increases in productivity and implementation time as key factors to improve competitiveness [35]. Eurocopter follows a bottom up approach of introducing a SPL and succeeds to manage many software variants [45]. Also domain potential analysis is an important activity for increasing product diversity. It aims determining suitable domains to make investments for reuse [2].

Customer satisfaction is an important outcome that indicates high competitiveness. Cummins [9], CelsiusTech [22] and Siemens Metals Technologies [46] report improved customer satisfaction due to high responsiveness to customer needs. U. S. Naval Undersea Warfare Center also experience increasing customer satisfaction by covering customer requirements reliably and predictably[14]. Common-look-and feel with a high usability is an important factor for the customer satisfaction. AKVASmart [2], Siemens Healthcare [24] and Mondragón Sistemas de Información (MSI) [34] aimed and succeeded improved uniform look-and-feel in their products.

2.2.6 Painless Maintenance and Modifiability

Several experiences show that code size can be significantly reduced by eliminating duplicated copy-and-paste codes. SPL approach reduces the complexity

of the products by increasing the similarity between different systems with managed reuse [2]. Reduced complexity and decreased code size improvements of SPL leads to easier and cost effective maintenance, adaptability and modifiability of software products. There are plenty of case studies addressing this issue. Bosch Gasoline Systems [2] and Wikon GmbH [12] reduce maintenance efforts with SPL introduction. Less complexity leads HomeAway to more effective testing, radically reduced deployment times, and higher quality [47]. In LG's experience with the reduction of the complexity the maintenance costs are decreased too [13]. Danfoss Drives [10] and Ericsson-axe [48] reports adaptability improvements. Reduction in code size and cyclomatic complexity leads Ricoh significant maintainability and reusability improvements and fail-safe refactoring [49].

Reduction in the code size further leads to production and integration simplicity and developer satisfaction. AKVASmart ASA adopts SPL with an evolutionary strategy and reduces the code size by more than 70%. It is also reported that their integration and maintenance process is easier than before [2]. CelsiusTech emphasize integration simplicity [22]. Nokia [17] and Testo [40] reports increased developer satisfaction as a consequence of SPL adoption.

2.2.7 Summary

Most of the SPL case studies report cost, time-to-market and quality improvements as a result of systematic large-scale reuse. A full list of improvements achieved through case studies is given in Appendix A. Achieving most of these improvements may depend on conforming to some general tips:

• CelsiusTech carry out most of the important improvements of SPL adoption. Their success comes with a substantial upfront investment and regarding organizational and management issues beside the technical issues. CelsiusTech's high reuse ability came with their product line architecture and the applied design practices like information hiding and encapsulation [22].

• Danfoss Drives' mentions that slow and careful stepwise migration to SPL and

the simplicity of the environment and the processes are key points to achievement [50].

• Danfoss Drives and Market Maker Software AG case studies show that having a small team is helpful because of the communication and management issues [2, 10].

• DNV Software's key success factor is to have a long-term vision, strong management support and leadership and endurance in the organization [2].

• Mondragón Sistemas de Información (MSI) states the importance of the management's support and applying the adoption process progressively and iteratively [34].

2.3 Measuring Quality

2.3.1 Quality Model

Software product quality is a large and complex concept that is not easily manageable. Quality models provide a taxonomy that split the software product quality concept into smaller and more manageable parts. This decomposition facilitates the measurement of software product quality. ISO/IEC 25010 is a software product quality model standard that is released in 2011 by the International Organization for Standardization. It is the successor of ISO/IEC 9126 and the most well-known type of software quality models [51]. The model defines attributes, sub-characteristics and characteristics of software product quality. These definitions describe a hierarchical classification where characteristics at the highest level and attributes at the lowest level. The model is divided into two main parts: (1) Product quality model (2) Quality in use model. Product quality model is based on eight characteristics (functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability) and subdivided into sub-characteristics. Quality in use is defined as "the degree to which a product or system can be used by specific users to meet their needs to achieve specific goals with effectiveness, efficiency, freedom

from risk and satisfaction in specific contexts of use" [52]. This definition also gives the five characteristics of quality in use model.

ISO/IEC 25010 product quality model has 8 characteristics and 31 sub-characteristics and most of the definitions are not easily applicable in a particular domain [53]. In the domain of the present study, satisfaction, reliability and maintainability are relevant and measurable characteristics of the ISO/IEC 25010 model. Thus, these aspects are discussed in more details.

2.3.1.1 Customer Satisfaction

In [54], customer satisfaction is given as a success indicator for the measurement of software process improvement. It is further regarded as a software quality indicator. A change in product quality can be noticed by examining customer satisfaction. There are two methodologies to assess customer satisfaction. Qualitative customer satisfaction is generally appraised by questionnaires and quantitative customer satisfaction requires objective measures. Using questionnaires to measure customer satisfaction gives a better view on software quality; however it requires the involvement and cooperation of the customer.

ISO/IEC 25010 quality in use model defines satisfaction as a quality characteristic. Although product quality model describes the characteristics of the product, the quality in use model analyzes the characteristics of the interactions of users with the product. Thus, quality in use is usually associated with usability [51]. Measurement of usability should regard users' subjective reactions to system. Standardized questionnaires like SUS are useful to make usability assessments [55].

2.3.1.2 Reliability

Being a sub-characteristic of reliability in ISO/IEC 25010 model, fault tolerance is measured by defect rate [56]. Defect density rate is a common metric to measure software quality. The number of defects those emerge during the software

tests can be divided to project size to measure defect density rate and product quality. In software product lines literature software quality improvement is generally measured by defect density rate.

2.3.1.3 Maintainability

Maintainability is one of the most important characteristic of software quality because it typically comprises 50 percent of the total lifetime cost of a software system [57]. ISO/IEC 25010 model sub-characterizes maintainability as modularity, reusability, analyzability, modifiability and testability; however it is not easy to directly measure each attribute. A characteristic of highly maintainable software is given in [58] as being open to changes that means less effort requirement when adding new features. Thus, simply recording the time required to perform a maintenance task gives an idea about the maintainability of a software product.

2.3.2 Object Oriented Quality Metrics

Code-based metrics are used widely to measure and improve the software product quality. These metrics are collected by analyzing the source code to verify the quality of the software design. This leads developers to improve software quality and productivity. It further helps prediction of fault-proneness and early determination of unsafe components [56]. Most popular object oriented metrics defined in literature are Chidamber and Kemerer's (CK) metrics suite [56, 59]. A brief description of each CK metric is given in the subsections.

2.3.2.1 Weighted Methods per Class (WMC)

WMC is the sum of complexities of methods in a class. Cyclomatic complexity (CC) is a convenient method to be used in the WMC calculation. CC is the number of independent paths through the source code of a method. Also, the number of tests required for a software module is equal to the Cyclomatic com-

plexity of that module [60]. Usually a small value of WMC is wanted since it indicates complexity. WMC value is further correlated with the effort required to develop and maintain the class and the possibility of reuse [61].

2.3.2.2 Depth of Inheritance Tree (DIT)

DIT of a class is the maximum length from that node to the root in the inheritance tree[61]. The number of inherited methods increases with the depth of inheritance tree and it becomes harder to estimate its behavior. Also design complexity increases with the depth of inheritance tree. However, the chance of reuse is better with a deeper tree [56].

2.3.2.3 Number of Children (NOC)

NOC is the number of immediate subclasses of a class in the inheritance tree. Having a large number of children requires attention because it may indicate a misuse of sub classing and more test cases are required. However, when used carefully, a great number of children lead to improved reusability [61].

2.3.2.4 Coupling Between Object Classes (CBO)

CBO of a class is calculated by counting the number of other classes to which it is coupled [61]. Coupling affects modularity of the design negatively causing low reusability. Excessive coupling leads to difficult maintenance and testing of a class [56].

2.3.2.5 Response for a Class (RFC)

RFC of a class is the total number of methods that can potentially be invoked by that class. A large value of RFC metric increases the complexity of a class which leads to complicated and difficult testing and debugging [62].

2.3.2.6 Lack of Cohesion of Methods (LCOM)

LCOM of a class is calculated by subtracting the number of method pairs whose similarity is not zero from those having zero similarity. Similarity of two classes is defined as the number of elements in the intersection set of the instance variables used by them [61]. Creating cohesive methods is always advised thus a smaller LCOM value is desirable. Classes with larger LCOM value should probably be split into several subclasses. Larger LCOM values also indicate higher complexity and low reusability of a class [56].

CHAPTER 3

OVERVIEW OF THE INDUSTRIAL SETTING

3.1 Preparation

3.1.1 Conducting Action Research

Action research is accepted as an appropriate approach for this thesis study because of the main characteristics of the domain. In [7] ideal domain of action research methodology is described as follows:

1. The research is expected to be beneficial for both researcher and organization.

2. The researcher is actively involved in the change process and can apply the obtained knowledge immediately.

3. The research process comprises both theory and practice.

Since the author of the present study is actively working in the organization where the research is intended to be conducted, the requirement of active involvement is satisfied. Organizational benefit of the research is expected to be the improvement in software development efforts of software testing department. The foundation of this improvement will be based on software product line approach that constitutes the theoretical aspect of the process.

As a result of action research, mostly qualitative data will be collected and

evaluated. Actually, qualitative methodology is appropriate for this study because of the following reasons [5]:

1. It is not very easy to measure the subject of study. Some real software should be developed in order to see the improvement in software development which is a labor-intensive process. Lack of sufficient sample counts prohibits quantitative methods to yield definitive results. In fact this is the fundamental reason for referring to the present study as action research. At this point it is considered beneficial to repeat what was stated in Chapter 1: Quantitative assessment of the benefits of SPL adoption shall be carried out as much as possible. But this does not change the inherently qualitative nature of the work done, as rather than establishing an all-encompassing theory and asserting general truths, the present aim is restricted to exemplifying the benefits of SPL as achieved in a specific project carried out in a specific organization.

2. The research targets the challenges that people encounter and complain about. Thus, people's reaction to the change that is handled during the research is important. Qualitative methods are helpful in understanding how and why people think or feel about something.

3. Qualitative methods support materializing the effects of social, organizational, and cultural environment on the inspected situation.

4. The process cannot be regarded as a black-box that is researched with its outcomes or impacts. It needs to be developed and changed during the research which is also a reason for qualitative research.

3.1.1.1 Ensuring Validity

Since the nature of data collection and analysis in qualitative research is subjective, the consideration of validity gains importance. Interests, perceptions, observations, knowledge, and critical faculties of the evaluator play a role in the study. In order to enlighten the potential influence on study results, evaluators

should report their backgrounds in the research documents [5]. The author of this thesis study is an electrical and electronics engineering graduate and has more than 3 years of software testing experience within the organization where the action research was conducted. He has developed over 20 software testing simulators in 4 different projects and has used these products for testing embedded software.

Five other strategies are suggested in [5] to further ensure validity. These are, collecting rich data, paying attention to puzzles, triangulation, feedback or member checking and searching for discrepant evidence and negative cases. Four different types of triangulation are described in [6]. These are data (source) triangulation, observer triangulation, methodological triangulation and theory triangulation.

Rich data are detailed and varied data those enlighten the whole situation. It prevents researcher from seeing only clues those support his or her prejudices and expectations [5]. In the present study, measurements regarding implementation, maintenance and testing efforts, quality and usability are collected to see what software product line causes in software development. So, besides its improvements, negative characteristic of such an approach will be visible. This study further utilizes data triangulation in some measurements to increase validity. Same data is collected in many different development projects to increase robustness. Finally, discrepant data and negative cases are sought in the study in order to figure out important defects that forces particular interpretations or explanations.

3.1.2 Outline of the Study

Basically, any action research practice comprises the actions of diagnosing and treatment. More particularly these actions are defined in a five stage process (diagnosing, action planning, action taking, evaluating and specify learning) which begins after the establishment of client-system infrastructure. This thesis study generally adheres to the action research approach and the aforementioned stages

are performed thoroughly [7].

First the client-system infrastructure is established informally with the management. The management was persuaded with the expectation of improvement in software development of software testing products. Author is well accepted as a researcher. Other employees of software testing department were employed as the participants of the research with a limited involvement. They participated in interviews and small sized experiments.

In the diagnosis stage, problems within the software development efforts were determined and these challenges specified the source of the organization's motivation for change. These problems are described in Section 3.3. Later, for the action planning stage researchers and practitioners collaboratively specified the actions to perform in order to solve the primary problems. The action concerned the adoption of software product line approach in software development which was a tremendous engagement that also includes some risks. Fortunately, a legacy-system COBALT was suitable for composing the reusable asset base and reference architecture. The change is expected to lead to a future state of improved development with fewer challenges.

After performing the actions described in Chapter 4 in detail, results of the qualitative investigations are evaluated. Details of the collected data are presented in Chapter 5 and they are evaluated in Chapter 6.

The knowledge gained by the organization through the investigation of changing the process will lead to the restructuring of organizational norms.

3.2 REWIS Software Testing Process

3.2.1 Overview

ASELSAN is the leading defense industry company of Turkey developing state of the art land, air, naval and space electronic systems and products. Radar, Electronic Warfare and Intelligence Systems (REWIS) division of ASELSAN designs, develops and produces high technology military radar systems, Electronic Warfare Self Protection Systems, Electronic Support and Attack Systems and their sub-systems [63]. These products are developed according to customer requirements and usually composed of several sub-systems including complicated hardware and software components. Software components of the systems are called computer software configuration item (CSCI) and hardware components are called the hardware configuration item (HWCI) [64]. CSCIs vary from embedded software running on real time operating systems to JAVA applications including complex graphical user interfaces.

Software is an important part of products in REWIS. Software development is carried out by 65-70 developers. Software testing is performed by an independent group of software test engineers. Testers constitute a smaller part of the organization with 9 staffs. Each REWIS product is required to be tested carefully to ensure high quality and customer satisfaction. Because of the product diversity and the high testing effort costs of their complicated structures, a need for continuous technology improvements in software testing emerges. Next section describes the method and scope of the REWIS software testing efforts. Later, the encountered problems with the software testing are discussed.

3.2.2 Test Method and Scope

REWIS Test Engineering Department performs functional requirement based software tests which is called black-box testing. Unlike structural testing (white-box), in black-box testing the specified behavior of the software is tested without any knowledge about the program code. This method is also called as specifica-

tion based, behavioral, and responsibility-based testing [65].

REWIS uses V development process model in software development. As seen in a simplified model shown in 3.1, it defines test levels by determining which test activity to perform at each development step [13].

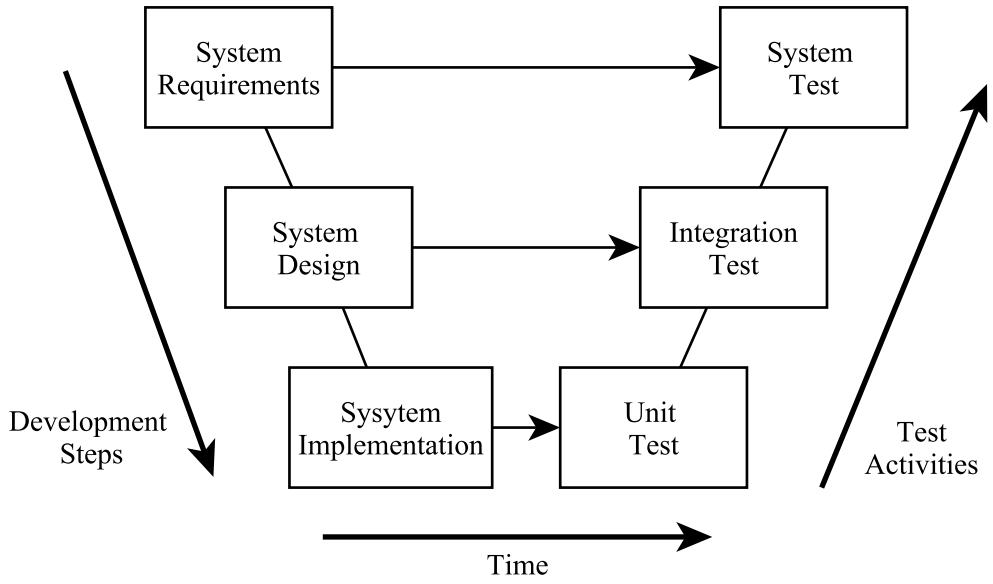

Figure 3.1: V-model [13]

In V-model main test levels are unit test, integration test and system test.

Unit Test: A component, method, or class maybe the unit under test. Actually each implemented unit should be unit tested. The unit test validates a unit against its specified requirement which is the input/output behavior of that unit. In REWIS, the programmer of the unit performs the unit test.

Integration Test: Units that have passed the unit test, constitute a configuration which is specified in the architecture. The integration test validates the behavior of these units when they come together according to the reference architecture.

System Test: The system test is performed in order to validate the behavior of the system according to the system requirements specification. REWIS test engineering department performs the system level tests of the software products.

3.2.3 Software Development in the Software Testing Process

3.2.3.1 Software Testing Simulator

A software testing simulator is the software that is developed and used for the testing of other software. Software requirements of the REWIS products usually comprise electrical messaging scenarios which is only visible by creating and using simulators of other units. These messages are transmitted between different CSCI and HWCIs via interfaces like serial channel, Ethernet, PCI-Express and MIL-STD-1553. According to the configuration and state of the unit, the transmitted messages and their parameters change. The simulators are developed by software test engineers to control the messaging to verify these kinds of software requirements. An overview of the software testing simulators is given in Figure 3.2.

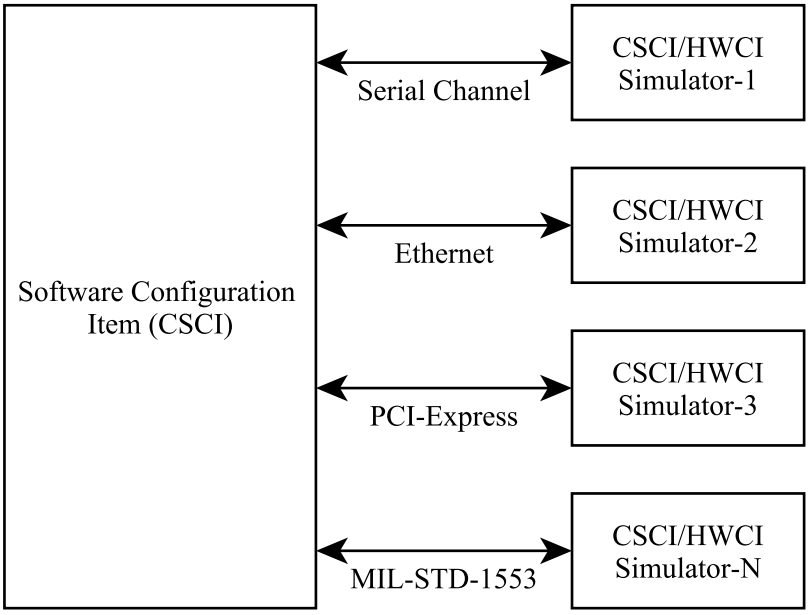

Figure 3.2: Software testing simulators overview

3.2.3.2 The Effort Required for Simulator Development

In general software testing efforts includes test case derivation from requirements, simulator development, performing the tests and generating the test reports. Simulator development constitutes a significant part of the overall effort. Simulator development covers the complete steps of software development process as requirements specification, design, implementation, testing and maintenance. The requirements of the software under test and communication interface specifications determine the requirements of the simulators. After the implementation of the simulator according to a convenient design, the simulator is also tested with the black box approach. And as in any software development process maintenance is required to keep simulator up-to-date. Furthermore, usually testing of a CSCI requires more than one simulator to be developed.

3.2.3.3 Software Testing Simulators Product Portfolio

In general, software testing simulators product portfolio includes products that do the following:

- Send and receive messages from a communication interface
- Record the properties and parameters of the received and sent messages in a specified location
- Include a user interface to change the parameters of the messages to be sent
- Display the sent and received messages

The type of communication interface and the real-time working specifications create a variation in the type of the simulators. The simulators are developed with different programming languages (Java, C#, C++) and for different operating systems (Windows, VxWorks) and platforms (Embedded systems, PC). When a graphical user interface (GUI) is needed or the messaging doesn't need to be real-time; a PC simulator is developed with Java or C# according to developer's preference. However, when high frequency messaging in orders of millisecond is required, then C++ is used to develop an embedded simulator for real-time operating systems. Sometimes the type of the implemented interface

requires embedded programming (e.g. PCIe, message queue). Also, when test automation is in question, the simulator needs to be compatible with the test automation tools. Actually, a full set of requirements for an individual simulator product is determined being specific to project that the simulator is used for.

3.3 Problems with Simulator Development Efforts

Previously, traditional single system development (SSD) approach was used in software development. In this approach each product is developed separately with an opportunistic reuse method.

The following problems and challenges were seen in the context of test engineering department's simulator development efforts. This observation is based on three years software test engineering experience of the author and well accepted by department management.

3.3.1 Effort Overload

The similarity between different products reveals the consideration of effort overload in many aspects. Redundant effort is spent for the design, implementation, testing and maintenance of the software those have very similar requirements but discrete development life-cycles like single system development.

Single system approach leads to diverse technological platform and code style where each developer works independent of each other. After some period teamwork of simulator development becomes troublesome. Also the maintenance of the software becomes very hard to be performed by a different engineer than its developer.

Besides, difficulty of the new technology insertion becomes troublesome with the conventional development methods. Diverse technological platform and code style causes to redundant effort by developing the same new features separately for different products.

Effort assessment

Measuring the effort spent for the development of a software asset is relatively simple. Time required to create source codes, perform a maintenance task and execute the software tests shall be measured in units of man-hour.

3.3.2 Demand for Higher Product Quality

Higher product quality is essential because final products are used for the testing of the other software products. When an incompatibility is identified in the tests, it is probable to have originated from the testing software. Reporting these kinds of defects in the software leads to redundant effort both for developers and testers. Thus, defects in the simulators lead to delays in the overall development process.

Quality assessment

Defect density rate is a metric to measure software quality. The number of defects in the simulators those emerge during the simulator tests can be divided to project size to measure defect density rate and product quality. Another quality indicator is customer satisfaction that shall be evaluated by interviews and questionnaires. Furthermore, Chidamber and Kemerer's metrics suite shall be used to measure software product quality. Since CK metrics are code based, they shall be measured by using a tool that analyzes the compiled bytecode files. Finally, in order to assess the maintainability of the software, maintenance effort shall be utilized.

3.3.3 Heterogeneous Look-and-Feel

User interface consistency is an important factor of usability of developed simulators. There is a demand for the high usability of the simulators and this can be ensured by a common and well accepted graphical user interface. However

it is not ensured with conventional methods. Not always the developer of the simulator uses it for the software testes but sometimes other test engineers and even system and software engineers need these simulators. Then usability and homogeneous look-and-feel gain importance.

Usability assessment

It is not easy to measure usability property; however standardized questionnaires like SUS [55] are useful to make usability assessments. Also, interviews and questionnaires those are specific to the current setting shall be applied to provide quantitative data for the research.

3.3.4 Low Reusability

Reuse level is not very high with the traditional development methods. Conventional reuse efforts are limited to reuse of source codes by copy-pasting in a non-systematic way. However, requirement, code and test assets for the simulators are estimated to have high potential for large-scale reuse.

Reuse assessment

Reuse rate is the size of reused assets divided by the total assets. Reuse rate can be measured in the requirements, code and test assets. When source code level reuse is the subject, the most suggested size measurement metric is non-comment source line of code (SLOC) [56]. Source code level reuse shall measured by dividing reused SLOC by total SLOC.

Similarly, requirements level reuse shall be measured by dividing reused requirement count by total requirements count.

Finally, test level reuse shall be measured by dividing reused test case count by total test case count.

3.3.5 Summary

Problems with the simulator development efforts of test engineering department are determined as effort overload, demand for higher product quality, heterogeneous look-and-feel and low reusability. As given in Chapter 2 - Literature Review, software product lines enable improvements in quality, usability, reusability and effort reduction areas. A series of measurements will be taken in order to show the amount of improvements enabled by adopting SPL in the context of this thesis work. Table 3.1 gives the problems and related metrics to be measured for examining the consequences of the SPL adoption. Details of how and under which conditions the measurements are taken and their results are given in Chapter 5.

Table3.1: Problems with the Simulator Development Efforts and Related Metrics

Problem	**Metric**
Effort Overload	Implementation effort Maintenance effort Testing effort
Demand for higher product quality	Defect density rate Average LCOM Average CBO Average DIT Average NOC Average WMC Maintenance effort Customer satisfaction
Heterogeneous look-and-feel	Customer satisfaction
Low reusability	Source code reuse rate Requirements reuse rate Test case reuse rate

CHAPTER 4

IMPROVING SOFTWARE DEVELOPMENT

The problems listed in the Section 3.3 are the main reasons for SPL adoption in software testing. In the past, some improvement efforts for the simulator development have been carried out. In the next section, these efforts will be explained. Later, software testing simulators software product line will be defined in details. Finally, some improvements will be applied to the STS-SPL.

4.1 Previous Improvement Efforts

Previously an infrastructure for the simulator development has been established. It is called COBALT (Communication Basics Library for Testing) framework and used in over 50 simulator development projects. The main motivation for the development of the COBALT framework was to increase the reuse level of common software assets in individual products. Thus, it provided most of the reusable software assets and reference architecture of the STS-SPL.

4.1.1 COBALT Framework Overview

COBALT framework is an infrastructure that has been locally developed in ASELSAN, not formally documented, and is used in the development of simulators. This is the legacy system that constitutes the basis of the software testing simulators software product line. It is mainly composed of three main abstract parts:

- GUI components (CNodeExplorer, LoggerTable)
- Data parser/generator
- Data tree representation

An important task of the simulators is to send and receive messages from a specific communication interface. The structure of the messages is different for each product. COBALT represents messages with data trees in a composite structure. Developer generates the message classes by using the infrastructure and the infrastructure carries out the display and serialization/deserialization tasks. COBALT takes the communication library's output and input streams and enables the transfer of messages over this channel. Data parser reads the bytes from the input stream and generates the message objects. Data generator transforms the message object into a byte array and sends it over the output stream. CNodeExplorer displays the contents of a message with data tree representation. It also enables user to change any parameter of that message on runtime. LoggerTable is a list that is used to display the sent and received message's name, source and time stamp.

COBALT framework was developed for traditional opportunistic reuse in simulator development and began to be used by a few staff. Later, COBALT users increased and it is used widely by many software testers. We believe that it increased the software reuse level and development efficiency but there are no qualitative measurements to prove that. With the adoption of an SPL approach the improvements are assumed to be increase and the problems depicted in Section 3.3 are assumed to be addressed.

In the next section, usage of COBALT framework in software development will be extended to an SPL approach and will be defined in details.

4.2 Introduction to Software Testing Simulators Software Product Line (STS-SPL)

This section describes the SPL adoption process by extending the existing infrastructure to be coherent with the SPL approach. COBALT framework already includes a business-centric, reference architecture and two-life-cycle approach but these aspects haven't been defined explicitly so far.

Fundamental concepts of product line engineering are: variability management, business-centric development, architectural-centric development and two-life cycle approach [2]. There exist a reference architecture reflecting the generic design of the simulators, common software assets and a development approach similar to SPL's two-life cycle approach. However, variability is not managed explicitly.

4.2.1 Business-Centric

4.2.1.1 Short and Long Term Strategy of the Business

The characteristic of the market in this work is so different than the other examples described in the literature. Software developers are also the customer of the product. The market and the product portfolio are well defined because the requirements and the needs of the customer is determined by the developer. On the other hand, although there is not any real competition for the market, developers should compete with themselves to overcome the demand for producing higher quality products in less time. With the improved simulator development, overall software testing can be performed more efficiently and effectively.

The domain is quite suitable for SPL adoption because of the product variability and the commonalities among different products. Short time strategy of the business is to adopt SPL approach for the product portfolio described in the next section. In long term, applying continuous domain potential analysis, new domains will be searched and majority of the simulators used for software testing efforts will be developed by SPL infrastructure.

4.2.1.2 SPL Product Portfolio

General product portfolio of the simulators required for the software testing efforts is quite wide as described in Section 3.2.3.3. However SPL adoption will cover a subset of this portfolio. SPL product portfolio is restricted to Java products.

4.2.2 Software Testing Simulators Software Product Line Requirements Specification

STS-SPL requirements are given in Table 4.1. In order to represent variability in requirements, labels are attached on each entry. These labels specify the variation dependencies and constraint dependencies.

4.2.2.1 Variability Dependencies

Meanings of the variability dependency labels are as follows:

`<Mandatory>`: Mandatory requirement that has to reside in every product

`<VP(X)_Option(N)>`: N. option of variation point X. It can be reused in applications optionally.

`<VP(X)_Alternative(N)>`: N. alternative of variation point X. Application engineer chooses one of the alternative variants.

4.2.2.2 Constraint Dependencies

Meanings of the constraint dependency labels are as follows:

`<VP(X)_Option(N):Requires_VP(Y)_Option(M)>`: If there are constraint dependencies they are stated after the variability dependencies. This label states that N. option of variation point X requires M. option of variation point Y.

`<VP(X)_Option(N):Excludes_VP(Y)_Option(M)>`: This label states that N.

option of variation point X excludes M. option of variation point Y.

Table4.1: STS SPL Requirements Specification

Id	**Requirement**	**Test Level**
	Data Tree Representation Requirements	
SRS-1	`<VP1_Option1>` Simulator shall receive TBD (to be defined) message over the communication interface.	System
SRS-2	`<VP1_Option2>` Simulator shall send TBD message over the communication interface.	System
	GUI Requirements	
SRS-3	`<VP2_Option1>` STS-SPL shall include a tree view that displays all of the fields of the message classes. *Tree view shall be a GUI component that is developed with Java Swing toolkit.*	System
SRS-4	`<VP2_Option1:Requires_VP1_Option2>` Tree view shall let the user of the simulator to change the parameters of the displayed message to be sent on runtime.	System
SRS-5	`<VP3_Option1>` STS-SPL shall include a logger table that lists the sent and received messages. *Logger table shall be a GUI component that is developed with Java Swing toolkit.*	System
SRS-6	`<VP3_Option1>` The logger table shall include time stamp, name and source columns. *An object shall be hold in the memory for each of the rows of the table.*	System
SRS-7	`<VP3_Option1:Requires_VP2_Option1>` If the related object of a row is a message class, it shall be displayed on any tree view when the row is clicked by the user.	System

This set of requirements will be extended with asset scoping efforts that aim to determine which components must be built for reuse [2]. Details are described in Section 4.3.1.

4.2.3 Reference Architecture of the Legacy System

COBALT framework has a reference architecture that application engineers should derive in order to generate application architecture. Architecture is organized under four layers being similar to the MVC architecture [66]. Each abstraction layer hides the implementation details from the others. COBALT includes the components those are included in the user interface and data layers. Components under the communication and controller layers are developed being product-specific. The Figure 4.1, prepared by the present author as a first step towards documenting the hitherto undocumented COBALT architecture, shows an overview of the architecture. The arrows indicate the data flow direction.

On the communication layer, communication standards like Ethernet, serial channel and MIL-STD-1553 are implemented. The data transfer between the data layer is enabled with the input and output streams. Data layer gets the input and output streams from the communication layer and sends messages regardless of the implementation details of the communication layer.

Message classes are implemented in the data layer. The structures of the messages vary in every simulator so the exact definition of the message classes are done by application engineers. Data tree representation (DTR) enables message classes to be represented in a composite structure. When the user defines message classes with the DTR, the message parser and generator functions are adopted automatically. When CSAbstractComposite class is inherited, subclasses become serializable. To enable this, application engineer composes the instance variables of the message class with the COBALT wrapper classes and other CSAbstractComposite classes. Here, the use of inheritance adaptation mechanism leads to a variation point.

COBALT includes the logger table and tree view components at the user inter-

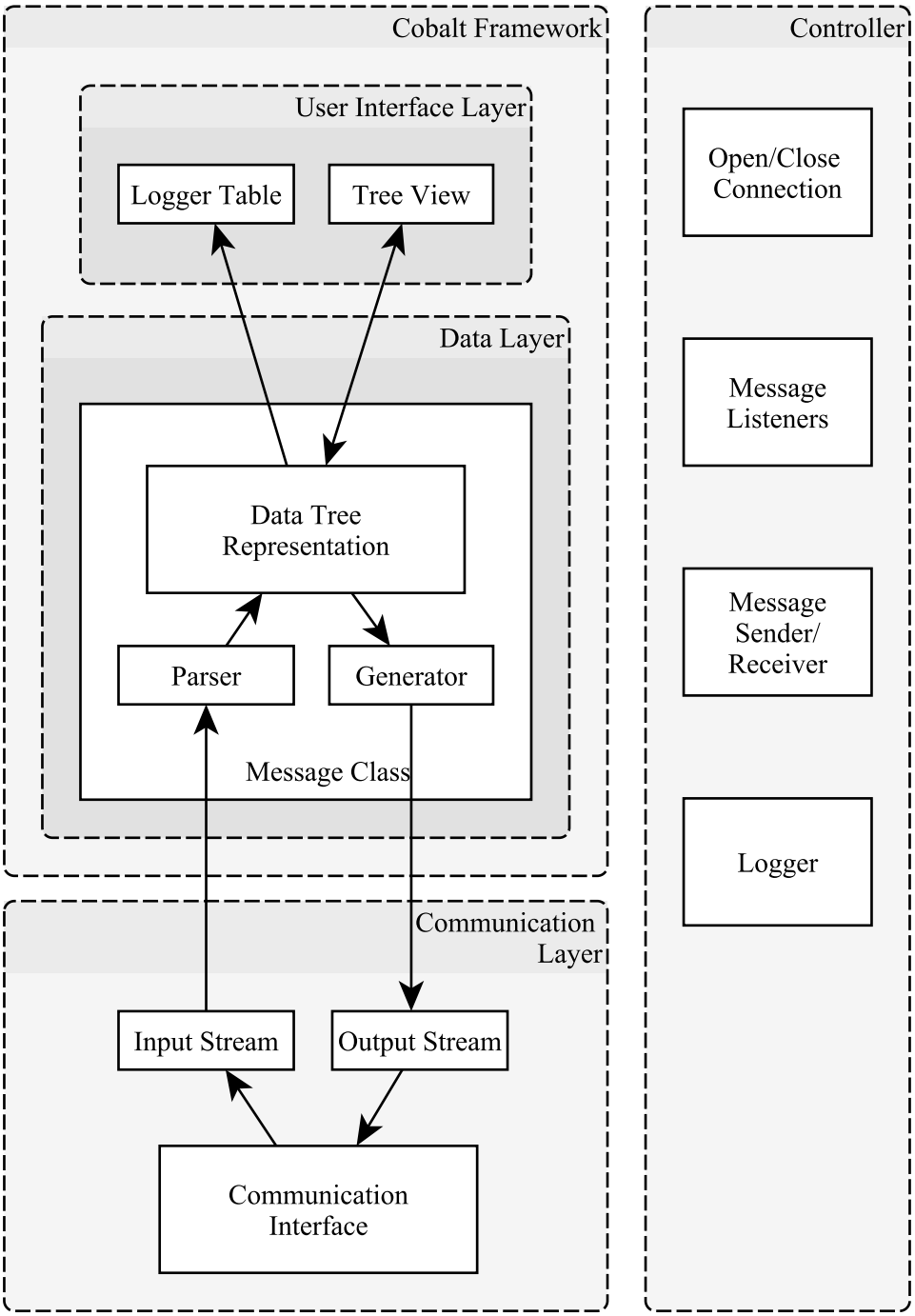

Figure 4.1: An overview of the legacy STS-SPL architecture

face level. These components are developed with the Java Swing library and can be reused at the development of GUI. Tree view enables data tree representation to be viewed and changed by the user at the runtime. Logger table is a table component that stores the sent and received messages. Each logger table can

hold a reference to a related tree view. When the user clicks a row on the logger table, the related message is displayed on the tree view.

The controller layer is similar to the MVC architecture's controller. It changes the state of the data layer and view's presentation of the model. It starts and stops the communication, includes the message listeners and controls functions like logging. Controller is developed by application engineers.

4.2.3.1 Variability Model

Explicitly managed variability is an essential concept of any SPL effort. The reference architecture of the STS-SPL is compatible with all of the variation points. STS-SPL requirements are also created being competible with the variation points described here. Variability is represented in the requirements with variability and constraint dependency labels. There are three variation points represented in the requirements.

Normally a specific product includes requirements like "The Simulator shall send and receive Parameters Message to/from the communication interface". In domain requirements, one choice is to include a requirement for each of the possible messages and make these requirements optional. However this is impossible because of the infinite number of possible message configurations. Thus, SRS-1 and SRS-2 cover all of these requirements by specifying a generic message name TBD. This results in variation point 1 as shown in 4.2.

Application engineers develop each variant (message classes) by inheriting the DTR infrastructure of COBALT. GUI requirements are optional, because if the test process is automated, GUI may not be used. For instance when Fitnesse tool is used to automate the testing, HTML pages replace the GUI. Simulator code and HTML pages communicate via fixtures which have to be product specific. Tree view component is the only optional variant for the variation point-2 (Message display by) and logger table is the only optional variant for the the variation point-3 (Log display by).

There are different approaches for representing variability. Applying the graph-

ical notation described in [13] the Figure 4.2, 4.3 and 4.4 are derived:

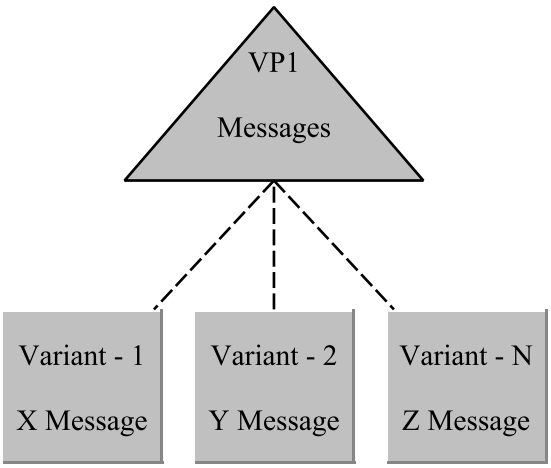

Figure 4.2: Variation point 1 - Message classes

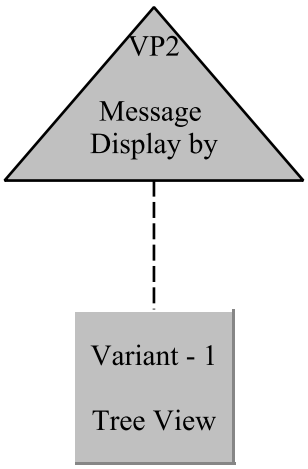

Figure 4.3: Variation point 2 - Message display by

4.2.4 Two-Life-Cycle Approach and Organization

Software development process of test engineering department is similar to the SPL's process approach. There are separate life-cycles of domain and application engineering which is a key characteristic of the SPL engineering. Domain engineering is responsible of the development of reusable assets organized under

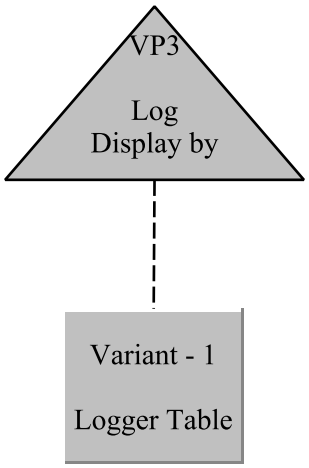

Figure 4.4: Variation point 3 - Log display by

COBALT framework. Domain requirements engineering, domain design, domain realization and domain testing is performed by domain engineers. Application engineering focuses on the development of the final products and gives feedback to domain engineering. They perform application requirements engineering, design, realization and testing activities.

Software test engineering department in ASELSAN consist of 9 staff comprising the entire SPL organization. The organizational structure is product-oriented where organization is distributed over domain and application engineering units [2]. There is a domain engineering unit and several application engineering units. Because the organization is very small, usually all application engineering activities of a product are performed by a single application engineer who is regarded as the customer of the Domain Engineering Unit for this specific product. After the development of the simulator, it is used for the software testing of a CSCI which is another software. All domain engineering activities including product and asset management are performed by a small group of domain engineers and there is not a clear distinction of responsibilities among them. Actually domain engineers also need to perform application engineering activities and thereby become the customer of the product which helps them to quickly respond to the needs of domain and customer. There is not any communication based problem which is usually encountered in large organizations. Figure 4.5 shows the roles

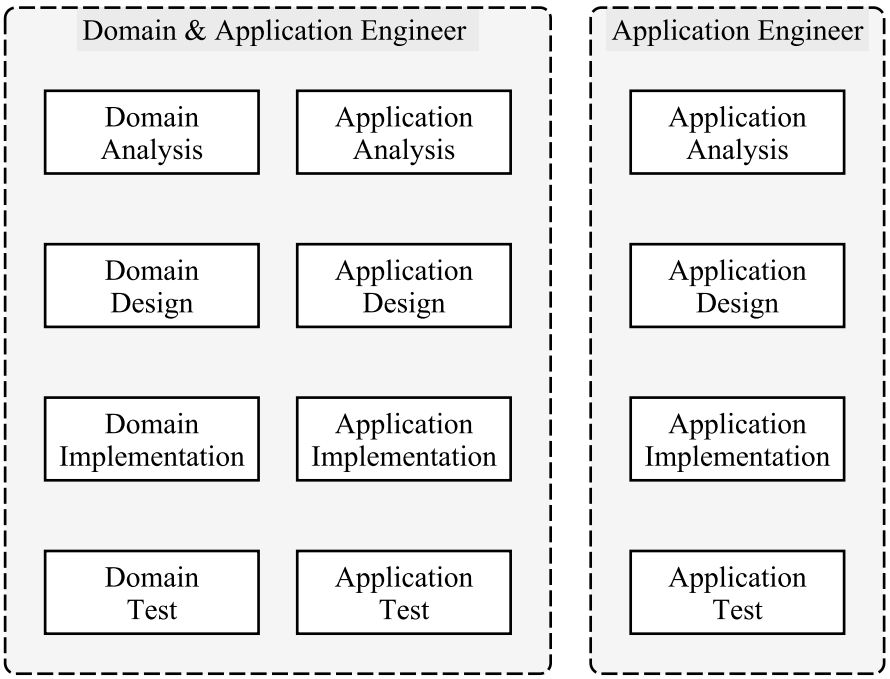

Figure 4.5: Roles and responsibilities

and their responsibilities in the organization.

4.2.5 FEF Evaluation

Up to now, an SPL initiation is enabled and FEF evaluation will be used to find out the maturity of the SPL and what have to be done to improve it.

The purpose of the Family Evaluation Framework (FEF) is to assess the effectiveness of the software product line engineering efforts of companies. There are four concerns of FEF evaluation: Business, architecture, organization and process. There are certain evaluation aspects for each dimension [67]. Business, architecture and organisation dimensions are evaluated in the context of this thesis work by the author and REWIS Software testing team leader approved the evaluation results. In 2012, ASELSAN was evaluated at CMMI level 3 and this information is referenced in the process dimension of the evaluation.

FEF evaluation result is a profile that includes four values for each one of the BAPO dimensions. Each evaluation value is formed in five levels determining

the maturity of related dimension.

4.2.5.1 Business Dimension – Level 1

Business dimension of FEF evaluation deals with the business issues that are unique to SPL organizations including investment decisions, measuring the costs and profits of SPL engineering and funding of domain engineering [2].

Sales, marketing, product management involvement. This aspect measures the level of involvement that marketing, sales and product management perform in SPL. In our case software test engineering team constitutes the entire SPL organization. Thus, there are not sales or marketing departments in the organization. Management is not aware of the opportunities of SPL.

This aspect is evaluated at level 1: Project based.

Budgeting and investment. There aren't specific budgets for domain and application engineering. Also there isn't any investment to generate a reusable asset repository.

This aspect is evaluated at level 1: Project based.

Vision and business objectives. Top management encourages any reuse and technology improvement efforts. Development of COBALT framework was also supported but hasn't been incorporated in the organization's vision and business objectives. There is not a defined SPL strategy of the organization.

This aspect is evaluated at level 2: Aware.

Strategic planning. There are not roadmaps for domain and application engineering. Application engineering uses the outcome of domain engineering in an opportunistic way.

This aspect is evaluated at level 2: Aware.

Combining the results, software testing department is evaluated at level 1 (project based) in the business dimension.

4.2.5.2 Architecture Dimension – Level 3

The Architecture dimension deals with the technical aspects of the product line development.

Asset reuse level. There is a collection of common assets but there is a lack of variants for the variable parts.

This aspect is evaluated at level 3: Software Platform.

Software product family architecture. There is a reference product line architecture containing common rules, and determining the use of the common platform. Reference architecture and components are developed in-house.

This aspect is evaluated at level 3: Software Platform.

Variability management. The reference architecture incorporates explicit variation points but doesn't determine rules for them to obey.

This aspect is evaluated at level 3: Software Platform.

Combining the results, software testing department is evaluated at level 3 (Software Platform) in the architecture dimension.

4.2.5.3 Process Dimension– Level 3

ASELSAN is at CMMI level 3.

4.2.5.4 Organization Dimension– Level 2

The organization dimension deals with the how effective the distribution of domain and application engineering over the organization performed.

Roles and responsibilities. Domain and application engineering roles are apart. However, major role in software development belongs to application engineering and there are not coordination roles between domain and application engineering.

This aspect is evaluated at level 3: Weakly connected.

Structure. Domain and application engineering have project-oriented structure and there is a separate domain-engineering project (COBALT framework).

This aspect is evaluated at level 3: Weakly connected.

Collaboration schemes. Collaboration is not document based but based on negotiations and information sharing.

This aspect is evaluated at level 2: Reuse.

Combining the results, software testing department is evaluated at level 2 (Reuse) in the organization dimension.

4.2.5.5 Results

REWIS test engineering department FEF evaluation results are shown in Figure 4.6:

4.3 Improving Software Testing Simulators Software Product Line

The improvement efforts described under this section have been undertaken exclusively by the author in the context of the present thesis work. The results

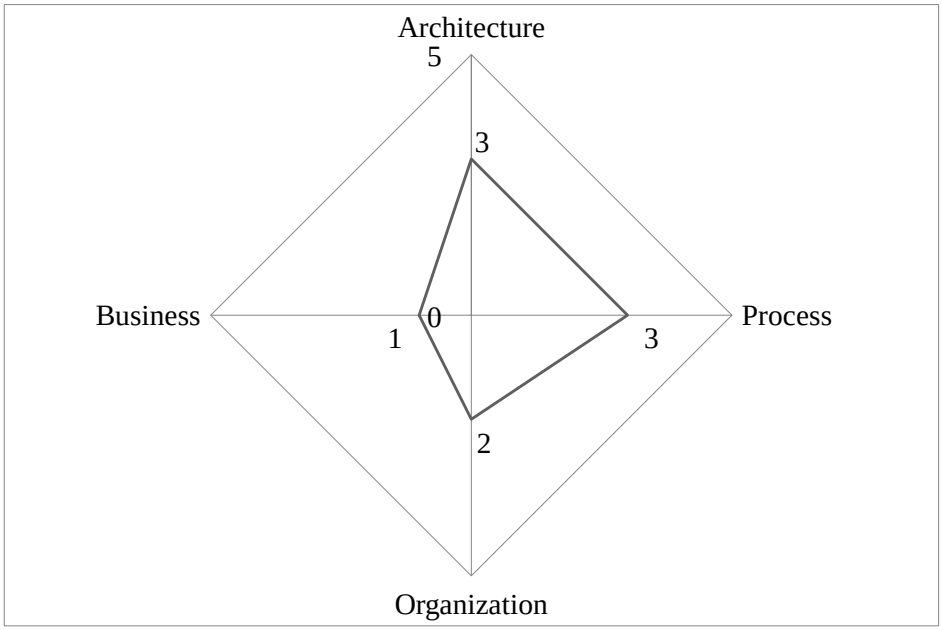

Figure 4.6: FEF evaluation results

of the overall SPL adoption process will be measured in Chapter 5 and evaluated in Chapter 6.

4.3.1 Asset Scoping

Legacy system was mainly focused on the commonalities between products. Asset base mostly included most common components and the number of variants wasn't satisfactory. With the asset scoping activity, the number of components included in the product line infrastructure is intended to be increased. Besides the common assets, the number of the variants is increased with asset scoping. As a rule of thumb, the feature that is expected to be used in at least 3 different products is included in the product line infrastructure.

In this context, two variations of the component that implements the communication interface is developed. These are the serial and Ethernet communication

components. Application engineer chooses and reuses the components by inheritance. This leads to a variation point of communication type as depicted in Figure 4.7: Two requirements are added with the arrival of the new variation

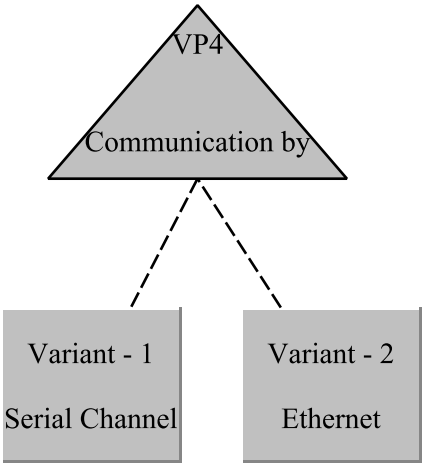

Figure 4.7: Variation point 4 - Communication by

point. Additional requirements SRS-8 and SRS-9 are given in Table 4.2.

Table4.2: Additional Requirements Related with the Variation Point-4

Id	**Requirement**	**Test Level**
	Communication Requirements	
SRS-8	<VP4_Alternative1> The messages shall be sent and received over serial channel interface.	System
SRS-9	<VP4_Alternative2> The messages shall be sent and received over Ethernet interface.	System

4.3.2 Creating Test Assets

The variability explicitly defined in the requirements is used to create system level test asset base to be reused for the application testing.

All system tests are performed by application engineers because the number of product variants is so much. Application engineers should derive application test case scenarios from the domain test case scenarios. Also other simulators may be required to verify the messaging requirements of the tested simulator. The following domain test cases are derived from the SPL requirements:

Domain Test Case Scenario -1

Addressed Requirements: SRS-1, SRS-3, SRS-5, SRS-6, SRS- 7, SRS-8, SRS-9

Test Procedure:

1. <VP4_Alternative1> Connect the simulator to the serial channel. [SRS-8]

2. <VP4_Alternative2> Connect the simulator to the Ethernet interface. [SRS-9]

3. <VP1_Option1> Send all of the messages to the simulator sequentially.

<VP3_Option1> Verify that the received messages are listed in the logger table in the correct order and with the correct name and time stamp information. [SRS-5]

<VP3_Option1> Verify that there exist time, name and source columns on the table.[SRS-6]

4. Click to each row on the logger table.

<VP2_Option1&VP3_Option1> Verify that the related message is displayed on the tree view. [SRS-3, SRS-7]

<VP1_Option1> Verify that all of the parameters of the received messages are in correct order and value. [SRS-1]

Domain Test Case Scenario -2

Addressed Requirements: SRS-2, SRS-4, SRS-5

Test Procedure:

1. <VP4_Alternative1> Connect the simulator to the serial channel.

2. <VP4_Alternative2> Connect the simulator to the Ethernet interface.

3. <VP1_Option2& VP2_Option1> Display and change all of the parameters of the sent messages. [SRS-4]

4. Send all of the messages from the communication interface sequentially.

<VP1_Option2> Verify that all of the parameters of the messages are sent in a

correct order and with the correct value. [SRS-2]
<VP3_Option1> Verify that the sent messages are listed in the logger table in the correct order and with the correct name and time stamp information. [SRS-5]

Example. The following example illustrates how requirements of a specific application are determined and how application test case scenarios are derived. The requirements of a sample project are derived from the domain requirements set. First, all mandatory requirements in the domain requirements set are taken to application requirements (There is not any mandatory requirement in this case). Later, all of the optional requirements are included in the application requirements. TBD expression is replaced with the Parmaters and Response names in SRS-1 and SRS-2 respectively. Then, alternative-2 variant is selected from the alternative requirements. Finally, product specific requirement SRS-10 is added to the requirement set.

Table4.3: Example Product Requirements Specification

Id	**Requirement**	**Test Level**
	Data Tree Representation Requirements	
SRS-1	<VP1_Option1> Simulator shall receive **Parameters** message over the communication interface.	System
SRS-2	<VP1_Option2> Simulator shall send **Response** message over the communication interface.	System
	GUI Requirements	
SRS-3	<VP2_Option1> STS-SPL shall include a tree view that displays all of the fields of the message classes. *Tree view shall be a GUI component that is developed with Java Swing toolkit.*	System

Continued on next page

Table 4.3 – *Continued from previous page*

Id	**Requirement**	**Test Level**
SRS-4	`<VP2_Option1:Requires_VP1_Option2>` Tree view shall let the user of the simulator to change the parameters of the displayed message to be sent on runtime.	System
SRS-5	`<VP3_Option1>` STS-SPL shall include a logger table that lists the sent and received messages. *Logger table shall be a GUI component that is developed with Java Swing toolkit.*	System
SRS-6	`<VP3_Option1>` The logger table shall include time stamp, name and source columns. *An object shall be hold in the memory for each of the rows of the table.*	System
SRS-7	`<VP3_Option1:Requires_VP2_Option1>` If the related object of a row is a message class, it shall be displayed on any tree view when the row is clicked by the user.	System
	Communication Requirements	
SRS-9	`<VP4_Alternative2>` The messages shall be sent and received over Ethernet interface.	System
SRS-10	`<ProductSpecific>` Simulator shall send **Response** message automatically as a response to the **Parameters** message.	System

After generating the requirements for the product, application test case scenarios are derived from domain test case scenarios as follows. The labels in the requirements are used to determine the test cases to be reused. Later, test cases are developed for the product specific requirements. Variability management labels may be removed after creating application requirements and test case scenarios.

Application Test Case Scenario-1

Addressed Requirements: SRS-1, SRS-3, SRS-5, SRS-6, SRS- 7, SRS-9

Test Procedure:

1. <VP4_Alternative2> Connect the simulator to the Ethernet interface. [SRS-9]

2. <VP1_Option1> Send all of the messages to the simulator sequentially.

<VP3_Option1> Verify that the received messages are listed in the logger table in the correct order and with the correct name and time stamp information. [SRS-5]

<VP3_Option1> Verify that there exist time, name and source columns on the table.[SRS-6]

<Product Specific> Verify that Response message is sent automatically as a response to the Parameters message. [SRS-10]

3. Click to each row on the logger table.

<VP2_Option1&VP3_Option1> Verify that the related message is displayed on the tree view. [SRS-3, SRS-7]

<VP1_Option1> Verify that all of the parameters of the received messages are in correct order and value. [SRS-1]

Application Test Case Scenario -2

Addressed Requirements: SRS-2, SRS-4, SRS-5

Test Procedure:

1. <VP4_Alternative2> Connect the simulator to the Ethernet interface.

2. <VP1_Option2&VP2_Option1> Display and change all of the parameters of the sent messages. [SRS-4]

3. <VP1_Option2> Send all of the messages from the communication interface sequentially.

<VP1_Option2> Verify that all of the parameters of the messages are sent in a correct order and with the correct value. [SRS-2]

<VP3_Option1> Verify that the sent messages are listed in the logger table in the correct order and with the correct name and time stamp information. [SRS-5]

4.3.2.1 Establishing Traceability between the SPL Assets

One of the most important activities of variability management is establishing traceability links between reusable assets. This leads to more efficient and controlled reusability. The complexity of the traceability increases with the number of requirements and test cases thus it must be managed explicitly. Traceability is established between the requirements (SRS) and test cases (DTCS) of the STS-SPL by generating a traceability matrix.

Table4.4: Traceability Matrix between Domain Requirements Specification and Domain Test Case Scenarios

Domain SRS	**Domain Test Case Scenario**
SRS-1	DTCS-1
SRS-2	DTCS-2
SRS-3	DTCS-1
SRS-4	DTCS-2
SRS-5	DTCS-1, DTCS-2
SRS-6	DTCS-1
SRS-7	DTCS-1
SRS-8	DTCS-1
SRS-9	DTCS-1

CHAPTER 5

MEASUREMENTS AND EVALUATION

Results of SPL adoption in test engineering department will be investigated in two contexts. First, measurements concerning real world practices will be taken. In this context, data will be acquired from real business performed in ASELSAN during and before this thesis work. It is difficult to compare different approaches in this context because it is not a controlled experiment. Thus, additional measurements were taken in another context which is more controlled.

The products developed with SSD and SPL approaches were compared with the measurements taken under each context. SPL products were developed by performing the whole life-cycle of application engineering where all reusable domain assets were utilized. Domain requirements, source code asset bases, reference architecture and domain test case scenarios were derived to be reused in applications. However, SSD products were developed from scratch by avoiding STS-SPL's infrastructure including reusable source code assets of COBALT. By this way SPL approach could be compared to development from scratch and the achievements of SPL adoption in a specific industrial setting could be inspected.

5.1 Industrial Experiences

In this context, six simulators those have been used in ASELSAN for the testing of real software were examined. Due to confidentiality reasons, names of the simulators will be suppressed but alias will be used instead. A, B and C simulators were developed with the single system development (SSD) approach and D,

E and F were developed with software product line SPL approach. Overview of the simulators used for the measurement in this context are given in Table 5.1.

Table5.1: Overview of the Simulators in Industrial Context

Simulator Name	**Development Approach**
A	SSD
B	SSD
C	SSD
D	SPL
E	SPL
F	SPL

When determining the simulators to be examined in the context of industrial experiences, the most similar ones were selected. In the assessment of the results under this topic, it is important to consider the following factors:

• All simulators have similar capabilities.

• All simulators were developed by the same developer but at different times when he has different level of experience. A, B and C were developed in an earlier period.

• Message structures are similar.

• A, B, C and D implement serial channel while E and F implement TCP communication interface.

• A, B and C were developed with C# and D, E and F were developed with JAVA programming language.

5.1.1 Reuse Rate

Reuse rate was measured by dividing reused SLOC by total SLOC in an individual simulator and given in Table 5.2.

A line of code is considered reused only if it is reused in both three SSD or SPL projects. In order to calculate reuse level for C# and JAVA source codes a reuse level calculator tool was developed. The tool parses the source code of a project and distinguishes the non-comment source line of codes. Then it splits the source code around matches of the characters '{', '}' and ';'. Then each split is considered a SLOC unless it starts with the keywords "using", "else", "break", "get", "set" and "try". Finally the tool counts the reused SLOCs by checking if that SLOC exists in all of the compared projects.

Table5.2: Reuse Rates of the Simulators in Industrial Context

Simulator Name	**Source Code Reuse Rate**
A	6%
B	14%
C	22%
D	85%
E	89%
F	94%

5.1.2 Implementation Effort

Implementation effort time was measured by taking the overall time required for the coding of each simulator and given in Table 5.3.

Table5.3: Implementation Effort of the Simulators in Industrial Context

Simulator Name	LOC	Implementation Effort (ManHour)	Effort/KLOC
A	4161	55	13,2
B	2176	30	13,8
C	951	17	17,9
D	9633	13	1,35
E	9213	8,5	0,92
F	8752	8	0,91

5.1.3 Maintenance Effort

Quantitative data is not available to measure this metric for the A, B, C, D, E and F simulators. However, the developer questionnaire given in Appendix C was filled by four application engineers those have developed at least three products with STS-SPL infrastructure. According to the questionnaire, developers agree that, developing a simulator from scratch is preferable when they have to maintain it in case of 50% of the requirements are modified and if the simulator is developed by another developer with single system development approach. They also strongly disagree that developing a simulator from scratch is preferable when they have to maintain it in case of 50% of the requirements are modified or 50% of the messages structures are modified and if the simulator is developed by another developer with software product line approach. A complete result of the questionnaire is given in Appendix C.

5.1.4 Testing Effort

Data is not available to measure this metric for the A, B, C, D, E and F simulators.

5.1.5 Defect Density Rate

Quantitative data is not available to measure this metric for the A, B, C, D, E and F simulators. However, according to the developer questionnaire given in Appendix C, developers strongly agree that there is a significant decrease in error counts with the SPL adoption.

5.1.6 CK Metrics

CK metrics suite provide quantitative data about the quality of the code being analyzed. However, in the present context, the software being compared do not implement the same requirements and message structures making strictly quantitative measurements insignificant. Thus, this metric suite was used in the context of controlled experiments.

5.1.7 Customer Satisfaction

A customer interview was made with 4 software test engineers who have at least 4 years of software testing experience. Each participant has used several STS-SPL products for their software testing efforts. Interview questions are given in Appendix D. Again, quantitative measurements like SUS questionnaire were avoided because of the differences among software being compared in this context.

All of the participants state that STS-SPL products have common look-and-feel and this leads to improved usability of the products and short learning periods. Users can intuitively use a new simulator after a few experiences with the STS-SPL products. A participant also emphasizes the importance of common look-and-feel in complex software testing setups where several simulators may be utilized simultaneously.

General customer satisfaction is quite high. Each participant indicates that STS-SPL products improve effectiveness and efficiency of software testing. The

main reason for this improvement is considered the improved quality and low defect density rates of products. A negative criticism about the STS-SPL is the late response time of new feature requests when they require the modification of SPL assets. Two participants suggest extending SPL capabilities by continuous asset scoping activities to improve time-to-market.

5.2 Controlled Experiments

In this context, four simulators were developed in a more controlled environment. X and Y simulators were developed with the single system development approach from scratch. Z and Q simulators were developed with SPL approach. Overview of the simulators used for the measurement in this context are given in Table 5.4.

Table5.4: Overview of the Simulators in Controlled Context

Simulator Name	**Development Approach**
X	SSD
Y	SSD
Z	SPL
Q	SPL

In the assessment of the results under this topic, it is important to consider the following factors:

• All simulators were developed by the same developer in a short period.

• X and Z simulators have the same requirements and message structure but developed with different approach.

• Y and Q simulators have the same requirements and message structures but developed with different approach.

• All simulators were developed with JAVA programming language.

5.2.1 Reuse Rate

Source code level reuse rate of the simulators in controlled experiments context were calculated using the reuse level calculator tool and given in Table 5.5. In this context, requirements and test cases reuse rates are also measured by dividing the number of reused items to the overall number of items. An item is regarded as reused when it is reused in all of the SSD or SPL products. In this context reuse improvement is not as much as in industrial context because of applying opportunistic reuse in SSD. However, average reuse level of SPL is still significantly higher than the SSD case. Reuse level of product Z is 100% because all of the requirements of Z is reused in product Q but product Q has some product specific requirements.

Table5.5: Reuse Rates of the Simulators in Controlled Context

Simulator Name	**Source Code Reuse Rate**	**Requirements Reuse Rate**	**Test Cases Reuse Rate**
X	55%	45%	50%
Y	79%	43%	50%
Z	96%	100%	71%
Q	97%	94%	71%

5.2.2 Implementation Effort

Implementation efforts for each simulator are given in Table 5.6.

Table5.6: Implementation Efforts of the Simulators in Controlled Context

Simulator Name	LOC	Implementation Effort (ManHour)	Effort/KLOC
X	2655	11,73	4,42
Y	1627	1,5	0,922
Z	9298	2,58	0,277
Q	9150	0,38	0,042

5.2.3 Maintenance Effort

In this experiment, the effort required to maintain the simulator was measured. Two application engineers made modifications to the simulators X, Y, Z and Q those were developed by a third developer. Modifications were done in terms of extensions by new message classes and new capabilities. Modifications done to X-Z and Y-Q pairs are just the same. The time required to make each modification was recorded and given in Table 5.7.

Table5.7: Maintenance Effort of the Simulators in Controlled Context

Simulator Name	Maintenance Effort of Developer 1 (min)	Maintenance Effort of Developer 2 (min)
X	33	47
Y	29	75
Z	9	30
Q	26	45

5.2.4 Testing Effort

Time required to create software requirement specification and related system level test case specification and performing the tests were measured and given in Table 5.8.

Table5.8: Testing Efforts of the Simulators in Controlled Context

Simulator Name	**SRS Creation Effort (min)**	**Test Case Creation Effort (min)**	**Testing Effort (min)**
X	46	56	60
Y	20	26	100
Z	14	14	100
Q	6	15	60

5.2.5 Defect Density Rate

The number of defects encountered during the system tests of each product was divided to software size to calculate defect density rate. Software size was measured by counting the thousand lines of code (KLOC). Results are given in Table 5.9.

Table5.9: Quality Level of the Simulators in Controlled Context

Simulator Name	**Defect Count**	**Defect Density (Defects/KLOC)**
X	12	4,5
Y	4	2,45
Z	2	0,215
Q	2	0,218

5.2.6 CK Metrics

CK metrics were measured using ckjm program [68]. All class files of the compiled software were given to the tool and the measurement results for each class file were summed up and divided to the file count to calculate average values

for each metric. When calculating results for the SPL products, class files of the infrastructure were given to the tool with the project specific class files and average values were calculated for the product. Measurement results are given in Table 5.10.

Table5.10: Chidamber and Kemerer's Metrics Results

Simulator Name	**WMC**	**DIT**	**NOC**	**CBO**	**RFC**	**LCOM**
X	4.45	1.76	0	2.45	16.82	19.47
Y	3.94	1.59	0	2.53	14.81	16.74
Z	6.15	1.21	0.24	3.21	15.59	20.71
Q	6.19	1.18	0.24	3.28	15.61	20.84

The description of how each metric is measured is as follows [69]:

WMC: The complexities of the methods are summed up to measure the weighted methods per class metric. Instead of calculating the cyclomatic complexity, the ckjm program assigns 1 to each method's complexity value. Hence, WMC value is equal to the number of methods of a class.

DIT: Depth of inheritance tree metric of a class is calculated by counting the levels of inheretance tree from top to the node being measured.

NOC: Number is childeren is calculated by counting the number of immediate descendants of the class.

CBO: Coupling between object classes is measured by counting the number of classes coupled to a given class through method calls, field accesses, inheritance, arguments, return types, and exceptions.

RFC: Ckjm prgram calculates the response for a class metric by counting the number of method calls within the class's method bodies instead of calculat-

ing the transitive closure of the method's call graph. (Transitive closure of the method's call graph is the set of methods those can be reached through the method calls of this class or by repeating this for each called method.) This simplication is suggested in the original (1994) Chidamber and Kemerer descriptions [61].

LCOM: The lack of cohesion in methods is calculated by subtracting the number of method pairs that access at least one common field of the class, from the number of method pairs that don't share a field.

5.2.7 Customer Satisfaction

The end user questionnaire given in Appendix B was given to 3 people to answer the questions about the usability of the simulators X, Y, Z and Q. In this context, each simulator was developed by the same developer and each participant evaluated all of the simulators. Results are given in Table 5.11. Average SUS scores indicate that SPL products' usability is relatively higher than the SSD products.

Table5.11: Usability of the Simulators in Industrial Context

Development Approach	**Sample Count**	**Average SUS Score**
SSD (X and Y)	6	51
SPL (Z and Q)	6	91

CHAPTER 6

DISCUSSION AND CONCLUSION

In this study, action research was conducted within ASELSAN, a major defense industry company in Turkey. The aim of the research was inspecting the consequences of SPL adoption in software testing department for the software testing simulators product family. First, the problems with the traditional single system development approach of the department are reported as effort overload, demand for higher product quality, heterogeneous look-and-feel and low reusability. Then, SPL adoption is suggested to overcome these challenges of software development and software testing simulators software product line is constructed by extending the legacy system COBALT. Short and long term strategy of the business and SPL product portfolio are defined. STS-SPL requirements specification, reference architecture and the variability model are designated. An initiation of the SPL is enabled by these definitions and later additional improvements of SPL are performed. Applying asset scoping enabled SPL to extend with two new requirements. Later, a test cases asset base was created by initially covering all STS-SPL requirements. Applied labeling system facilitated reusing these domain test case scenarios in application development. These efforts and related measurements were carried out in accordance with action research methodology.

Consequences of SPL adoption are inspected from the perspective of previously reported problems with the traditional single system development approach. Thus, there are four aspects of the assessments: Effort, quality, look-and-feel and reusability.

Measurements show that implementation, maintenance and testing efforts were significantly reduced with SPL adoption. Effort required performing the tests didn't improve because all products were tested from scratch with SSD testing approach; however SRS and test case creation efforts reduced significantly.

Four quality indicators were utilized in this study: Defect density rate, code-based CK metrics, maintenance effort, customer satisfaction. First, decreased defect density rates indicate highly reliable and of good quality products. Second, code based metrics reveal the complexity problem of SPL infrastructure. CK measurements show that SPL products have relatively high WMC, NOC, CBO and LCOM values, low DIT values and roughly equal RFC values. Five metrics are related to complexity and three of them (WMC, CBO and LCOM values) indicate high complexity which leads to difficult maintenance and testing. However, most of this complexity comes from the shared components and it generally complicates the rework and testing efforts performed by the domain engineers. Contrarily to other metrics, DIT results indicate low design complexity and roughly equal RFC mean equal complexity. Combining the results, we can conclude that attention should be given to the high complexity components and problematic components should be revised. Third, since maintenance efforts reduce remarkably, it indicates high maintainability of SPL products. Finally, applied questionnaires and interviews exhibit high customer satisfaction that is an indicator of improved quality.

Applied SUS questionnaire and customer interview results show that STS-SPL products' usability is better and STS-SPL products have common look-and-feel.

Source code, requirement and test case reuse rates increased with the SPL adoption. Actually it is a natural consequence of any SPL initiative because the development for reuse is one of the fundamental actions of SPL development. Actually, improved reusability enables the previously described benefits of the SPL.

In conclusion, SPL adoption led to significant effort reduction, improved product quality, homogenous look-and-feel and improved reusability in the software development efforts of software testing department of ASELSAN.

The main challenge of this study was measuring the effects of proposed changes in software development in actual industrial settings due to time and effort constraints. The process of developing a simulator product for real-world software testing purposes is a large task and creating lots of them with control sample pairs in order to make quantitative assessments was not possible in the period of this study. Hence, two identical product pairs were developed with SSD and SPL approaches under the controlled experiments context; but other measurements taken within the industrial experiences context were done with control samples those are analogous in terms of features and message structures but not identical. Furthermore, using these simulators in software tests and measuring effectiveness and efficiency of the simulators was not possible because of the projects' scheduling.

In the short term, a future study can aim to collect quantitative data for the further assessment of SPL improvements. In addition, effectiveness and efficiency measurements shall be taken by investigating the consequences of software testing projects those utilize STS-SPL products.

The improvements of SPL adoption can be expended by creating SPL management tools. Tool support is an efficient mechanism for configuring domain components to be reused in applications [2]. Thus, SPL management tools may be created and used for managing reusable assets. Furthermore, STS-SPL can be extended to incorporate automation abilities for the products. Because automation is an efficient way of performing software tests and this can be enabled by using simulators with automation support.

Finally, SPL testing is an important concern of the SPL adoption process. The lack of attention given to SPL testing leads it to become the bottleneck of the entire process. Thus improvements in testing shall be handled as a long term fu-

ture work especially regarding automated test case generation strategies. There are significant amount of studies regarding automatic test case derivation strategies in literature [70, 71, 72, 73, 74]. These strategies mostly adopt model-based testing approach. This approach includes a test model (state charts, activity diagrams) which is derived from the requirements. Then, test cases are derived from the test model. Most important challenge for model-based testing in software product lines is the variability management [75]. To deal with the variability, testing process of an organization may need to be revised comprehensively. In industry where SPL concepts are not strictly followed, it is hard to apply a model-based test case derivation technique [76].

REFERENCES

[1] D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2(1):1–9, January 1976.

[2] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[3] Lawrence G. Jones and Linda M. Northrop. Clearing the way for software product line success. *IEEE Softw.*, 27(3):22–28, May 2010.

[4] Jan Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, pages 257–271, London, UK, UK, 2002. Springer-Verlag.

[5] Bonnie Kaplan and Joseph Maxwell. Qualitative Research Methods for Evaluating Computer Information Systems. In James Anderson and Carolyn Aydin, editors, *Evaluating the Organizational Impact of Healthcare Information Systems*, Health Informatics, chapter 2, pages 30–55. Springer New York, New York, 2005.

[6] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14:131–164, April 2009.

[7] Richard L. Baskerville. Investigating information systems with action research. *Communications of the Association for Information Systems*, 2(19), October 1999.

[8] P. Reason and H. Bradbury. *Handbook of Action Research: Participative Inquiry and Practice*. SAGE Publications, 2001.

[9] Cummins, inc: Diesel engine software product line. `http://splc.net/fame/cummins.html`. Accessed: 2013-07-21.

[10] Hans Peter Jepsen and Danilo Beuche. Running a software product line: standing still is going backwards. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 101–110, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[11] Paul Clements and John K. Bergey. The u.s. army's common avionics architecture system (caas) product line: A case study. Technical report, 2005.

[12] Daniel Pech, Jens Knodel, Ralf Carbon, Clemens Schitter, and Dirk Hein. Variability management in small development organizations: experiences and lessons learned from a case study. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 285–294, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[13] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[14] Sholom Cohen, Ed Dunn, and Albert Soule. Successful product line development and sustainment: A dod case study. Technical report, Carnegie Mellon University, 2002.

[15] Dong Li and Carl K. Chang. Initiating and institutionalizing software product line engineering: From bottom-up approach to top-down practice. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 01*, COMPSAC '09, pages 53–60, Washington, DC, USA, 2009. IEEE Computer Society.

[16] Mark Staples and Derrick Hill. Experiences adopting software product line development without a product line architecture. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 176–183, Washington, DC, USA, 2004. IEEE Computer Society.

[17] Ari Jaaksi. Developing mobile browsers in a product line. *IEEE Softw.*, 19(4):73–80, July 2002.

[18] *Reducing avionics software cost through component based product line development*, volume 2, 1998.

[19] D. Faust and C. Verhoef. Software product line migration and deployment. *Software Practice and Experience, John Wiley and Sons, Ltd*, 33:933–955, 2003.

[20] Liang Liang, Zhiqiang Hu, and Xiangyun Wang. An open architecture for medical image workstation. pages 470–479, 2005.

[21] Paul Clements, Sholom Cohen, Patrick Donohoe, and Linda Northrop. Control channel toolkit: a software product line case study. Technical report, 2001.

[22] L. Brownsword and P.C. Clements. *A Case Study in Successful Product Line Development.* Technical report. Carnegie Mellon University, Software Engineering Institute, 1996.

[23] Mark Ardis, Nigel Daley, Daniel Hoffman, Harvey Siy, and David Weiss. Software product lines: a case study. *Software: Practice and Experience*, 30(7):825–847, 2000.

[24] Siemens healthcare: Software product line for 3d routine and advanced reading (syngo.via). `http://splc.net/fame/siemens.html`. Accessed: 2013-07-21.

[25] Yoshihiro Matsumoto. A guide for management and financial controls of product lines. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 163–170, Washington, DC, USA, 2007. IEEE Computer Society.

[26] J. Bergey. *Software Product Lines: Experiences from the Sixth DoD Software Product Line Workshop*. Technical note. Carnegie Mellon University, Software Engineering Institute, 2004.

[27] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying software product-line architecture. *Computer*, 30(8):49–55, 1997.

[28] Charles W. Krueger. Time-to-market benefits of software product lines. `http://www.softwareproductlines.com/benefits/time.html`. Accessed: 2013-07-24.

[29] Paul Jensen. Experiences with product line development of multi-discipline analysis software at overwatch textron systems. *Software Product Line Conference, International*, 0:35–43, 2007.

[30] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[31] Holt Mebane and Joni T. Ohta. Dynamic complexity and the owen firmware product line program. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 212–222, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Product line hall of fame - general motors powertrain (gmpt). `http://splc.net/fame/gm.html`. Accessed: 2013-07-21.

[33] Lsi logic: Raid controller firmware product line. `http://splc.net/fame/lsilogic.html`. Accessed: 2013-07-21.

[34] David Sellier, Gorka Benguria, and Gorka Urchegui. Introducing software product line engineering for metal processing lines in a small to medium enterprise. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 54–62, Washington, DC, USA, 2007. IEEE Computer Society.

[35] Gerard Quilty and Mel Ó. Cinneide. Experiences with software product line development in risk management software. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 251–260, Washington, DC, USA, 2011. IEEE Computer Society.

[36] Philips: Telecommunication switching system, philips. `http://splc.net/fame/philips-switching-system.html`. Accessed: 2013-07-21.

[37] Salion's product line of revenue acquisition management systems. `http://splc.net/fame/salion.html`. Accessed: 2013-07-21.

[38] Walter J. Slegers. Building automotive product lines around managed interfaces. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 257–264, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[39] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Trans. Softw. Eng. Methodol.*, 11(2):191–214, apr 2002.

[40] Ronny Kolb, Isabel John, Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. Experiences with product line development of embedded systems at testo ag. In *Proceedings of the 10th International on Software Product Line Conference*, SPLC '06, pages 172–181, Washington, DC, USA, 2006. IEEE Computer Society.

[41] Dharmalingam Ganesan, Jens Knodel, Ronny Kolb, Uwe Haury, and Gerald Meier. Comparing costs and benefits of different test strategies for a software product line: A study from testo ag. *Software Product Line Conference, International*, 0:74–83, 2007.

[42] Parastoo Mohagheghi and Reidar Conradi. An empirical investigation of software reuse benefits in a large telecom product. *ACM Trans. Softw. Eng. Methodol.*, 17(3):13:1–13:31, jun 2008.

[43] P. Sanchez, D. Garcia-Saiz, and M. Zorrilla. Software product line engineering for e-learning applications: A case study. In *Computers in Education (SIIE), 2012 International Symposium on*, pages 1–6, 2012.

[44] Product line hall of fame - homeaway: Online vacation rental marketplace. `http://splc.net/fame/homeaway.html`. Accessed: 2013-07-21.

[45] Frank Dordowsky and Walter Hipp. Adopting software product line principles to manage software variants in a complex avionics system. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 265–274, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[46] Michael Vierhauser, Gerald Holl, Rick Rabiser, Paul Grunbacher, Martin Lehofer, and Uwe Sturmer. A deployment infrastructure for product line models and tools. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 287–294, Washington, DC, USA, 2011. IEEE Computer Society.

[47] Charles W. Krueger, Dale Churchett, and Ross Buhrdorf. Homeaway's transition to software product line practice: Engineering and business results in 60 days. In *Proceedings of the 2008 12th International Software Product Line Conference*, SPLC '08, pages 297–306, Washington, DC, USA, 2008. IEEE Computer Society.

[48] Ericsson axe family of telecommunications switches. `http://splc.net/fame/ericsson.html`. Accessed: 2013-07-21.

[49] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. Refactoring a legacy component for reuse in a software product line: a case study: Practice articles. *J. Softw. Maint. Evol.*, 18(2):109–132, March 2006.

[50] Hans Peter Jepsen, Jan Gaardsted Dall, and Danilo Beuche. Minimally invasive migration to software product lines. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 203–211, Washington, DC, USA, 2007. IEEE Computer Society.

[51] S. Wagner. *Software Product Quality Control*. Springer-Verlag New York Incorporated, 2013.

[52] ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, 2011.

[53] Francisca Losavio, Ledis Chirinos, Alfredo Matteo, Nicole Lévy, and Amar Ramdane-Cherif. Iso quality standards for measuring architectures. *Journal of Systems and Software*, 72(2):209–223, 2004.

[54] A. K. M. M. Islam, T. Gorschek, M. Unterkalmsteiner, R. Feldt, R. B. Permadi, and Chow Kian Cheng. Evaluation and measurement of software process improvement - a systematic literature review. *IEEE Transactions on Software Engineering*, 38(2):398–424, 2012.

[55] John Brooke. Sus: A quick and dirty usability scale, 1996.

[56] B. Deniz. Investigation of the effects of reuse on software quality in an industrial setting. Master's thesis, Graduate School of Natural and Applied Sciences of Middle East Technical University, January 2013.

[57] Jane Huffman Hayes, Sandip C. Patel, and Liming Zhao. A metrics-based software maintenance effort model. In *CSMR*, pages 254–260. IEEE Computer Society, 2004.

[58] T. Turk. The effect of software design patterns on object-oriented software quality and maintainability. Master's thesis, Graduate School of Natural and Applied Sciences of Middle East Technical University, September 2009.

[59] C. Unlu. The effects of test driven development on software productivity and software quality. Master's thesis, Graduate School of Natural and Applied Sciences of Middle East Technical University, September 2008.

[60] Arthur H. Watson and Thomas J. Mccabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical Report NIST Special Publication 500-235, National Institute of Standards and Technology (NIST), September 1996.

[61] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[62] Seyyed Mohsen Jamali. Object oriented metrics (a survey approach), 2006.

[63] Aselsan, inc: Field of activities. `http://www.aselsan.com.tr/content.aspx?mid=375&oid=375`. Accessed: 2013-07-25.

[64] MIL-STD-498. Military standard: Software development and documentation. Technical Report AMSC NO. N7069, US Department of Defense, 1994.

[65] Ljubomir Lazic and Nikos Mastorakis. Cost effective software test metrics. *W. Trans. on Comp.*, 7(6):599–619, jun 2008.

[66] Edward Curry and Paul Grace. Flexible Self-Management Using the Model-View-Controller Pattern. *IEEE Software*, 25(3):84–90, may 2008.

[67] Klaus Schmid and Frank van der Linden. Improving product line development with the families evaluation framework (fef). In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*, pages 15–16. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007.

[68] Diomidis Spinellis. Tool writing: A forgotten art? *IEEE Software*, 22(4):9–11, July/August 2005.

[69] Diomidis Spinellis. Metric descriptions. `http://www.spinellis.gr/sw/ckjm/doc/metric.html`. Accessed: 2013-11-30.

[70] M. Mohamed Ali and R. Moawad. An approach for requirements based software product line testing. In *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pages 1–10, 2010.

[71] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental test generation for software product lines. *Software Engineering, IEEE Transactions on*, 36(3):309–322, 2010.

[72] B.P. Lamancha, O. Diaz, M. Azanza, and M. Polo. Software product line testing: A feature oriented approach. In *Industrial Technology (ICIT), 2012 IEEE International Conference on*, pages 298–305, 2012.

[73] Yankui Feng, Xiaodong Liu, and Jon Kerridge. A product line based aspect-oriented generative unit testing approach to building quality components. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pages 403–408, Washington, DC, USA, 2007. IEEE Computer Society.

[74] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 459–468, Washington, DC, USA, 2010. IEEE Computer Society.

[75] Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis. Model-based system testing of software product families. In *Proceedings of the 17th international conference on Advanced Information Systems Engineering*, CAiSE'05, pages 519–534, Berlin, Heidelberg, 2005. Springer-Verlag.

[76] Emelie Engström and Per Runeson. Decision support for test management and scope selection in a software product line context. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 262–265, Washington, DC, USA, 2011. IEEE Computer Society.

APPENDIX A

CASE STUDIES OVERVIEW

An overview of the case studies and reported improvements in software development achieved with SPL adoption are given in Table A.1.

TableA.1: Case Studies Overview

Nu	**Company Name**	**Reported Improvements**
1	AKVAsmart ASA [2]	Code size, Look-and-feel, Maintainability, Integrability, Easier reuse
2	Argon Engineering [26]	Scheduling, Development cost, Upgade cost, Cost
3	Asea Brown Boveri (ABB) [13]	Development time , Quality, Cost
4	Boeing [18]	Cost
5	CelsiusTech [22, 30]	Scheduling, Time-to-market, Productivity, Cost, Reliability, Competitiveness, Improved reuse, Customer satisfaction, Integrability
6	Cummins [9]	Time-to-market, Competitiveness, Productivity, Quality, Customer satisfaction
7	Danfoss Drives [50]	Production simplicity, Development efficiency
8	Deutsche Bank [19]	Cost
9	Dialect Solutions [16]	Quality, Development efficiency, Productivity
10	DNV Software [2]	Time to market, Quality, Life cycle costs

Continued on next page

Table A.1 – *Continued from previous page*

Nu	Company Name	Reported Improvements
11	E-COM Technology Ltd. [20]	Cost, Time to market, Productivity
12	Ericsson-axe [48]	Adaptability
13	Ericsson-telecom network [42]	Defect count
14	Eurocopter [45]	Variability
15	ElWM (E-learning Web Miner) [43]	Customizability, Quality, Testing cost
16	Fiscan [15]	Cost, Time to market, Improved quality
17	General Motors Powertrain [32]	Time-to-market
18	Hewlett Packard [13, 31]	Labor cost, Defect count, Complexity, Productivity, Time-to-market
19	HomeAway [47, 44]	Performance, Complexity, Testability, Quality, Time-to-market, Maintenance cost
20	LG Industrial Systems [13]	Complexity, Function size, Upgrade cost
21	LSI Logic - Engenio Storage Group [33]	Variability, Time-to-market
22	Lucent Technologies [23, 13]	Productivity, Reliability, Performance
23	Market Maker Software AG [2]	Time-to-market, Maintenance cost, Cost of quality, Reliability
24	Mondragón Sistemas de Información (MSI) [34]	Development time, Product heterogeneity
25	Nokia-Mobile Browsers [17]	Development efficiency, Cost, Quality, Developer satisfaction
26	Nokia Networks [2]	Manageability, Improved reuse

Continued on next page

Table A.1 – *Continued from previous page*

Nu	Company Name	Reported Improvements
27	Nortel [27]	Cycle time
28	ORisk Consulting [35]	Time for changes Quality, Customizability, Competitiveness
29	Overwatch Textron Systems [29]	Time to market,
30	Philips- PKI telecommunications switching system [36]	Time to market, Improved reuse
31	Philips-High-end Televisions [2]	Reliability, Diversity, Variability
32	Philips Medical Systems [2]	Effort, Time-to-market, Defect count, Look-and-feel, Manageability
33	Raytheon; U. S. National Reconnaissance Office [21, 13]	Cost, Time to market, Quality
34	Ricoh citeKolb:2006:RLC:1133105.1133108	Maintainability, Code size, Complexity
35	Robert Bosch Corp. [2]	Calibration effort, Cost,
36	Rockwell Collins - Common Army Avionics System (CAAS) [11]	Maintenance cost, Documentation cost , Testing cost, Development time, Development cost, Competitiveness
37	Salion, Inc. [37]	Time to market
38	Siemens - Metals Technologies. [46]	Customer satisfaction

Continued on next page

Table A.1 – *Continued from previous page*

Nu	Company Name	Reported Improvements
39	Siemens - Software for viewing and quantifying radiological images [2]	Cycle time, Cost of quality,
40	Siemens Healthcare (syngo.via) [24]	Look and feel, Quality, Productivity
41	Telvent [2]	Competitiveness, Manageability
42	Testo [40]	Competitiveness, Diversity, Developer satisfaction, Scheduling, Quality
43	TomTom Automotive [38]	Time to market
44	Toshiba [25]	Productivity
45	U. S. Naval Undersea Warfare Center [13, 14]	Cost, Development time, Labor cost, Customer Satisfaction, Competitiveness
46	U. S. Army [39]	Development time, Manageability, Complexity
47	U.S. Army Technical Applications Program Office (TAPO) [11]	Development, maintenance, Integration and Documentation costs, Development time, Improved reuse
48	Wikon GmbH [12]	Quality, Effort, Maintenance cost

APPENDIX B

SUS QUESTIONNAIRE

The following questions are asked to the participant in order to evaluate the usability of the simulator which is not developed but have been used by the participant [55].

1. I think that I would like to use this system frequently
2. I found the system unnecessarily complex
3. I thought the system was easy to use
4. I think that I would need the support of a technical person to be able to use this system
5. I found the various functions in this system were well integrated
6. I thought there was too much inconsistency in this system
7. I would imagine that most people would learn to use this system very quickly
8. I found the system very cumbersome to use
9. I felt very confident using the system
10. I needed to learn a lot of things before I could get going with this system

B.1 Scoring SUS

The participant gives a score to each question between 1 (Strongly disagree) and 5 (Strongly agree). For items 1, 3 ,5, 7 and 9 the scale position minus 1 gives the score contribution and for items 2, 4, 6, 8 and 10; 5 minus the scale position gives the contribution. Then, sum of the scores is multiplied by 2,5 and overall

SUS value is obtained. SUS scores range from 0 to 100 [55]. Relatively bigger score states better usability and end user satisfaction.

APPENDIX C

STS SPL DEVELOPER QUESTIONNAIRE

The following questions are asked to the 4 participants who have used STS-SPL infrastructure in the development of at least three products.

(Please give a score between 1-5 to all of the questions where 1 stands for strongly disagree and 5 stands for strongly agree.)

1. I think that I would like to develop a simulator from scratch when it is developed by another developer with single system development approach and I have to maintain it in the case of...
a. 50% of the message structures are modified
b. 50% of the requirements are modified

2. I think that I would like to develop a simulator from scratch when it is developed by another developer using the STS-SPL infrastructure and I have to maintain it in the case of...
a. 50% of the message structures are modified
b. 50% of the requirements are modified

3. I think that there is a significant quality improvement (decrease in error counts) with the SPL adoption.

C.1 STS-SPL Developer Questionnaire Results

TableC.1: STS-SPL Developer Questionnaire Results

Question Number	Developer I Scores	Developer II Scores	Developer III Scores	Developer IV Scores	Average Scores
1.a.	4	2	2	3	2,75
1.b.	5	4	4	3	4
2.a.	1	2	1	1	1,25
2.b.	1	2	1	2	1,5
3.	4	5	5	5	4,75

APPENDIX D

CUSTOMER INTERVIEW

The following questions are asked to the participant in order to evaluate the customer satisfaction.

1. Do you think that SPL products have common look-and-feel? Why common look-and-feel is important for you?

2. What is your general opinion about STS-SPL? How does it effect software tests' effectiveness and efficiency?

3. Do you have further suggestions to improve STS-SPL?