INVESTIGATION OF THE EFFECTS OF REUSE ON SOFTWARE QUALITY IN AN INDUSTRIAL SETTING

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

BY

BERKHAN DENİZ

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

JANUARY 2013

Approval of the thesis:

INVESTIGATION OF THE EFFECTS OF REUSE ON SOFTWARE QUALITY IN AN INDUSTRIAL SETTING

submitted by **BERKHAN DENİZ** in partial fulfillment of the requirement 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. İsmet Erkmen _____
Head of Department, **Electrical and Electronics Engineering**

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

Examining Committee Members:

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

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

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

Assoc. Prof. Dr. Ali H. Doğru _____
Computer Engineering Dept., METU

Gökhan Öztaş, M.Sc. _____
Software Engineering Dept., Aselsan

Date: **07.01.2013**

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 : Berkhan Deniz

Signature :

ABSTRACT

INVESTIGATION OF THE EFFECTS OF REUSE ON SOFTWARE QUALITY IN AN INDUSTRIAL SETTING

Deniz, Berkhan
M. Sc., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Semih Bilgen

January 2013, 54 pages

Software reuse is a powerful tool in order to reduce development and maintenance time and cost. Any software life cycle product can be reused, not only fragments of source code. A high degree of reuse correlates with a low defect density. In the literature, many theoretical and empirical researches have examined the relationship of software reuse and quality. In this thesis, the effects of reuse on software quality are investigated in an industrial setting. Throughout this study, we worked with Turkey's leading defense industry company: Aselsan's software engineering department. We aimed to explore their real-life software projects and interpret reuse and quality relations for their projects. With this intention, we defined four different hypotheses to determine reuse and quality relations; and in order to confirm these hypotheses; we designed three separate case studies. In these case studies, we collected and calculated reuse and quality metrics i.e. Object-oriented quality metrics, reuse rates and performance measures of individual modules, fault-proneness of software components, and productivity rates of different products. Finally, by analyzing these measurements, we developed suggestions to further benefit from reuse in Aselsan through systematic improvements to the reuse infrastructure and process. Similar case studies have been reported in the literature, however, in Turkey, there are not many case studies using real-life project data, particularly in the defense industry.

Keywords: Software reuse, Quality metrics, Embedded software, Fault-proneness, Empirical study.

ÖZ

YENİDEN KULLANIMIN YAZILIM KALİTESİNE ETKİLERİNİN ENDÜSTRİYEL BİR ÇERÇEVEDE İNCELENMESİ

Deniz, Berkhan
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Semih Bilgen

Ocak 2013, 54 Sayfa

Yazılım yeniden kullanımı, geliştirme ve bakım zamanını ve maliyetini azaltmak için güçlü bir araçtır. Yalnızca kaynak kod parçaları değil; herhangi bir yazılım yaşam döngüsü ürünü, yeniden kullanılabilir. Yüksek derecede yeniden kullanım, düşük yazılım hata oranı ile ilişkilidir. Literatürde birçok teorik ve deneysel araştırma yazılım yeniden kullanımı ile yazılım kalitesi ilişkisini incelemiştir. Bu tezde, yazılım kalitesine yazılım yeniden kullanımının etkileri endüstriyel bir ortamda incelenmiştir. Bu çalışma boyunca, Türkiye'nin önde gelen savunma sanayi firması Aselsan'ın Yazılım Mühendisliği Bölümü ile çalıştık. Bu çalışmada, gerçek yazılım projelerinin incelenmesi ve bu projelerde yazılım yeniden kullanımı ve yazılım kalitesi ilişkilerinin yorumlanması amaçlanmıştır. Bu niyetle, yeniden kullanım ve kalite ilişkilerini belirlemek amacıyla, dört farklı hipotez tanımlanmıştır ve bu hipotezleri doğrulamak amacıyla, üç ayrı vaka çalışması tasarlanmıştır. Bu vaka çalışmalarında, nesne odaklı kalite ölçümleri, farklı modüllerin yeniden kullanım ve performans ölçümleri, yazılım bileşenlerinin hata yatkınlığı ve farklı ürünlerin üretkenlik oranları gibi metrikler toplanmış ve ölçülmüştür. Son olarak, bu ölçümler analiz edilerek, yeniden kullanım altyapı ve süreçlerinde sistematik iyileştirmeler yapılması yoluyla, Aselsan'ın yazılım yeniden kullanımından daha fazla yararlanması için öneriler geliştirilmiştir. Literatürde benzer vaka çalışmaları rapor edilmiştir; ancak Türkiye'de, gerçek proje verileri kullanılarak yapılmış, özellikle savunma sanayisi alanında, çok fazla vaka çalışması bulunmamaktadır.

Anahtar Kelimeler: Yazılım yeniden kullanımı, Kalite metrikleri, Gömülü yazılım, Hata yatkınlığı, Deneysel çalışma.

To my grandfather Sabri Deniz

ACKNOWLEDGMENTS

This thesis would not have been possible without the guidance and the help of several individuals who, by any means, contributed and provided their valuable assistance in the preparation and completion of this study.

First and foremost, I would like to present my truthful gratitude to Prof. Dr. Semih Bilgen, whose patience, sincerity, supervision and encouragement I will never forget. Without his valuable guidance, this thesis would not have been completed.

I would like to thank TÜBİTAK for its support throughout this thesis.

I appreciate Emra Aşkaroğlu, for the fruitful discussions throughout the development of this thesis.

I am grateful to my team leader Soner Çınar, for his tolerance during this study.

I owe Zehra Burtay Cirit and Hazel Sürücü a debt of gratitude; since, during this exhausting process, they were with me while studying for long hours in Bilkent.

I feel thankful to Murat Güreken for his precious support throughout this study.

My special thanks go to Cansu Bender, Burcu Karasoy, and Deniz Karatay for reviewing this thesis.

I would like to thank all my friends and colleagues in Aselsan, for their understanding and continuous support during this thesis. I would like to express my gratitude to the Software Engineering Department since they assisted me obtaining the essential metrics in order to bring this study into a successful conclusion.

Last but not least, I would like to thank my mother and my father for their love, trust, understanding and every kind of support not only throughout this study but also throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

AVT	Automatic Video Tracking
BSP	Board Support Package
CBO	Coupling Between Objects
CBS	Component-Based Software
CCCC	C and C++ Code Counter
CK	Chidamber and Kemerer
CPU	Central Processing Unit
DIT	Depth of Inheritance Tree
FORM	Feature Oriented Reuse Method
HSSS	Air Defense Weapon Systems (Tr. Hava Savunma Silah Sistemleri)
IEC	International Electrotechnical Commission
IP	Internet Protocol
ISO	International Organization for Standardization
LCOM	Lack of Cohesion in Methods
McCabeCC	McCabe Cyclomatic Complexity
NBD	Nested Block Depth
NOC	Number of Children
NoCycles	Number of Cycles
OO	Object-Oriented
OOFP	Object-Oriented Function Point
RFC	Response for a Class
RTOS	Real-Time Operating System
SQuaRE	Software product Quality Requirements and Evaluation
SLOC	Source Line of Code
SPL	Software Product Line
SRS	Software Requirements Specification
SSRM	Fire Control Systems Reference Architecture (Tr. Silah Sistemleri Referans Mimarisi)
SST	Defense System Technologies (Tr. Savunma Sistem Teknolojileri)
TADES	Tactical Fire Support Systems (Tr. Taktik Ateş DEstek Sistemleri)
TCP	Transmission Control Protocol
UML	Unified Modeling Language
VT	Video Tracker
WMC	Weighted Methods per Class
YMM	Software Engineering Department (Tr. Yazılım Mühendisliği Müdürlüğü)

CHAPTER 1

INTRODUCTION

Software reuse is a powerful tool to reduce development and maintenance time and cost. Any software life cycle product can be reused, not only fragments of source code. This means that developers can reuse requirements documents, system specifications, design structures, and any other development artifact [1]. As software assets are reused, the accumulated defect fixes result in higher quality [2]. Therefore, a high degree of reuse correlates with a low defect density.

Since software reuse and its effects on software quality are key concepts of software development, measurement and comparison of software reuse and software quality are also essential in the software life cycle.

Using reused and non-reused source line of codes is the most used software reuse measurement type; however additionally the number of function-points, number of uses-cases or set of requirements can be used to measure software reuse [3, 4]. Number of reused components in component-based software development is also another metric to evaluate software reuse.

Similar to software reuse measurement, there is no single standard method for software quality measurement. ISO/IEC 9126 standard [5] is the most commonly used guide for software quality measurement. In this standard, the quality framework for external and internal quality is defined. Software quality attributes are categorized into six characteristics (functionality, reliability, usability, efficiency, maintainability and portability), which are further subdivided into sub-characteristics which can be measured by internal or external metrics. Fault Detection metric of the Reliability characteristic of the model is the most reported quality metrics in the literature. This metric is about how many faults were detected in the software product [6]. The number of defects per non-comment source line value is suggested for measuring these metrics. In addition to defects density, rework effort -using Stability sub-characteristic's Change Impact metrics of the model- is also used for measuring quality [7].

Additionally, object-oriented (OO) software metrics, in order to determine the quality of object oriented software, are used for measuring quality. These metrics can be used in many ways: making system level predictions, early identification of high-risk software components, and the establishment of preventative design and programming guidelines [8]. Chidamber and Kemerer's metrics suite for object-oriented design [9] is one of the most rigorous studies in OO metrics investigation.

In the literature, many studies and experiments have been reported on software reuse and software quality using historical data on defects in order to assess impacts of reuse on quality in object-oriented systems [6, 10, 11]. Most of these studies have shown the strong correlation between software reuse and software quality even though these works have differences in measurement types, experimental approaches, and etc.

In this thesis, we investigated real-life software projects of Turkey's leading defense industry company – Aselsan, in order to observe reuse and quality relations of these projects. With this intention, we designed three separate case studies and collected and calculated metrics i.e. Object-oriented quality metrics, reuse rates and performance of individual modules, fault-proneness of components, and productivity rates of the products. Then by analyzing these metrics, we reached

some useful conclusions: based on these case studies, we also develop suggestions to further benefit from reuse through systematic improvements to the reuse infrastructure and process. In literature, similar case studies have been reported; however, in Turkey, there are not many case studies using real-life project data, particularly in the defense industry.

The remaining chapters of the study are organized as follows:

In Chapter 2, the background information about reuse types, the methods of measuring reuse and quality and ISO/IEC 9126 quality model are presented. Furthermore, object-oriented software quality metrics are introduced. Also, previous works on the effects of software reuse on software quality are summarized.

In Chapter 3, the experimental work is explained. The research hypotheses are presented. The designed case studies and the collected reuse and quality metrics are introduced. Then, the collected data is analyzed, and the research hypotheses are verified. Finally, suggestions to improve the reuse infrastructure in Aselsan are formulated based on the reported case studies.

Chapter 4 concludes the thesis. The work done and the obtained results are summarized. The achievements and difficulties of this study are reviewed; further suggestions for future studies are offered.

CHAPTER 2

LITERATURE REVIEW

In this chapter, the relationship of software reuse and software quality is summarized. First, reuse types and reuse measurement methods are described. Then, quality measurement methods are covered: ISO/IEC 9126 quality standard is described, metrics derived from this standard are listed, and the critical literature on the standard is summarized. Additionally, object-oriented software quality metrics are explained: Theoretical and empirical analyses of the selected metrics are provided. Finally, effects of software reuse on software quality are summarized for both general purpose software and embedded software; furthermore relationship of performance requirements of embedded software systems and software reuse is investigated.

2.1 REUSE TYPES

Software reuse refers to the usage of the same software artifact in multiple instances [2]. Software reuse can be applied not only to source code but also to any software life cycle product, process and information. This means developers can utilize reuse via most software related entities such as requirements, system specifications, design reports, and processes such as domain engineering for product lines, and any other development products as well [1, 3].

In [12], four types of reuse are classified as: “data reuse, architecture reuse, design reuse and program reuse”. Apart from them, ten possible reuse approaches of software projects are listed as follows: “architectures, source code, data, designs, documentation, estimation templates, human interfaces, plans, requirements, and test cases” [1]. The first aspect of source code level reuse is the function-level reuse which is for structural languages [2]. Whereas, object-oriented systems support further reuse options in contrast to structural languages. In OO software development, modules can be reused through references, inheritances, any types of templates, object-oriented frameworks or components [7, 11].

There are two main reuse categories: Components-based reuse and transformation-based reuse [13]. Developers choose appropriate components, modify them if necessary and reuse in the first category. Open source software components, commercial-off-the-shelf components, software architecture modules, and product-line components are some examples of reusable components [6].

In the second category, an automated engine produces outputs by transforming appropriate inputs. In transformation-based reuse, the user focuses on specifying an input definition. In order to provide this description, developer reuses the mechanisms of earlier software development efforts, thus reuses processes [12]. Most reuse-engines are samples of transformation-based reuse.

Many software developers create components which share similar functionality or develop components with slight variations. For this reason, documentation of these components has many similarities too. Hence, documentation of software artifacts can also be reused in a systematic way [14].

In this review, unless otherwise stated, what we mean by the term reuse is reuse of any software life cycle product, but not only source code level reuse.

2.2 MEASURING REUSE

Source code level reuse is differentiated as "reused verbatim" or "with slight modification, i.e. less than 25% of lines changed" [1]. Similarly, authors categorize components according to their origins as modules or components reused verbatim, with minor (less than 25%) or major (more than 25%) modifications, or newly developed [11]. Verbatim reuse is reusing an artifact "as-is" in a black-box style.

Reuse ratio or rate is the percentage of compiled units reused verbatim or with minor modifications and in literature the most suggested metric for reuse size measurements is non-comment source line of code (SLOC) [1, 11, 15]. Reuse rate is the size of reused assets divided by the software size: Reused SLOC over total SLOC.

Researchers justify the procedure of source code for reuse measurement by the fact that reuse of assets other than source code is hard to be computed and measuring source code also includes these other assets' reuses [11].

In framework-based development, reuse rate is measured with the rate of the magnitude of what is reused from the framework and the whole product dimensions delivered in a single application. In this measurement, sizes are defined in Object Oriented Function Points (OOFP). Because of the practical problems of SLOC measurements for framework-based development, this metric is excluded. Another reason is that this metric is not appropriate for productivity measurements of framework-based development. Furthermore, researchers suggest OOFPs since this metric is more straightforward to implement [3].

For component-based software (CBS) or lifecycle products other than source code, "size" is not an available metric as it is for standard systems. Therefore, researchers suggest "the number of use cases" (i.e. Business tasks) as an alternate means of size measurements [4]. For the other lifecycle products, we can also use the number of reused products as an indicator of reuse size.

2.3 MEASURING QUALITY

In this section, quality measurement methods are reviewed. First, ISO/IEC 9126 quality standard is described. Then, object-oriented software quality metrics are explained: Theoretical and empirical analyses of the selected metrics are provided.

2.3.1 ISO/IEC 9126 QUALITY MODEL

ISO/IEC 9126 describes the quality model which is based on six characteristics and 27 sub-characteristics of software product quality and additionally one or more metrics to evaluate each of its sub-characteristics. ISO/IEC 9126 Quality Model consists of four parts:

- Part 1: Quality model [5],
- Part 2: External metrics [16],
- Part 3: Internal metrics [17],
- Part 4: Quality in use metrics [18].

Part1 explains the standard specifications of the quality model. Part2 describes the external metrics; similarly Part3 describes the internal metrics and finally Part4 determines quality in-use metrics about assessing the software product.

Evaluation of software products in order to ensure software quality needs is one of the critical processes in the software development life cycle. Software product quality can be evaluated by measuring internal attributes (metrics inside code), or by measuring external attributes (the behavior of the code or system), or by measuring quality in use attributes. The objective of the standard is to keep the desired effect for the product in the context of use [5].

User quality requirements and quality in use requirements specify “external quality requirements” which indicate the level of quality from the external view [16]. External metrics examine these requirements. Internal quality requirements are based on external quality requirements. Internal quality metrics derived from internal quality requirements determine the quality amount of temporary products [17]. The total understanding of internal and external quality is used to predict the estimated “quality in use” for the software product during the development process [18]. Quality in use is the user’s expression of quality and is measured from the usage of software in the user context instead of the software itself.

The quality model categorizes the software quality into six characteristics for both external and internal qualities: Functionality, reliability, usability, efficiency, maintainability and portability. Then, these characteristics are additionally divided into sub-characteristics as shown in Table 2.1. Internal and external metrics measure these sub-characteristics [5].

Table 2.1 – Characteristics and sub-characteristics of ISO/IEC 9126 quality model [5]

Functionality	**Reliability**	**Usability**	**Efficiency**	**Maintainability**	**Portability**
Suitability	Maturity	Understandability	Time behavior	Analysability	Adaptability
Accuracy	Fault tolerance	Learnability	Resource utilization	Changeability	Installability
Interoperability	Recoverability	Operability	Efficiency compliance	Stability	Co-existance
Security	Reliability compliance	Attractiveness		Testability	Replaceability
Functionality compliance		Usability compliance		Maintainability compliance	Portability compliance

The latest quality model released by ISO/IEC is Software product Quality Requirements and Evaluation (SQuaRE). This model covers the software quality requirements with a systems perspective. New standard includes a mechanical parts section including mechanics, hydraulics, electronics, and human processes, etc. Hence, the new system-description investigates a wide range of applications [19]. ISO/IEC 25023 Measurement of the system and software product quality model of SQuaRE collects and replaces ISO/IEC 9126-2 and ISO/IEC 9126-3 revised [20].

In this review, we preferred to refer ISO/IEC 9126 Quality model; as in the relevant literature, this model is still dominant.

In many other works, different quality measurement methods are suggested using the ISO/IEC 9126 Quality model’s characteristics and sub-characteristics. In the rest of this section, we will introduce these quality measures.

Fault detection metric of the Reliability characteristic of the standard is the most reported quality metrics in the literature. This metric focuses on the total number of defects detected in the software product [4, 6]. Defects per non-comment source line of codes value is suggested for measuring this

metric. Any types of errors (Compile, errors, and logic errors) are counted in these measurements [1, 2].

In [15], the following metrics are used for measuring quality:

- Defect rate (It is the number of errors per SLOC),
- The total number of changes (improvement or repair) in the software product during the maintenance period (It is an example of Changeability sub-characteristic of the Maintainability characteristic of the Quality model [6]).

In addition to defect rate, which is put forward by previous works, "rework effort" (an example of Stability sub-characteristic's Change Impact metrics) is also used for measuring the quality [7]. The entire work spent for correcting problems is defined as a rework effort [11].

Isolation and ease of fixing of problems and strength of errors are used as quality indicators by utilizing Maturity and Recoverability characteristics of the ISO Quality model [11]. Additionally, other indicators of software quality are defined: metrics related to software changes i.e. The change density (number of changes per SLOC) and the modified code ratio between software releases by utilizing Changeability characteristic of ISO/IEC 9126 Quality model [3].

Another software quality metric is defined as the difference between development effort and rework effort needed to fix defects detected by the acceptance tests (Maintenance characteristic of ISO/IEC 9126 standard). Furthermore, fault-proneness is designed and used as a quality metric (Fault tolerance metrics of ISO/IEC 9126 standard). Fault-proneness is the probability of defect detection in a software system as a function of structural characteristics of the system [21].

Table 2.2 shows the list of the software quality metrics which we covered in this section with the related ISO/IEC 9126 standard characteristics and sub-characteristics.

Table 2.2 – Software quality metrics with the related ISO/IEC 9126 standard characteristics and sub-characteristics

Metric	**ISO/IEC 9126 characteristic**	**ISO/IEC 9126 sub-characteristic**
Fault-proneness/ Defect rate [7, 15]	Reliability	Fault tolerance
Total number of defects [1, 2, 4, 6]	Reliability	Fault tolerance
Total number of changes [6, 15]	Maintainability	Changeability
Change density [3]	Maintainability	Changeability
Modified code ratio [3]	Maintainability	Changeability
Rework effort [7, 11]	Maintainability	Stability
Difference between development effort and rework effort needed to fix defects [21]	Maintainability	Stability
Isolation and ease of repair of problems [11]	Reliability	Maturity, Recoverability
Strength of errors [11]	Reliability	Maturity, Recoverability

In each of the studies above, there are several metrics to determine the software quality. Data provided from problem reports are one of the few measures of quality applied to most metrics [22]. Therefore, it is quite essential for a company to collect all recorded problems of the software and other parts of the system in "problem reports".

2.3.2 CRITIQUE OF THE ISO/IEC 9126 QUALITY MODEL

There is an excessive amount of criticism about ISO/IEC 9126 standard in the literature, although it is a world-wide standard for measuring software product quality.

In some cases, usage of the ISO/IEC quality model makes quality requirements' management hard. Because, although quantification of some quality requirements is easier; for many other requirements, quantification is difficult, expensive, and sometimes even impossible [23]. In order to use the standard, the developer needs to compute the associated characteristics, sub-characteristics, and metrics; which makes life extremely difficult for the developer. Therefore, the cost of quantifying a quality requirement can be limited by quantifying only some selected metrics.

The standard is not always easy to understand. In a reported experiment, it was found that in the ISO/IEC 9126 quality standard, some characteristics are too abstract to follow, some metrics have more than one meaning, and also some others have overlapping meanings with other metrics [21].

The size of the standard is another problem which makes it difficult to use. In order to use the complete standard, it is essential to detail quality requirements for a software product to the characteristics, sub-characteristics, and metrics levels of the standard [19]. However, in most cases, usage of the standard as a checklist, in order to ensure that all necessary quality requirements are included, is sufficient.

ISO/IEC 9126 standard does not give any instructions about how to decide on metrics, how to obtain them directly, or how to give priority to measures [24]. According to the discussion in [25], most of the quality metrics of the standard cannot be measured directly. For this reason, the user should take the necessary quality characteristics of the standard specific to their area and evaluate them.

The definitions of quality characteristics, sub-characteristics, and metrics in a distinct field may not match with the related definitions in the standard [26]. In order to obtain a satisfactory outcome, the user should perform the process of matching the definitions of the characteristics in their domain and the corresponding definitions in the standard.

Categorization of characteristics in the ISO/IEC 9126 quality model has some fundamental problems. Not recognizing maintainability and reliability characteristics as being originated from design is a serious problem. Apart from that, the standard does not consider some key attributes of design feature such as validity and modularity [24].

The characteristics and sub-characteristics of the model are not internally consistent. In an experiment designed in order to determine correlations of the sub-characteristics of the standard, the researchers found that the standard consists of some high-correlated sub-characteristics [27].

Table 2.3 – Key problems of the ISO/IEC 9126 quality model

Makes quality requirements management hard [23]
Not easy to understand [21]
Does not give any instructions about how to use it [24]
Most of the quality metrics of the standard cannot be measured directly [25]
The definitions of the metrics in the standard may not match with the related definitions in a particular domain [26]
Categorization of the standards has structural problems [24]
The characteristics and sub-characteristics are not always consistent [27]

The key problems of the standard are summarized in Table 2.3. In spite of these complaints, ISO/IEC 9126 standard is still the de facto standard for measuring the quality attributes of a

software product. The users can overcome the challenges of the standard by selecting some necessary metrics and measuring only them in their projects.

2.3.3 OBJECT-ORIENTED SOFTWARE QUALITY METRICS

Above, we discussed the use of metrics in order to determine the quality of the software using ISO/IEC 9126 quality standard. The standard suggests both internal and external metrics for measuring quality, and various other metrics derived using this model. However, since the standard measures a software product via a large number of aspects; it does not provide specific code-based metrics; hence using this standard does not make sense for code-based measurements. Therefore, we decided to use another model for code-based metrics. Below, we will discuss object-oriented software quality metrics.

2.3.3.1 ADVANTAGES OF CODE-BASED METRICS

Software developers and managers can use code-based metrics for different purposes: system level forecasting, prior determination of unsafe components through early measures, and the development of safety design and programming instructions [8, 28]. Furthermore, selecting appropriate metrics from all the alternative metrics support the software developers and managers to identify the quality and structure of the software design and code [29]. In a previously published research, whether a software module would be fault-prone was successfully predicted by linking metrics and earlier software data [29]. Likewise, component defects were predicted using object-oriented metrics obtained from design, code, and requirements [30]. Other authors empirically investigated the usability of object-oriented metrics in forecasting fault-proneness while considering the severity of defects [31].

Furthermore, object oriented metrics support software developers and managers conduct assessments of the necessary development and testing efforts [32]. Moreover, developers analyze and collect metrics in order to validate the software design quality, and hence, help developers improve software quality and productivity [33]. Likewise, metrics are essential especially when the developers decide on a new technology because metrics material provides rapid response in the new feature for software designers and managers [34]. Hence, if metrics are properly used, the costs of the implementation and maintenance reduce, and software product's quality improves significantly. The advantages of software metrics are summarized in Table 2.4.

Table 2.4 – Summary of advantages of software metrics

System level forecasting [8, 28]
Prior identification of unsafe components [29]
Development of safety design and programming instructions [8, 28]
Identify the quality and structure of the software design and code [29]
Prediction of fault-proneness [29-31]
Prediction of development and testing efforts [32]
Validation of the software design quality [33]
Improve software quality and productivity [33]
Rapid response when a new technology is adopted [34]
Reduction of implementation and maintenance cost [34]

2.3.3.2 THEORETICAL AND EMPIRICAL ANALYSIS OF THE OBJECT-ORIENTED METRICS

There are different OO metrics defined in the literature. However, Chidamber and Kemerer's metrics suite for OO design, is the deepest research in OO metrics investigation. These metrics are known as CK metrics, and by far, these are the most popular OO metrics [8].

Chidamber and Kemerer have defined six metrics for the OO design [9]:

- Coupling between objects (CBO),
- Depth of Inheritance Tree (DIT),
- Lack of Cohesion in Methods (LCOM),
- Number of children (NOC),
- Response for a Class (RFC),
- Weighted Methods per Class (WMC).

In literature, the CK metrics have been widely argued, investigated and supported (e.g. [8, 28, 31-36]).

Definitions of the discussed metrics

In this part, we are going to discuss traditional metrics and object-oriented metrics and try to discover which metrics help developers measure design and code quality, and more specifically, which metrics are appropriate for predicting fault-prone software modules. Unless otherwise referenced, the metrics definitions in this section are taken from [32] which constitutes one of the most cited sources in the literature on the subject.

Traditional metrics

Traditional metrics are used in functional development and they also can be easily applied to object–oriented programming [35]. There are various traditional metrics. Complexity and size (i.e. Source Lines of Code - SLOC) are the suggested traditional metrics to use in OO design which we discuss in this part.

Source Lines of Code (SLOC)

It is the sum of counts of non-commented source lines of code in each class. It does not include lines of code in any associated super or sub-classes.

Theoretical foundations of SLOC

When software components exceed a certain size, fault-proneness increases rapidly [8]. Moreover, developers and maintainers use SLOC in order to determine understandability of code.

Object-oriented metrics

We will discuss CK metrics as object-oriented metrics. All CK metrics are defined at the class level. We selected class metrics, although there are other metrics defined on different program entities, e.g. Method, package, program; because the natural unit of object-oriented software systems is class and most metrics have been defined and measured on class level, and class level metrics express the concepts of inheritance, coupling, and cohesion [28].

Cohesion is the strength of co-working of the methods in a class to give a clear in-class characteristic. Cohesion improves encapsulation; therefore object-oriented designs improve cohesion.

Coupling is the strength of relations between two software modules. Classes are coupled when another class' methods are called, or attributes are used.

Inheritance is a relationship between classes in order to reuse previously defined objects, attributes, and operators.

Coupling between Objects (CBO)

It is the total count of the classes to which a class is coupled. It is measured by counting all non-inheritance related classes on which a class depends.

Theoretical foundations of CBO

Redundant coupling affects modular design negatively and restricts reuse of the class; because when a class becomes more independent, its reuse in other applications becomes easier [34]. Changes in different parts of the design affect unpredicted parts of the design if the coupling increases in a class; therefore, maintenance of the class gets harder. A process gets complicated when there is redundant coupling; since it becomes hard to recognize, change or improve. Reduction in complexity of systems is possible if systems are designed with the least possible coupling between classes. This situation results in modularity and encapsulation enhancement. Consequently, CBO indicates classes which are less predictable, less reusable and harder to maintain. Additionally, coupling is also suitable for deciding on testing complexity of the design.

Depth of Inheritance Tree (DIT)

The depth of a class inside the inheritance chain is the highest step count from the class itself to the root of the tree.

Theoretical foundations of DIT

As DIT of a class increases, the number methods the class is expected to inherit also increases; therefore predicting its behavior becomes more complex [34]. Deeper trees cause more design complexity since they include more methods and classes; on the other hand deeper trees are more promising for reuse of the inherited methods. Henceforth, higher percentages for DIT show a higher degree of reuse; however increased complexity.

Lack of Cohesion in Methods (LCOM)

It is the total number of methods in a class which have no common attributes, minus the number of methods which have common attributes [8].

Theoretical foundations of LCOM

LCOM measures the unlikeness of methods in a class. A highly cohesive module stands alone; since high cohesion is an indication of good class separation. Furthermore, low cohesion increases complexity; and high cohesion is an indicator of simplicity and high reusability.

Number of Children (NOC)

It is the number of subclasses of a class in the inheritance chain.

Theoretical foundations of NOC

NOC indicates the possible effect of a class on the software design and the whole system. As NOC children increases, the risk of improper abstraction of the parent class also increases. On the other

hand, as this metric increases, the reuse level increases since inheritance is a reuse type. Additionally, a class with an excessive number of children needs more testing [34].

Response for a Class (RFC)

It is the total count of all methods which are called 1) as a reaction to a message received from a class or 2) by other methods of the class.

Theoretical foundations of RFC

This metric analyzes the sum of the complexity of a class through the number of methods and the amount of communication with other classes. As RFC increases, the complexity of the class also increases. Additionally, as RFC increases, the testing and debugging effort required also rises [34]. Therefore, classes with high RFC are more complex and less predictable.

Weighted Methods per Class (WMC)

It is a standard complexity metric. It is a weighted-count of the methods implemented within a class. Some authors weight the count with cyclomatic complexity; however others do not weight the count [8]. In this study, we will use cyclomatic complexity; since WMC simply becomes the number of methods if the count is not weighted.

Theoretical foundations of WMC

The total count of methods and the complexity of these methods help to predict the total time and effort needed to develop and maintain the class[34]. Furthermore, as the number of methods in a class increases, the possible impact of it on its children also accumulates. Classes with a greater number of methods are more likely to be application-specific i.e. The possibility of reuse reduces.

After the descriptions above, we provide the table below (Table 2.5). In this table, we tabulate the related object-oriented concepts for each CK metric.

Table 2.5 – CK metrics and object-oriented Concepts

Metric acronym	**OO Concept (*: Primary concept)**
CBO	Coupling*/Complexity
DIT	Inheritance*/Complexity
LCOM	Cohesion*/Complexity
NOC	Inheritance
RFC	Complexity*/Coupling
WMC	Complexity

2.3.3.3 EMPIRICAL LITERATURE ON CK METRICS AND SOFTWARE QUALITY

The effects of CK metrics on software quality, especially fault-proneness, have been widely argued in the literature. Many field experiments have been reported on these metrics. In this part, some of these studies will be reviewed, the relationship between the software quality (i.e. Fault-proneness, modified code ratio, productivity and rework effort) and the CK metrics will be discovered, and the results will be tabulated.

In [31], the researchers made an analysis based on a public data set. They analyzed the data set by measuring the CK metrics and additionally the SLOC metric. Additionally, they compared their

results with fault severity. They have concluded most of these metrics are statistically correlated to fault-proneness of classes across fault severity: WMC, RFC, CBO, LCOM and SLOC are related to fault-proneness across all severity; however DIT and NOC are not related to fault-proneness.

Subramanyam and Krishnan conducted a literature survey on empirical analysis of CK metrics [33]. In this analysis, effects of CK metrics on software quality were displayed using previously published studies. In this analysis, the practitioners and the researchers employed fault-proneness, modified code ratio, productivity and rework effort as indicators of software quality.

Another review about empirical results on CK metrics is conducted in [31]. Similar to work done in [33], the researchers also made a literature review. However, in this study, only fault-proneness was compared among the reviewed empirical results. The CK metrics and additionally the SLOC metric were employed in this study.

The empirical results of [31], and [33] were tabulated below in Table 2.6. We categorized the data based on the metric types, and effects of these metrics on fault-proneness, modified code ratio, productivity, and rework effort are indicated using a plus ("+") or a minus ("-"). A plus means the metric is correlated with the depicted quality metric; however a minus means the metric is not correlated with the depicted quality metric. The space (" ") means the reviewed study has not measured the related metric for that quality metric. In this table, each column under a quality metric indicates an individual empirical study reviewed in [31], or [33].

Table 2.6 – Summary of empirical results in [31] and [33] on CK metrics and software quality

Metric acronym	Effects on																				
	Fault-proneness																	Modified code ratio		Productivity and rework effort
	[33]					[31]												[33]		
CBO	+	+	+			+	+	-	+	+	+	+	+	+	+	+	+	-	+	+
DIT	+		+	+		+		-	+	-	+	+	-	+	-	-	-	+		-
LCOM	-	+	-										+		+	+	+	+		+
NOC	+		+	+		+		-	+				+	-		+	+	+	-	-
RFC	+	+	+		+	+	+	+	+	+			+	+	+	+	+	+	+	-
WMC	+		+		+	+		+	+	+	+	-	+	+	+	+	+	+		-
SLOC									+	+	+	+	+	+	+	+	+			

The results tabulated in Table 2.6 signify that, in most of the studies, the measured metrics correlate with software quality. Using the data in Table 2.6, we calculated the correlation percentages of these metrics and tabulated them in Table 2.7. This table indicates that **all measured metrics are correlated with fault-proneness**. Modified code ratio, productivity, and rework effort metrics are measured in a limited number of studies; therefore we were unable to derive a strong result as in the fault-proneness case.

In Table 2.8, we tabulated the correlation of OO concepts with respect to the only strongly correlated quality metric which we have found in the above analysis, i.e. Fault-proneness. As shown below, we have observed that **OO concepts of coupling and complexity are strongly correlated with fault-proneness**.

Table 2.7 – Correlation percentages of the metrics with software quality (derived from the empirical data in [31] and [33])

Metric acronym	Effects on					
	Fault-proneness		Modified code ratio		Productivity and rework effort	
	Number of correlated studies / Total studies	% Correlated study	Number of correlated studies / Total studies	% Correlated study	Number of correlated studies / Total studies	% Correlated study
CBO	14/15	93,33	1/2	50,00	1/1	100,00
DIT	8/14	57,14	1/1	100,00	0/1	0,00
LCOM	5/7	71,43	1/1	100,00	1/1	100,00
NOC	8/10	80,00	1/2	50,00	0/1	0,00
RFC	14/14	100,00	2/2	100,00	0/1	0,00
WMC	13/14	92,86	1/1	100,00	0/1	0,00
SLOC	9/9	100,00	NA	NA	NA	NA

Table 2.8 – Correlation of OO concepts with fault-proneness (derived from the empirical data in [31] and [33])

OO Concept	Effects on fault-proneness		
	Metrics	Number of correlated studies / Total studies	% Correlated study
Coupling	CBO, RFC	28/29	96,55
Complexity	CBO, DIT, LCOM, RFC, WMC	54/64	84,38
Cohesion	LCOM	5/7	71,43
Inheritance	DIT, NOC	16/28	57,14

2.4 EFFECTS OF REUSE ON QUALITY

In this section, a literature review will be presented on the effects of reuse on software quality. First, the reuse effects on general software will be introduced, and then reuse effects on embedded software specifically will be summarized.

Increase of software products' quality usually arises when they are reused; because of the reason that as software artifacts are reused, the collection of the defect corrections in sequential versions brings about a higher quality [2]. A high degree of reuse is found to be correlated with a low defect rate, and reduced development effort when the earlier industrial measurements are interpreted [1]. Another industrial research has displayed a similar relationship between reuse and quality: The subjective quality values assigned by software developers (i.e. Quality ratings) are found to be correlated with reuse level and reuse frequency of the software product [15].

Practitioners have reported results of an experiment employing the earlier software product details of a large telecom system built by a global company in [6]. In this experiment, a dozen of product releases are examined, and the following data were collected: Number of all detected defects, size of the components, and size of modified software. The conclusions of the study are reported as:

- The defects detected in the reused components are given higher priority. Therefore, defect rates of reused components are found to be lower than of the non-reused components,
- Non-reused components are found to be more defect-prone than reused components,
- The number of modifications in reused components is found to be lower, when compared to the non-reused components, although having different requirements for various products.

In a literature review which is about the effects of reuse on software quality for various components; it is reported that, systematic reuse (either verbatim reuse or reuse with insignificant changes or reuse with new code) is related to a convincing reduction in defect rate [11].

In [3], an industrial experiment which is about software production using frameworks (An effective reuse approach), is presented. In this research, rework efforts of two software production are compared: A classical production and a framework-based production. The practitioners report that products developed with frameworks have higher quality than the products developed traditionally, due to the learning effect.

The summary of what we mention about the reasons of quality improvements when the components are reused, are summarized below in Table 2.9.

Table 2.9 – Several reasons of the quality rise when components are reused

Components to be reused are designed more carefully [11]
Components to be reused are better tested [11]
The defects detected in the reused components are given higher priority [6]
The collection of the defect corrections in sequential versions [2]
Experienced software developers due to repetition of similar works [3]

2.4.1 REUSE AND QUALITY IN REAL-TIME EMBEDDED SOFTWARE SYSTEMS

In this sub-section, the reuse and quality relations of embedded software systems will be summarized.

For embedded software, quality features do not only include functional requirements; but also include performance, reliability, safety, and maintainability requirements [37]. Additionally, embedded software systems require more complex functional and quality requirements as a result of their embedded nature; since embedded systems work in critical and sometimes dangerous environments. In addition, it is essential for these systems to predict, determine and integrate their quality demands as early as possible in the software development life cycle, and additionally it is important to capture detected defects against predicted defects; in order to reduce the cost impacts of the shortcomings of these requirements [38]. Therefore, these systems call for new software development methods [37]; such as:

- Model-driven development,
- Numerical modeling for performance and security analyses,
- Automatic design and code checking,
- Automatic testing,
- Static code analyses of performance, security, and memory.

Employing design patterns and frameworks are another way of systematic reuse for embedded software development. New models are suggested in the literature based on design patterns including analyses, and documentation [39]. Measurements of productivity and quality are achieved from the collected data through reuse of the patterns during the software development. Practitioners reported in a case study that correct usage of components and design patterns improves effort, time, and costs of embedded software projects [39].

2.4.1.1 PERFORMANCE REQUIREMENTS OF EMBEDDED SYSTEMS

In embedded systems, performance requirements arise from the demand of the effective employment of the hardware resources. Consequently, performance constraints are more critical and more influential in embedded systems than in other general purpose software systems such as IT systems. However, large abstraction layers, common platforms, and virtualization methods require many resources [40]. Therefore, these techniques are not suitable for most embedded systems.

The key metrics used in embedded software systems are physical metrics: memory, performance, power, energy, and size [41, 42]. Furthermore, real-time systems should satisfy non-functional requirements such as timeliness, and reliability [43].

Above, in the "Measuring Quality" section, we summarized the use of code-based metrics for measuring software quality. In addition, relationship of software quality and the OO concepts are also discovered. However, while traditional software quality relates to these OO concepts such as abstraction, reuse, coherence, and coupling; these concepts generally oppose to physical metrics of embedded systems such as performance, and memory [44].

Quality metrics, when employed successfully in general purpose software systems, are known to improve software quality, because of the progresses in the reuse and total effort [44]. However, the case is not the same for the embedded systems since embedded systems do not benefit from these improvements due to the strict performance constraints [41]. Nonetheless, some losses in reuse or maintainability cannot be avoided to gain a better performance for embedded systems [42].

It is shown that object-oriented programming can considerably increase both execution time and resource consumption in embedded software systems. Although OO design improves maintainability and portability requirements of the embedded software, it costs performance, memory, and size [45]. Nevertheless, in the literature, researchers employed OO metrics together with physical metrics, during the design phase of the software development; in order to achieve balance between these metrics [44]. It is reported that by using the software quality metrics, it is possible to improve performance metrics of embedded software with a small decrease in code reuse [41].

While many other studies contradict, it is reported in [46] that, reuse improves performance in terms of memory and speed by employing the product-line approach. However, most of the improvement is due to recovering the previous design and making optimizations on it.

Consequently, in order to employ and maintain reuse in embedded software systems, the techniques suggested for general purpose software systems such as object-oriented design, OO concepts, and abstraction should be implemented. However, there are some drawbacks which are summarized in Table 2.10. The major one is the loss in performance metrics.

Table 2.10 – Summary of the drawbacks of OO programming in embedded software

Abstraction layers, common platforms, and virtualization methods require many resources [40]
OO concepts generally oppose physical metrics of embedded software [44]
Object-oriented programming can considerably increase both execution time and resource consumption in embedded software [45]
OO design costs performance, memory, and size [45]

CHAPTER 3

RESEARCH CONTENTS

In this chapter, firstly the research hypotheses of the study are stated. Then, the software teams and the related projects, which we have taken measurements from, in Aselsan Defense System Technologies Group (SST) - Software Engineering Department (YMM), are explained. Then, the case studies, which are designed in order to investigate reuse and quality relationships in each team, are presented. Finally, the research hypotheses are verified by discussing the measurements and some suggestions regarding the reuse infrastructure and process, to enhance the benefits of software reuse in Aselsan are formulated.

3.1 RESEARCH HYPOTHESES

In this thesis work, we examined the software reuse and quality connections in Aselsan SST-YMM. For this study, we defined four different hypotheses (Table 3.1) and in order to confirm these hypotheses; we designed three different case studies for each team in Aselsan.

In the first study, object-oriented quality metrics and physical metrics of an embedded system developed in Aselsan SST-YMM were compared with changing reuse rates. Reused SLOC over total SLOC is used for reuse rate. OO metrics are selected from CK metrics, and number of cycles is the only physical metric measured.

Table 3.1 – The hypotheses defined

Hypothesis 1 – Code-based Quality:
The quality of software products is improved as reuse rates of the products increase.
Hypothesis 2 – Performance of Embedded Software:
Performance of the embedded software products decays as reuse rate of the products increase.
Hypothesis 3 – Fault-proneness:
The number of defects detected in components decreases as these components are reused in various products.
Hypothesis 4 – Productivity:
The productivity rates of products increase as the reuse rates of these products increase.

In the second study, we investigated a software product line (SPL) in Aselsan SST-YMM and compared changing defect counts in common-components as components are reused in various products. Moreover, the productivity rates of these consecutive products are measured. As a measure of productivity, number of requirements over total effort is used.

In the third study, we investigated another SPL in Aselsan SST-YMM and compared changing productivity rates for different products as reuse rates of these products rise. Similar to the first study, reused SLOC over total SLOC is used for reuse rate. Additionally, productivity was defined as SLOC over total effort. Furthermore, the performance of a critical scenario is measured, before and after employing the product line approach and the impact of reuse on performance for this case is noticed. CPU usage and delay in the scenario are the performance metrics used in this study.

The summary of the case studies and the metrics employed is displayed in Table 3.2.

Table 3.2 – Case studies and the corresponding metrics employed

Metrics / Case Study	Code-based Quality	Fault-proneness	Performance	Productivity
Case Study 1	✔		✔	
Case Study 2		✔		✔
Case Study 3			✔	✔

3.1.1 JUSTIFICATION OF THE HYPOTHESES

Hypothesis 1:

In the literature, it was argued that code-based metrics are strongly related to predicting fault-proneness of software products (see "Theoretical And Empirical Analysis Of The Object-Oriented Metrics" sub-section in Chapter 2). Hence, there is a positive correlation between these metrics and software quality; since fault-proneness is a measure of software quality (see "Measuring Quality" section in Chapter 2).

Hypothesis 2:

Drawbacks of object-oriented programming in embedded software are widely discussed in the literature. In order to employ and maintain systematical reuse in embedded software systems, the object-oriented concepts, such as abstraction, coherence, and coupling, should be extensively employed in these systems. However, these concepts weaken physical metrics of embedded systems such as performance, and memory (see "Reuse and quality in real-time embedded software systems" sub-section in Chapter 2).

Hypothesis 3:

As promoted in the literature fault-proneness is a widely used measure of software quality (see "Measuring Quality" section in Chapter 2). Moreover, the product line approach is expected to produce a reduction in fault-proneness in response to an increase in the reuse of components. Additionally, the previous studies show that as components are reused in more products, the defect counts for these components decays noticeably (see "Effects Of Reuse On Quality" section in Chapter 2).

Hypothesis 4:

In the literature, it is suggested that, the product line approach and, therefore, the systematic reuse, cause increase in productivity in response to increase in reuse rate due to reduction in the design effort and fault-proneness (see "Measuring Quality" section in Chapter 2). Additionally, the increase in the software quality causes a reduction in rework effort, which is correlated to productivity.

Consequently, as analyzed above, verification of these hypotheses is appropriate in order to investigate reuse and quality relationship in an industrial setting.

3.2 GENERAL INFORMATION ABOUT THE SOFTWARE TEAMS

Throughout this study, we worked with three different software teams.

The first team provided us with code-based measurements from their software. We compared three modules used in their software in terms of object-oriented software metrics and performance metrics with changing reuse rates.

We obtained defect counts of various components in different products from the second team and compared defect counts as the components reused in consecutive products. Furthermore, we acquired total efforts of these products and compared productivity rates regarding the percentages of the reused and non-reused requirements.

The third team provided us with productivity rates of various products they developed and we compared these rates with changing reuse-rates. Additionally, we obtained performance metrics of a critical scenario before and after application of the product-line approach.

These teams were selected due to their accessibility and the possibility of communication with them arising from the author's employment. As all projects undertaken in Aselsan have a confidential nature, not many other teams would be approachable and would be able to provide the measurements necessary for the present study. All interviews and data reported in this thesis have been authorized by Aselsan's responsible staff.

3.2.1 AIR DEFENSE WEAPON SYSTEMS TEAM

Air Defense Weapon Systems (HSSS) team develops real-time embedded software for various air defense weapon systems produced in Aselsan SST-YMM. The team creates software in C++ language.

HSSS team develops software for systems which have many common unit interfaces and common features [47]. However, the systems often have different unit interfaces, different features, and even work on different processors. HSSS team owns a pool of reconfigurable components which they developed for various systems. They reconfigure the necessary components and reuse them through this pool.

HSSS team has a reference layered architecture which is shown in Figure 3.1.

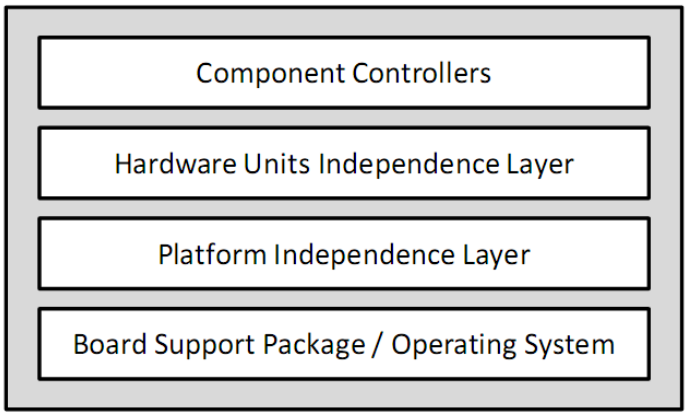

Figure 3.1 – HSSS Reference Layered Architecture

Processor card and operating system independence layer provides the ability for the HSSS team to work on different real-time operating system (RTOS) versions and also operating systems other than RTOS such as Windows operating systems. In this layer, common interfaces are developed for various operating systems and processors.

The hardware units' independence layer provides common interfaces for different brands and models of hardware units i.e. Same interfaces for different camera types used in HSSS projects.

3.2.2 TECHNICAL FIRE SUPPORT SYSTEMS TEAM

Tactical Fire Support Systems (TADES) team develops command and control software for technical fire support systems using TADES Software Product Line[48]. The team creates software in .NET environment.

TADES SPL is a composition-oriented SPL. In TADES SPL, there are two types of components: common platform components and product specific components. Common platform components are reused in various projects, and product specific components are developed for every single product. A product developed via TADES SPL has various numbers of common platform components and product specific components in it.

Members of the team are owners of one or more components in TADES SPL. In this SPL, component owners define a component as a composition of abilities which have properties in common.

Before developing a product using this SPL; the product owner chooses the components with appropriate versions, which will be in this product, out of the common platform components. The configurations of these common components are set. Then the product-specific components are defined, and owners of these components are assigned. If any of these components are thought to be generic enough to be reused in other products, then moved into the common platform.

All common-components have their own, separate software requirements specification (SRS) documents. Product specific requirements are written in product-level SRS documents. In product-level SRS documents, there exist no common-components related requirements, but a reference to these documents.

3.2.3 FIRE CONTROL SYSTEMS TEAM

Fire Control Systems team develops real time embedded software for fire control systems using Fire Control Systems Reference Architecture (SSRM) Software Product Line. The team creates software in C++ language.

SSRM SPL uses Feature Oriented Reuse Method (FORM) and also is a composition-oriented SPL. The capabilities, which can be included in the product line or excluded from the product line, are modeled as separate components. In SSRM SPL, there are various common-platform components reused in different products and also there are product-specific components.

The components in SSRM SPL are grouped as follows [49]:

- Missions,
- Capabilities,
- Software manager,
- External interface,
- System environment,
- Operating environment.

3.3 CASE STUDY 1: AN EXPERIMENT FOR COMPARING OO SOFTWARE QUALITY METRICS AND EMBEDDED SOFTWARE PERFORMANCE METRICS WITH CHANGING REUSE RATE

Below, in the first sub-section, the software modules used in case study 1 are explained. Then, the OO software quality metrics and performance metrics used in this study are discussed. In the last sub-section, how the corresponding metrics are measured is explained, and these measurements are tabulated.

3.3.1 SOFTWARE MODULES USED

In this study, three different software modules are investigated. All modules are implemented, using C++, by HSSS team.

The first module is the User Command and Control Interface of an embedded system (Figure 3.2). This module opens up a TCP/IP socket interface and the external users of the system connect and control the system through this interface.

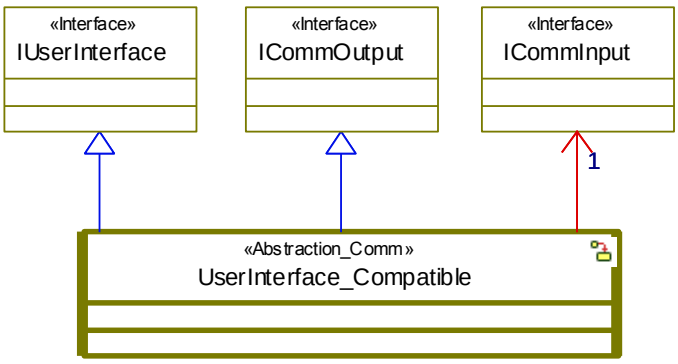

Figure 3.2 – Class diagram of the 1st module

This module is responsible for getting commands through socket, parsing them, and entering this command into the rest of the system. This module also sends updates and requests to external users; it formats messages in bytes level and sends through the socket.

In order to obtain socket updates, this module (UserInterface_Compatible) inherits ICommOutput interface; in order to send messages through socket this module has an association with ICommInput interface and in order to receive commands from the system it inherits IUserInterface interface. The sequence diagrams for the message receiving and sending scenarios are shown in Figure 3.3 and Figure 3.4, respectively.

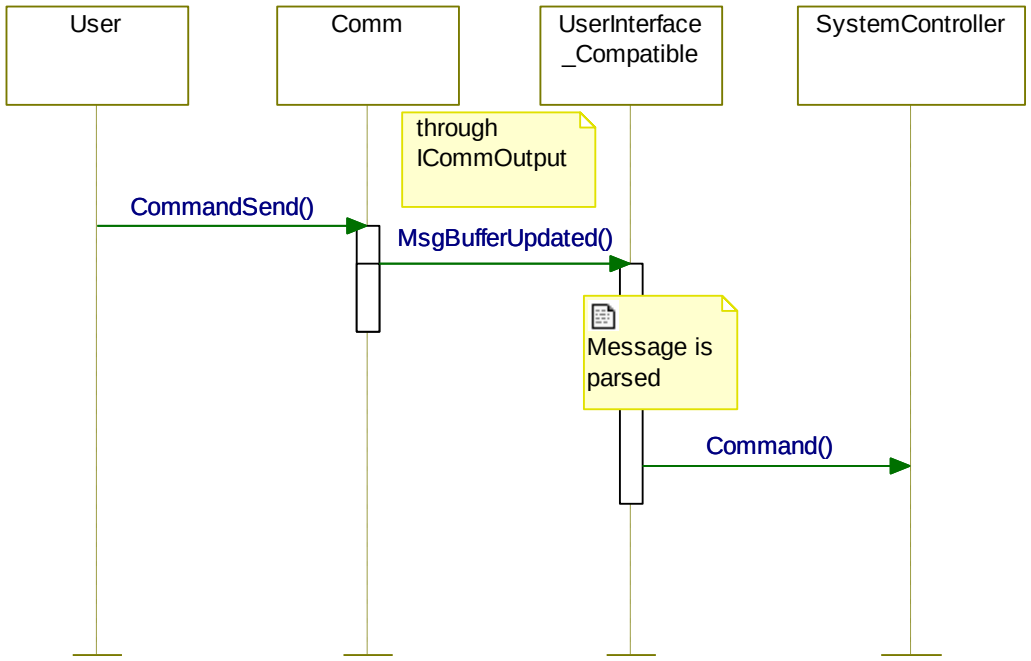

Figure 3.3 – Sequence diagram of the user command scenario of the 1st module

Since message taking and sending include parsing and formatting in bytes level, it is time-consuming adding a new message to the command interface. In order to do this message management easier, a new reuse engine was developed and used by the developers in HSSS team. This engine creates a middleware for all parsing and formatting parts of the socket interface. The user of this tool just decides on the interface functions, and the auto-created middleware is inserted into the main code.

This reuse engine is an example of transformation-based reuse (see “Reuse Types” section in Chapter 2) since this engine reuses all the process needed for message sending, receiving, and parsing. The developed middleware can be used without any changes in many different systems. Additionally, this engine can also be used by the test engineers and reduces test efforts. Furthermore, the documentary outputs of this engine can be used for documentation purposes as an example for documentation reuse.

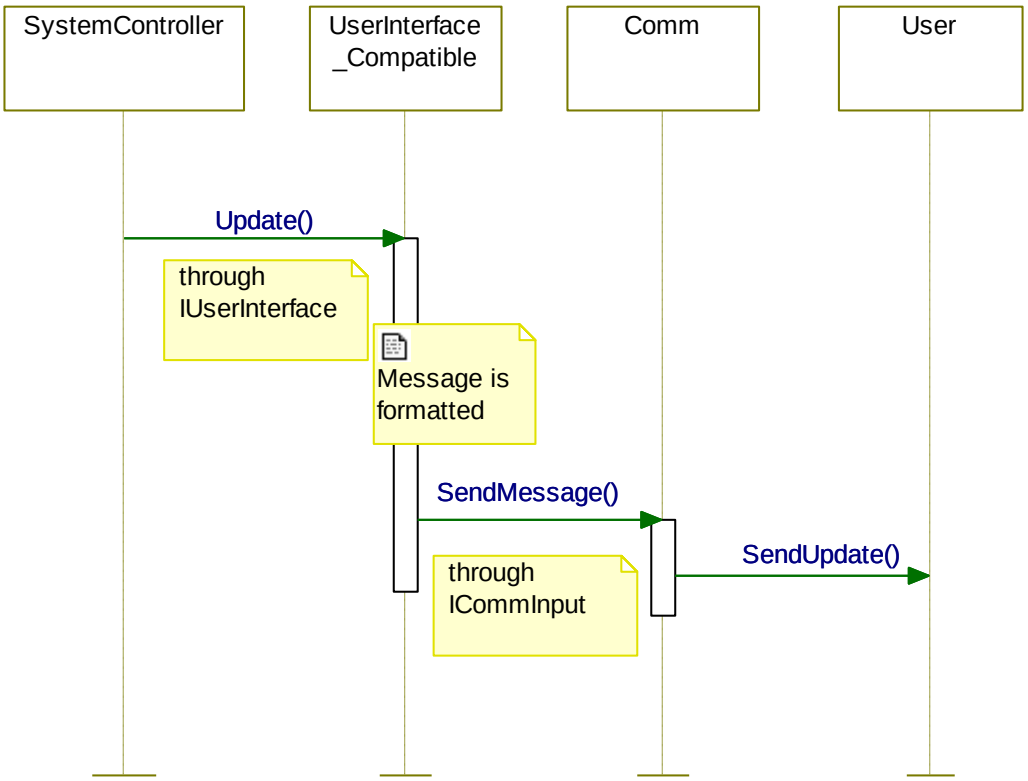

Figure 3.4 – Sequence diagram of the system update scenario of the 1st module

The second module does the same work as the first module, but is implemented using the above-mentioned reuse engine (Figure 3.5).

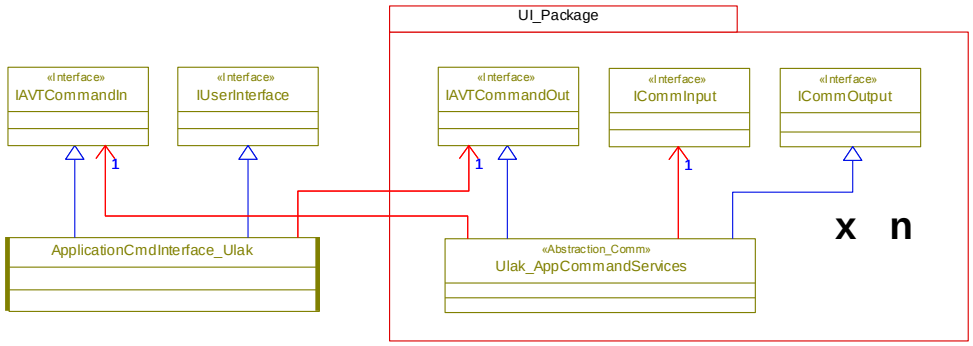

Figure 3.5 – Class diagram of the 2nd module

In Figure 3.5, the classes in UI_Package are the products of the reuse engine. For every message, a sub-class is created under Ulak_AppCommandServices (like Ulak_AppCommandService1 in Figure 3.6), and these classes inherit ICommOutput interface in order to get socket messages and have associations with ICommInput interface in order to send socket messages and inherit IAVTCommandOut interface in order to communicate with the outer system through ApplicationCmdInterface_Ulak class. ApplicationCmdInterface_Ulak class inherits IAVTCommandIn and IUserInterface interfaces in order to communicate with the system and the reused classes.

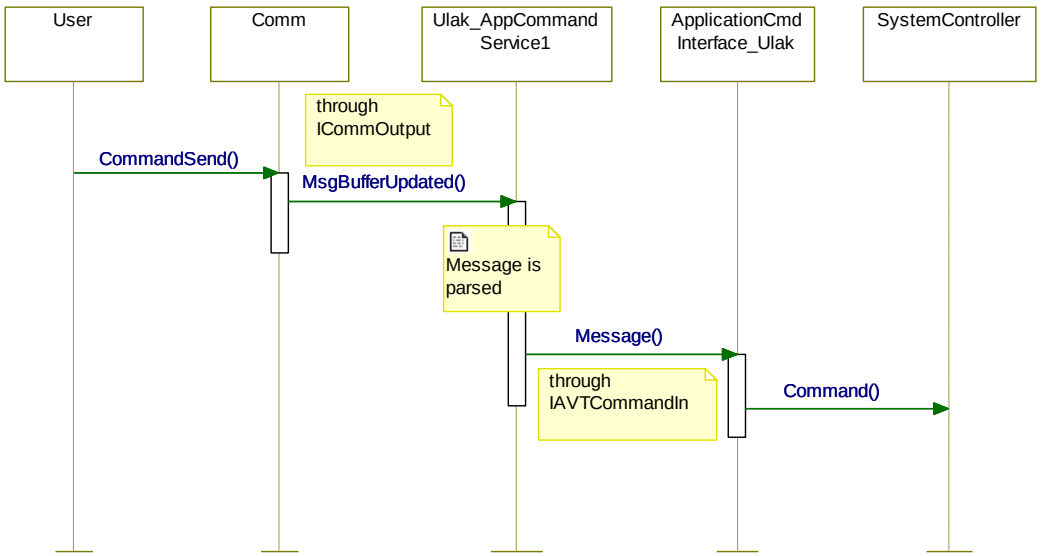

Figure 3.6 – Sequence diagram of the user command scenario of the 2nd module

In Figure 3.6 and Figure 3.7, the sequence diagrams of the message receiving and sending scenarios of the second module are shown, respectively. Ulak_AppCommandService1 is a product of the reuse engine. For every message in the interfaces, there is a dedicated class like Ulak_AppCommandService1, and these different classes are auto-generated.

The third module is also a product of the reuse engine; however, its developers are different from the second one. It does a similar work as the second one, but it is a small module: It has fewer messages than the second one.

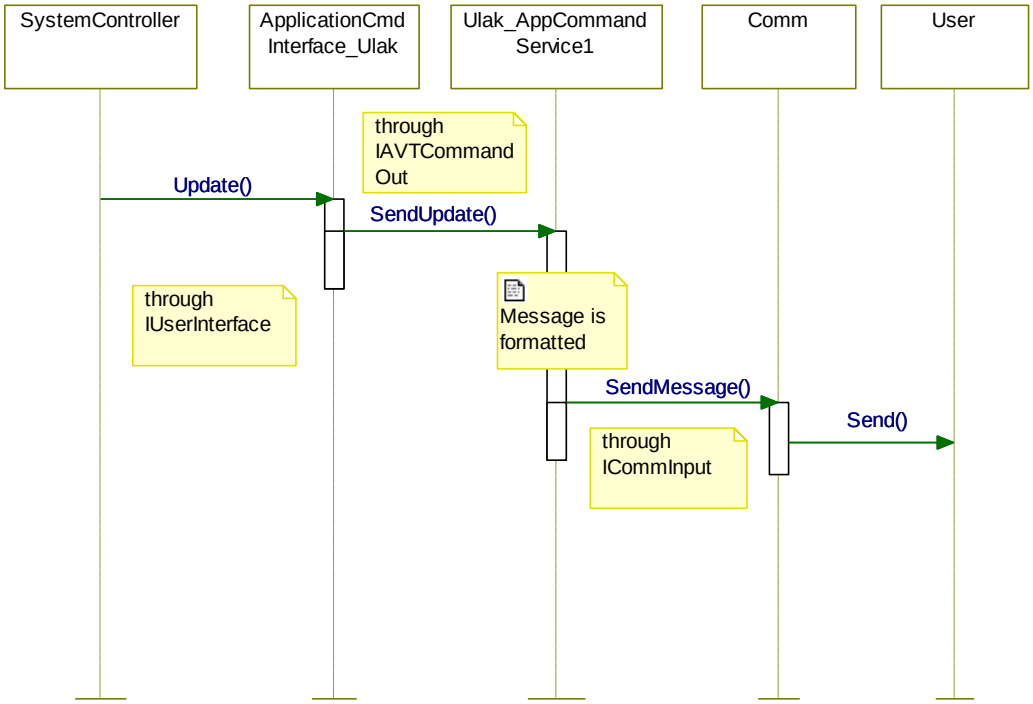

Figure 3.7 – Sequence diagram of the system update scenario of the 2nd module

3.3.2 CHOOSING OO METRICS FOR THE EXPERIMENT

In Chapter 2 (Theoretical And Empirical Analysis Of The Object-Oriented Metrics sub-section), the theoretical and empirical results on popular CK metrics are shown. Moreover, the role of the measures defined in the CK metric suite in explaining the object-oriented software quality at a class level is identified. Therefore, CK metrics are appropriate for this experiment. Because, we want to compare the software quality of the modules, and we have seen that the CK metrics are suitable for measuring the software quality.

In addition to CK metrics, we decided to employ additional complexity metrics in the experiment. In the OO environment, definite design concepts such as inheritance, coupling, and cohesion have been argued to embrace complexity [33]. Also, complexity metrics have been shown to correlate with defect density in a number of case studies [29]. Hence, we selected additional complexity metrics from [44], to strengthen the study.

3.3.2.1 ADDITIONAL COMPLEXITY METRICS

McCabe Cyclomatic Complexity (McCabeCC)

It counts the number of flows in a part of the code. A high value of this metric means the software is complex or at least it has many different flows. It has also been shown that McCabe's cyclomatic complexity is correlated with defects and maintenance changes in a software system [33].

Nested Block Depth (NBD)

It is the extent of nested blocks of code. More nested blocks lead to worse readability and more complex solutions.

Percent Branch Statements (% Branches)

Statements that create a break in the sequential execution of statements are counted separately. These are the following: if, else, for, while, break, continue, goto, switch, case, default, and return.

3.3.3 PHYSICAL METRICS USED IN THE EXPERIMENT

In Chapter 2, some physical properties of embedded software products to be measured in embedded system design are mentioned such as performance, memory, energy, power, size, and weight etc. In the experiment, the only available physical metric for evaluation is the number of cycles for the same work to be done for each module. Therefore, we used this metric to investigate the change of performance metrics for embedded systems with changing reuse rates.

3.3.4 MEASUREMENT OF METRICS

The necessary metrics were measured with the help of the developers in HSSS team. The reuse rates of the modules were calculated by using reused non-comment line of codes, and total non-comment line of codes. The calculated reuse rates are shown in Table 3.3.

Table 3.3 – Calculated reuse rates

Metrics Type	Module 1	Module 2	Module 3
% Reuse rate	0	81	52

In this work, the source and header files are created for the modules from the UML models. Then, we conducted a free Internet search in order to select a metrics tool. We selected the following free static analysis tools for C++ programs: SourceMonitor [50], and CCCC (C and C++ Code Counter) [51].

SourceMonitor is a freeware program which identifies the relative complexity of the software modules. It measures metrics for source code written in C++, C, C#, Java, Delphi, or Visual Basic. Using SourceMonitor, we measured the following metrics: LCOM, SLOC, McCabeCC, NBD and % Branches.

CCCC is an open source command-line tool. It analyzes C++ and Java files and generates reports on various metrics, including Lines of Code and metrics proposed by Chidamber and Kemerer. Using CCCC, we measured the following CK metrics: CBO, DIT, NOC, and WMC.

Using both metrics tools, we were unable to measure RFC; therefore, we skipped this metric from the analysis. The measured OO metrics are given in Table 3.4.

Table 3.4 – Extracted software quality metrics

Metrics Type	Module 1	Module 2	Module 3
Coupling Between Objects (CBO)	2,311111	1,944444	2,166667
Depth of Inheritance Tree (DIT)	0,333333	0,651166	0,066667
Number of Children (NOC)	0,422222	0,686047	0,133333
Weighted Methods per Class (WMC)	4,777778	2,166667	2,9
Lack of Cohesion of Methods (LCOM)	0,0820	0,0495	0,0823
Source Lines of Code (SLOC)	2819	4117	765
McCabe Cyclomatic Complexity (McCabeCC)	2,49	1,30	1,59
Nested Block Depth (NBD)	1,71	0,84	1,10
% Branches	18,2	7,4	8,9

Extracted physical metrics for the modules is given in Table 3.5. The only performance metric calculated is the number of cycles (NoCycles) for the modules to receive the command from the system and send the corresponding data. We calculated this metric by using the associated static BSP function which is for taking time measurements.

Table 3.5 – Extracted physical metrics

Physical Metrics	Module 1	Module 2	Module 3
NoCycles	7914	9756	9648

3.3.5 DISCUSSION OF THE MEASUREMENTS

Metrics based results cannot be compared when different metrics tools are used [28] since for different metrics tools, there will be differences in calculation techniques, assumptions in measurements etc. However, comparing the results of different software modules using the same metric tools makes sense and we employed this method.

Below, we group the measured metrics according to their primary OO concepts i.e. Size, inheritance, coupling, and complexity, and compare these metrics with respect to increasing reuse rates. In order to be able to draw figures with increasing reuse rates, we put the modules in 1,3, and 2 orders.

Figure 3.8 shows the change of size metrics with changing reuse rate. Although, there is a correlation between the reuse engine outputs, i.e. Module 2 and 3, it is clear that, the size metrics are uncorrelated with the reuse rate when module 1 is also taken into account.

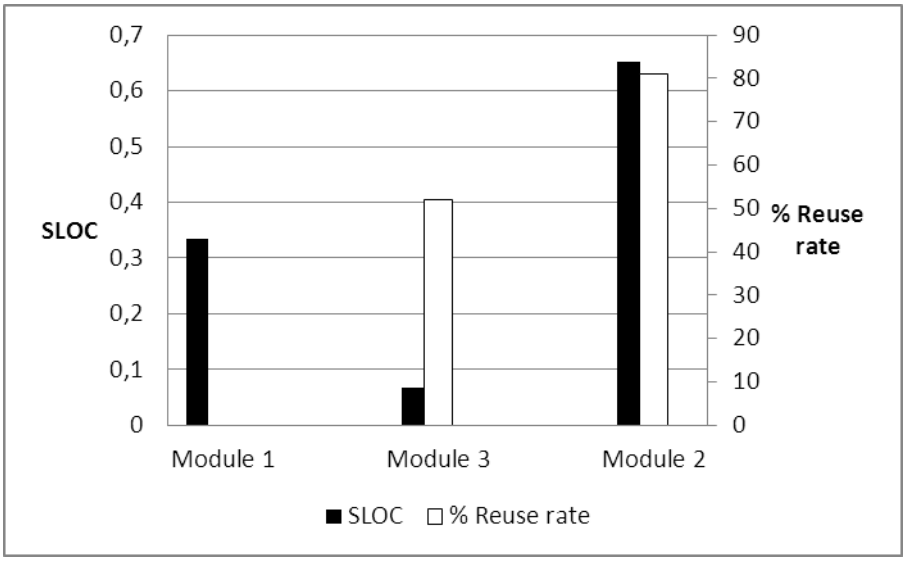

Figure 3.8 – Comparing size metric and reuse rate

In Figure 3.9, we see the comparison of inheritance metrics (DIT and NOC) with changing reuse rates. From the figure, it is easy to notice the connection between the inheritance metrics and size metrics. Similarly, we do not detect a correlation between reuse rates and inheritance metrics, although there is a connection between modules 2 and 3.

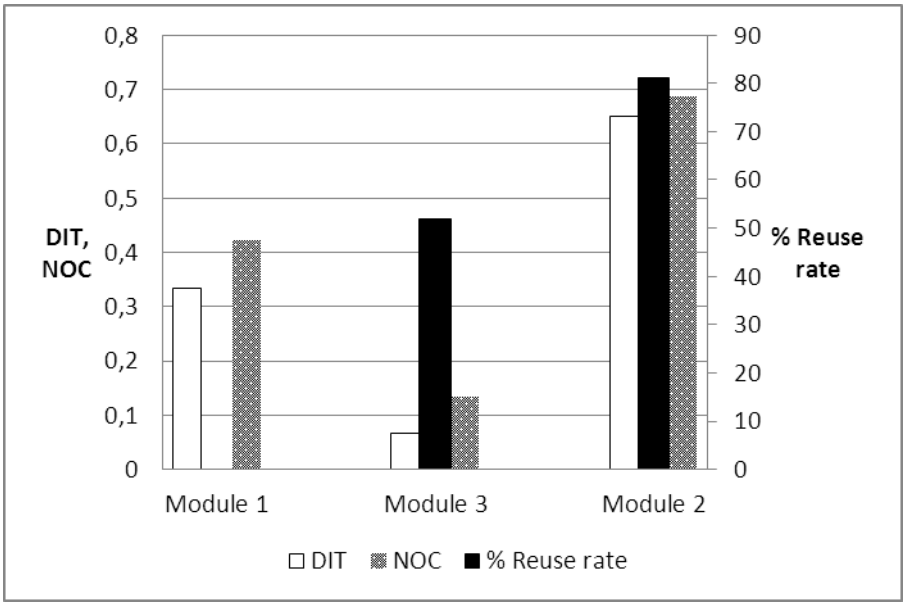

Figure 3.9 – Comparing inheritance metrics and reuse rate

In Figure 3.10, we see the comparison of coupling metric (CBO) with reuse rate. In this figure, we can observe that there is an improvement (reduction of metric) in terms of coupling as reuse rate increases. The change of the architecture for reuse and introducing interface classes in the system make the system less coupled.

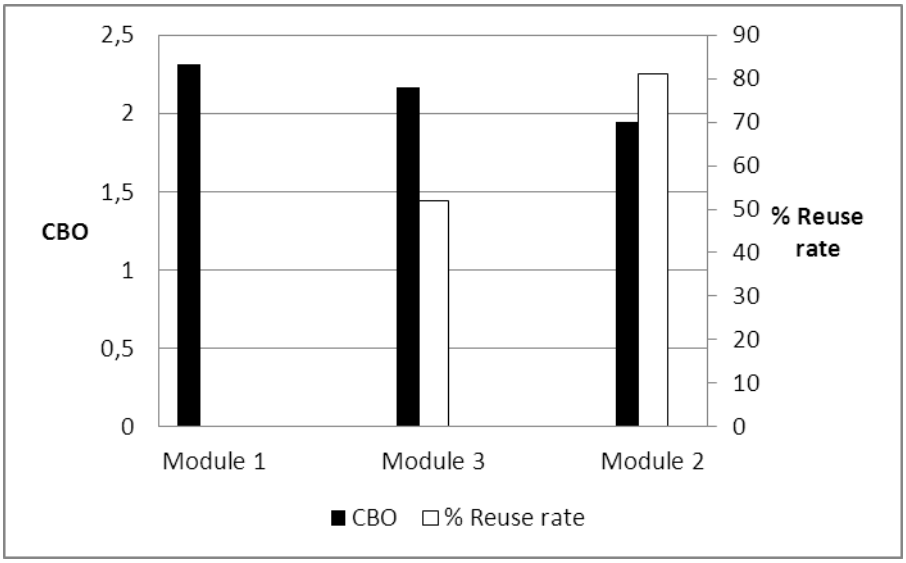

Figure 3.10 – Comparing coupling metric and reuse rate

We expect to observe the strongest association with changing reuse out of all quality metrics in complexity metrics. Therefore, we measured additional complexity metrics. In this case, the use of a reuse engine causes the reduction in complexity. In Figure 3.11, we notice a reduction in all complexity metrics (WMC, McCabeCC, NBD, and % branches) with increasing reuse rate.

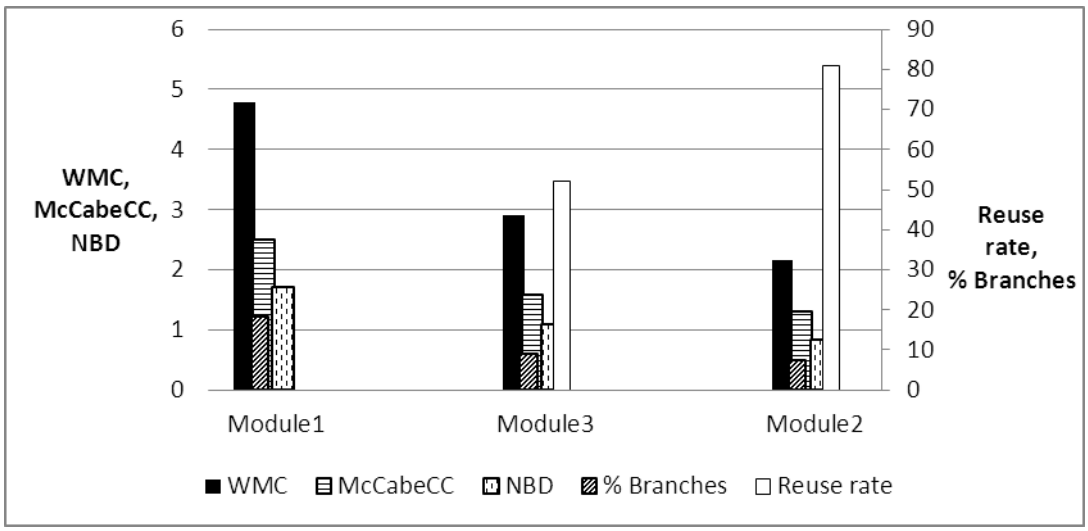

Figure 3.11 – Comparing complexity metrics and reuse rate

Figure 3.12 shows the comparison of cohesion metric (LCOM) and reuse rates. According to the measurements, there is not a valid connection between cohesion and changing reuse. It is mainly because of the reason that the non-reuse module is also a cohesive one.

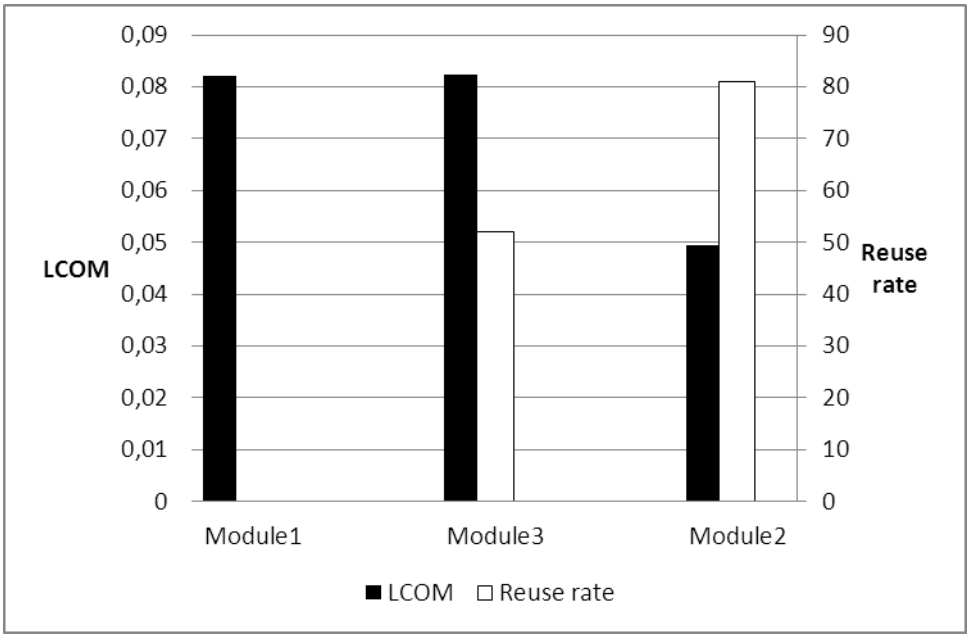

Figure 3.12 – Comparing cohesion metric and reuse rate

Figure 3.13 shows the variation of performance metric (NoCycles) with different reuse rates. As expected, the number of cycles increases with increasing reuse. In the first module, after the command is received from the system, it is sent through related socket directly; however in second and third modules the system architecture changed in order to reuse the middleware and now between sending and receiving, a middleware is introduced which increases the number of cycles.

Here, between non-reuse case and reuse cases, we expected and observed a difference in the number of cycles; but the rise of cycles in different non-zero reuse cases is not clear; which is expected, since the reuse mechanisms are the same in modules 2 and 3.

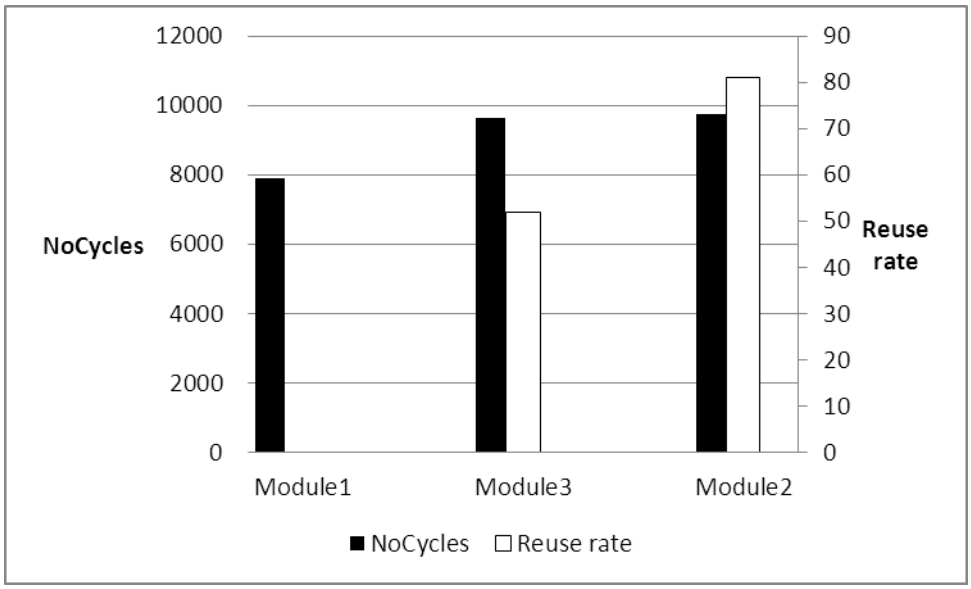

Figure 3.13 – Comparing performance metric and reuse rate

3.4 CASE STUDY 2: CHANGE OF DEFECT COUNTS AND PRODUCTIVITY BY REUSING COMPONENTS

In this part, the measurements done in case study 2 are explained. These measurements include defect counts of the components developed by TADES team, which are reused in different products and the productivity rates of these products.

Defect counts are obtained from the problem reporting system used in Aselsan SST. As a result of company politics, all defects detected during system integration and acceptance testing and also those reported by customers are kept in the problem reporting system i.e. Defects during software development process are not included in these measurements.

Measurements about requirement counts are acquired from the requirements management tool used in Aselsan SST.

Total efforts of the products are measured by the business management software used in Aselsan SST. However, due to the commercial confidentiality, we do not provide exact measures of the efforts in this study.

About the specifications of TADES SPL's various products, and about the properties of the common and product-specific components in these products, we worked with one of the configuration managers of TADES SPL.

3.4.1 MEASUREMENT OF METRICS

In this work, three different products are explored. All products have at least one product specific component, and other components are common platform components. The three products were developed sequentially with six months between the completions of each one.

We classified the components which we analyzed as "new" and "reused" components. New components are not used in earlier products, and reused components are used previously in other products. Table 3.6 and Figure 3.14 show the new and reused component counts in the products analyzed.

Table 3.6 – New and reused component counts in three different products

Product No / Component count	New	Reused
Product 1	19	0
Product 2	3	16
Product 3	6	15

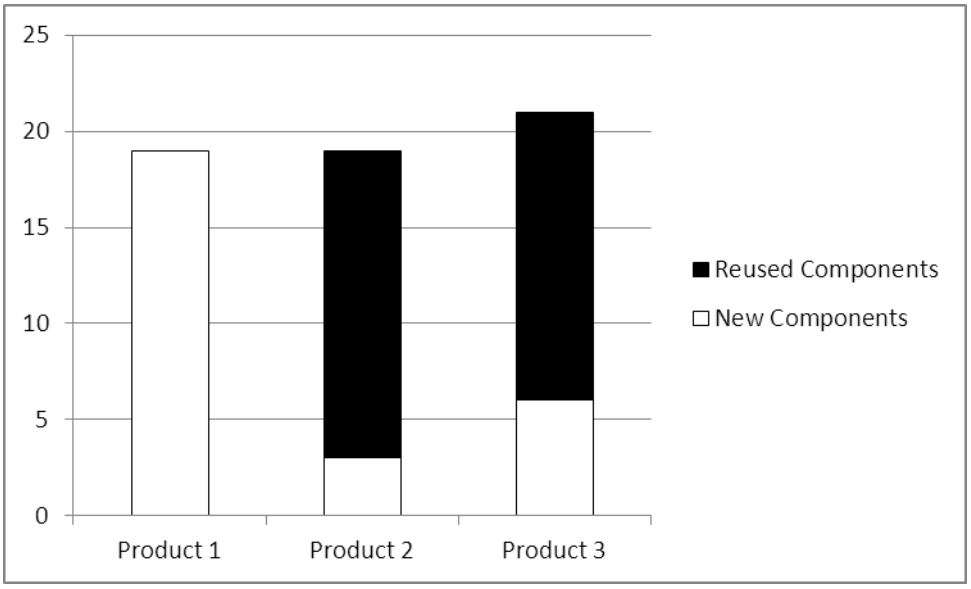

Figure 3.14 – New and reused component counts in three different products

Table 3.7 and Table 3.8 display new and total requirement counts in all components for each product, respectively. Table 3.9 displays total effort in man-hour for each product.

Table 3.7 – New requirement counts in all components for each product

Product No / New Requirements	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
Product 1	291	383	216	167	301	304	126	220	-	-	275	-
Product 2	4	10	0	0	4	11	21	32	-	-	-	141
Product 3	0	0	1	0	0	0	1	27	211	177	14	-

Table 3.8 – Total requirement counts in all components for each product

Product No / Total Requirements	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
Product 1	291	383	216	167	301	304	126	220	-	-	275	-
Product 2	295	393	216	167	305	315	147	252	-	-	-	141
Product 3	295	393	217	167	305	315	148	279	211	177	289	-

Table 3.9 – Total effort for each product

Product No	Total Effort (man-hour)
Product 1	2,75 * N
Product 2	1,5 * N
Product 3	N

Defect counts of the components used in three different products are shown in Table 3.10. Components 1-8 are common in all three products; component 11 is partly common; and components 9, 10, and 12 are new components (i.e. They are used in corresponding products for the first time). All 12 components are common-platform components.

Table 3.10 –Defect counts of the components in three different products

Product No / Component Type	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
Product 1	87	35	54	20	63	100	24	48	-	-	55	-
Product 2	2	5	1	0	0	6	0	3	-	-	-	27
Product 3	0	0	1	0	0	0	1	0	24	34	7	-

3.4.2 DISCUSSION OF THE MEASUREMENTS

3.4.2.1 DEFECT-COUNTS VS REUSE RATE

In this section, evolution of software quality in different products is analyzed when common components are reused. In this work, software quality is measured using defect counts of the components as suggested and used in many studies in the literature (see Chapter 2 – Measuring Quality section).

Table 3.11 shows reused requirement percentages (calculated as shown in formula 3.1) in all components for each product. According to this table, in product 2 the average of the reused requirements rate for common components (i.e. Components 1-8) is 95.5 %, and the same rate in product 3 is 94.3 %.

$$\%\ Reused\ Requirements = \frac{Number\ of\ reused\ requirements}{Number\ of\ total\ requirements} x100 \qquad (3.1)$$

Table 3.11 – Reused requirement percentages in all components for each product

Product No / % Reused Requirements	C1	C2	C3	C4	C5	C6	C7	C8	C 9	C 10	C1 1	C 12
Product 1	0	0	0	0	0	0	0	0	-	-	0	-
Product 2	98,64	97,46	100	100	98,69	96,51	85,71	87,3	-	-	-	0
Product 3	98,64	97,46	99,54	100	98,69	96,51	85,14	78,85	0	0	95,2	-

Distribution of defect counts of the common components in all three products is shown in Figure 3.15, Figure 3.16, Figure 3.17, and Figure 3.18:

- Figure 3.15 shows defect counts of the common components for each product,
- Figure 3.16 shows the average of the defect counts of the common components for each product (average was calculated by adding all defect counts up and dividing by eight for each product),
- Figure 3.17 shows defect percentages of the common components for each product,
- Figure 3.18 shows average defect percentages of the common components for each product (average was calculated simply by adding all defect percentages up and dividing by eight for each product).

According to these figures, the average defect amount in the first product is more than 50, less than 3 in the following product and less than 1 in the third product. More than 95 % of the total defects are detected in the first product. There are various reasons for this improvement: the reused components are less modified than non-reused ones; therefore, they are more stable than those. Additionally, the reused components are designed more intensely; since defects in them affect different products. Furthermore, the employment of the common components in various products causes them to become faultless, and finished components.

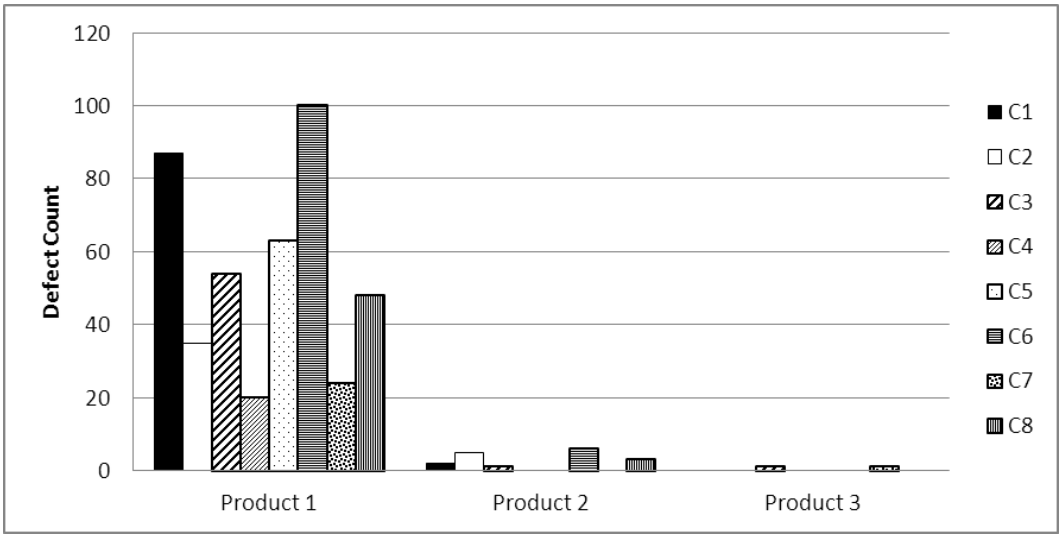

Figure 3.15 – Defect counts of the common components (C1-C8)

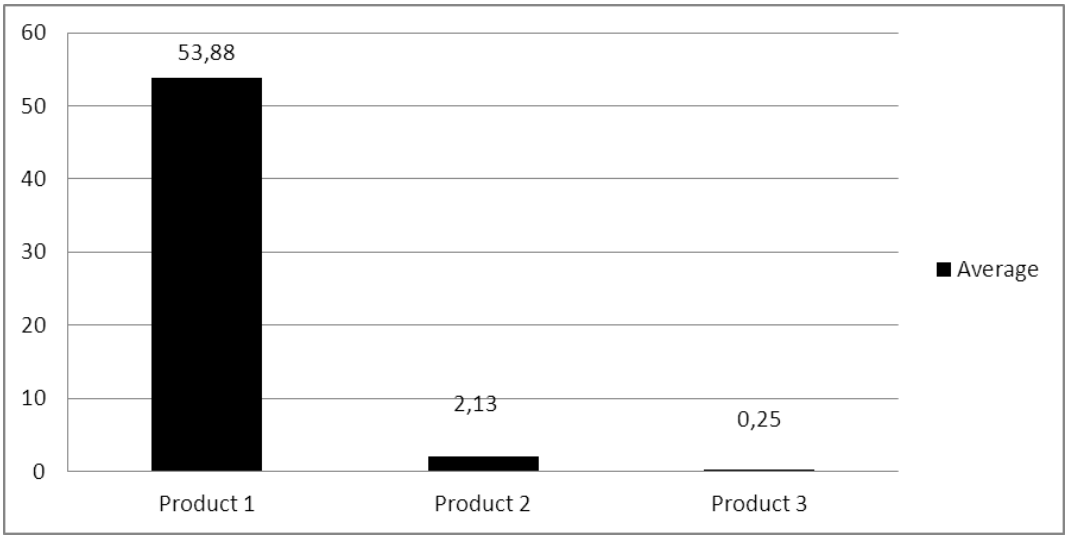

Figure 3.16 – Average defect counts of the common components (C1-C8)

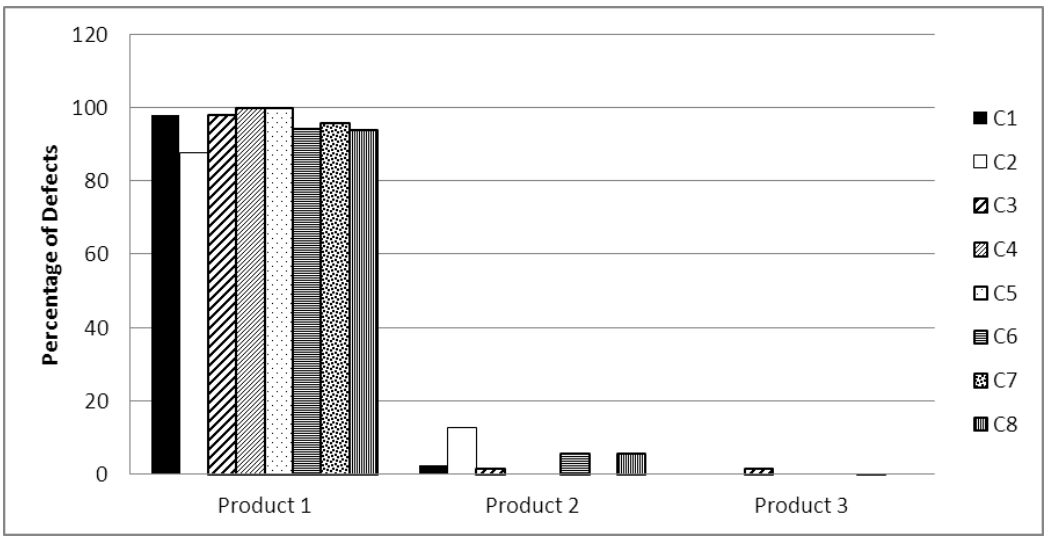

Figure 3.17 – Defect percentages of the common components (C1-C8)

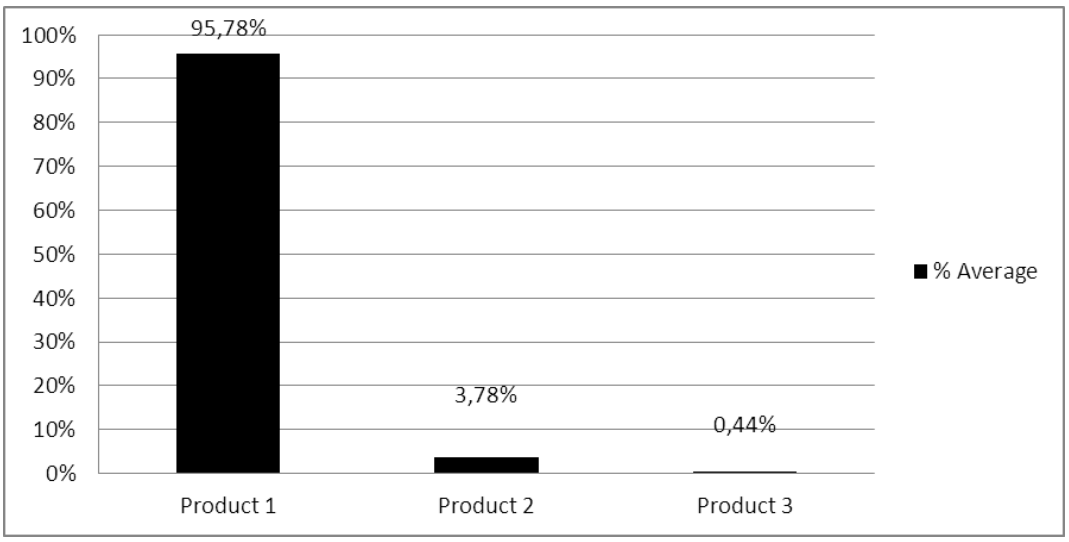

Figure 3.18 – Average defect percentages of the common components (C1-C8)

In Figure 3.19, defect counts of the product-specific components (i.e. Components 9, 10, and 12), and the average defect counts of common components are shown. For all three components, defect counts are more than 20. Since, these components are not reused; we observe a similar distribution as the defect counts of the common components in the first product they are used.

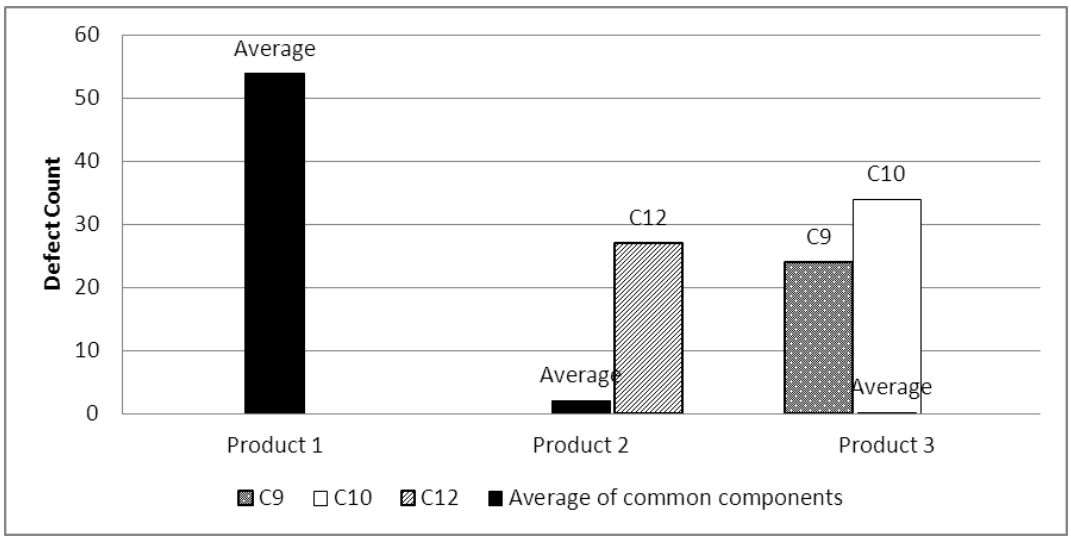

Figure 3.19 – Defect counts of the product-specific components (C9, C10, and C12)

Defect amount of component 11 is shown in Figure 3.20. This component is partially-common in products 1 and 3 (see Table 3.7 and Table 3.8). In product 1, more than 50 defects are detected, and in product 3 almost 10 defects are detected. The defect distribution of this component is similar to common components' distribution of products 1 and 2.

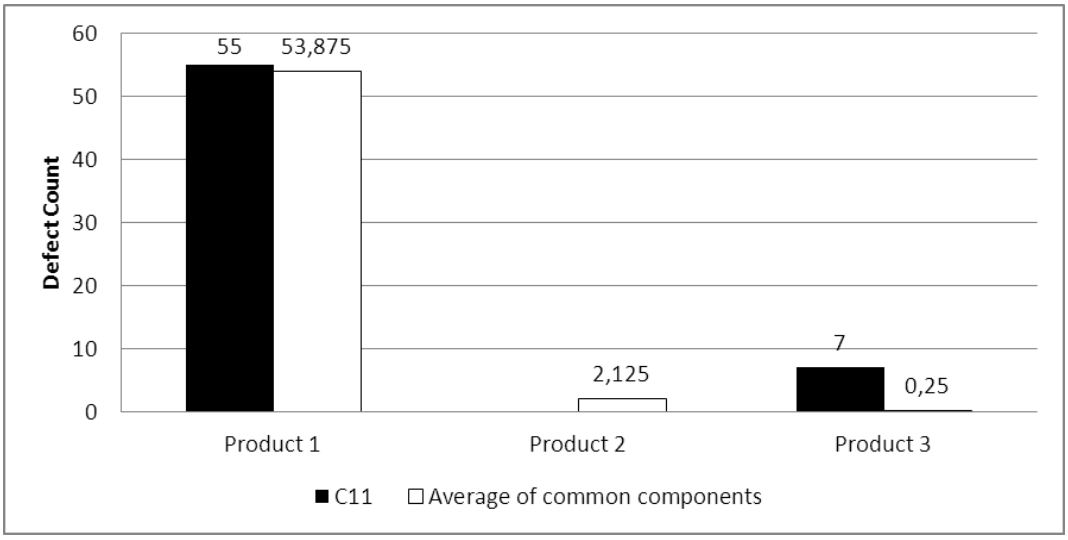

Figure 3.20 – Defect counts of the partially-common component (C11)

3.4.2.2 PRODUCTIVITY RATES IN DIFFERENT PRODUCTS

In this sub-section, we will compare productivity rates of the three products. The productivity will be calculated as shown in formula 3.2.

$$Productivity = \frac{Number\ of\ requirements}{Total\ efforts\ in\ man-hours} \tag{3.2}$$

First, productivity rates are calculated using the number of new requirements (Table 3.12).

Table 3.12 – Productivity rates using new requirements

Product No	New Requirements	Productivity (requirements / man-hour*N)
Product 1	2283	830,2
Product 2	223	148,7
Product 3	431	431

Then, productivity rates using the total number of requirements are calculated (Table 3.13).

Table 3.13 – Productivity rates using total requirements

Product No	Total Requirements	Productivity (requirements / man-hour*N)
Product 1	2283	830,2
Product 2	2231	1487,3
Product 3	2796	2796

In Figure 3.21, the comparisons of the productivity rates are displayed. The comparison of the productivity rates with total requirements indicates that as the components are reused in different products, the productivity rates increase remarkably, which is not surprising. The second comparison, the comparison of the productivity rates with new requirements exhibits a sharp reduction between the first and second products; and a noteworthy expansion between the second

and third products. We interpret the first case as a conclusion of the development by employing a product-line. Since the reused components are already developed in previous products, the productivity rates increase significantly. Furthermore, the first reduction in the second case is explained as an adaptation period of the development with the product line. Although there are developed-products, ready to be used, it is still time-consuming to gather up these components and integrate them with the recently developed components. Finally, the expansion between the second and third products in the second case is resolved as an evidence of being trained in development with the product line. It is seriously expected that, in the coming products, the productivity rates using the new requirements will exceed the productivity rate of the first product.

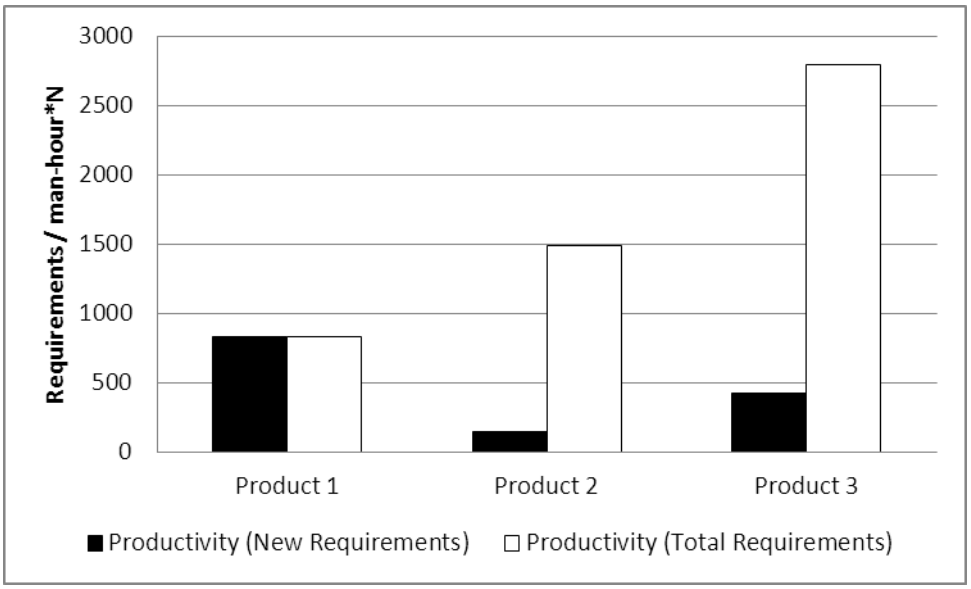

Figure 3.21 – Comparison of productivity rates of each product using new requirements and total requirements

3.5 CASE STUDY 3: CHANGE OF PRODUCTIVITY AND PERFORMANCE WITH INCREASING REUSE RATES IN SSRM SPL

In this section, we will show changing productivity rates as reuse rates vary for different products developed sequentially using SSRM SPL. In addition, the performance measurements of a critical scenario in this domain will be measured and compared before and after the SSRM SPL is employed.

3.5.1 MEASUREMENT OF PRODUCTIVITY METRICS

Reuse rates are calculated using reused non-comment line of codes and total non-comment line of codes. Productivity rates are calculated using total non-comment line of codes and total effort to develop the so called product (see formula 3.3). Total efforts are measured by the business management software used in Aselsan SST. However, due to commercial confidentiality, we do not provide total efforts and total source line of code metrics in this study. For making these measurements, we worked with one of the configuration managers of the SSRM SPL team.

$$Productivity = \frac{SLOC}{Total\ \ efforts\ \ in\ \ man-hours} \tag{3.3}$$

In Table 3.14, reuse and productivity rates for the products developed using SSRM SPL are given.

Table 3.14 – Reuse and productivity rates for products in SSRM SPL

Product No	% Reuse Rate	Productivity (SLOC / man-hour)
Product 1	35,73	54,22
Product 2	39,25	54,92
Product 3	48,86	43,38
Product 4	48,9	68,39

3.5.2 DISCUSSION OF THE PRODUCTIVITY MEASUREMENTS

In Figure 3.22, the comparison of reuse and productivity rates of the products is displayed. Reuse rates increase from product 1 to product 4; however productivity change does not have the equivalent attitude. Productivity rates increase slightly between the first two products, and then productivity rate decreases from product 2 to product 3. However, between the last two products, productivity rate extends noticeably. When we discussed this situation with the SSRM SPL team, we found out the following factors:

- There was a serious waste of time during the development of the non-reused (new) parts in product 3,
- Most of the developers of the product 3 were unfamiliar with software development by employing the product-line.

We can conclude that utilization of some normalizing factors i.e. Code complexity for the non-reused parts and experience of the developers, in measuring productivity rates can be useful. Furthermore, during the initial products, it is not surprising to observe productivity decays; since it is time consuming to get used to the product line in a software development team.

3.5.3 MEASUREMENT OF PERFORMANCE METRICS

In Case Study 3, in order to compare the performance before and after employing the product line, one of the most critical scenarios in the system, the automatic video tracking scenario is investigated. The performance metrics used are time delay and CPU usage in this scenario. These metrics are measured using the provided embedded operating system functions.

In this scenario, in order to achieve the critical mission, three different systems work together as displayed in Figure 3.23.

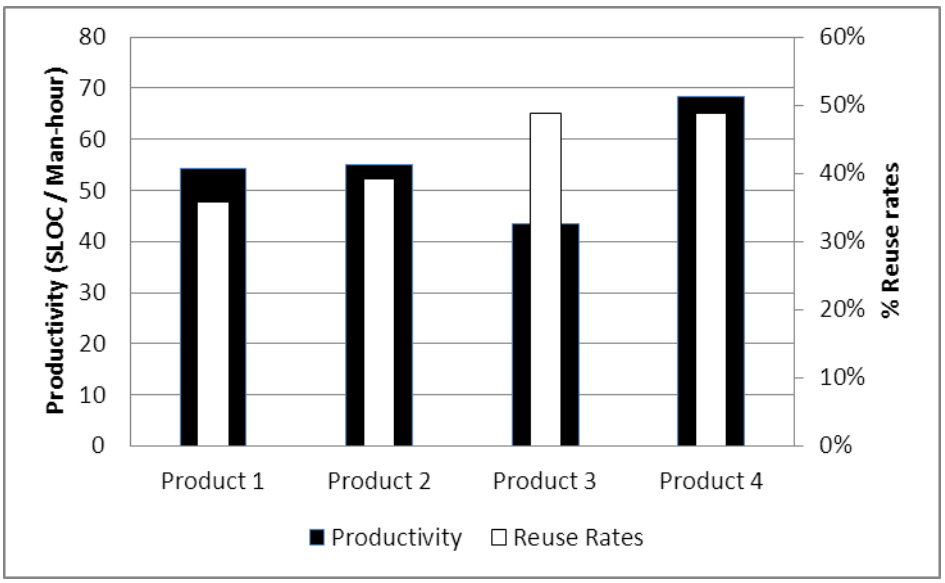

Figure 3.22 – Comparison of Reuse and Productivity for products in SSRM SPL

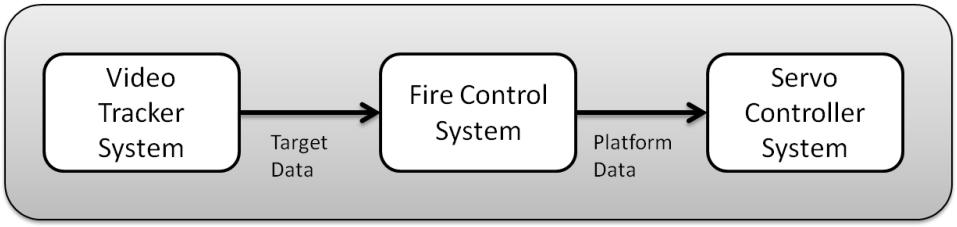

Figure 3.23 – Automatic target tracking scenario

Video Tracker (VT) system detects, and tracks the selected targets. The target data is sent from VT to Fire Control System periodically; then this system adds platform offsets into target data and generates the platform data. Finally, this platform data is sent to the Servo Controller System and then the Servo Controller System drives the servo motors using the platform data. This scenario is called the Automatic Video Tracking (AVT) scenario.

Before employing the SSRM SPL, the classes implementing the AVT scenario in Fire Control System are displayed in Figure 3.24.

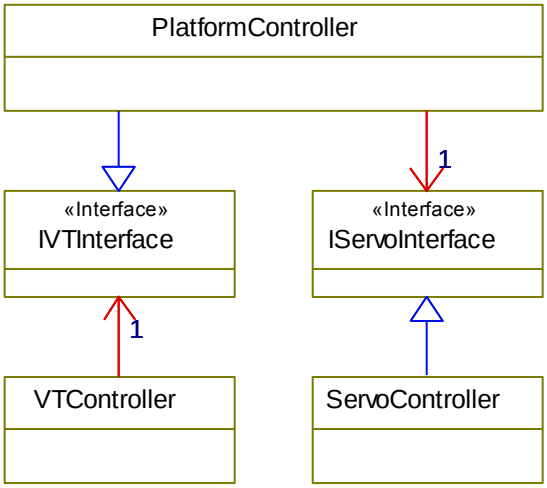

Figure 3.24 – Class diagram of Fire Control System AVT scenario before SSRM SPL

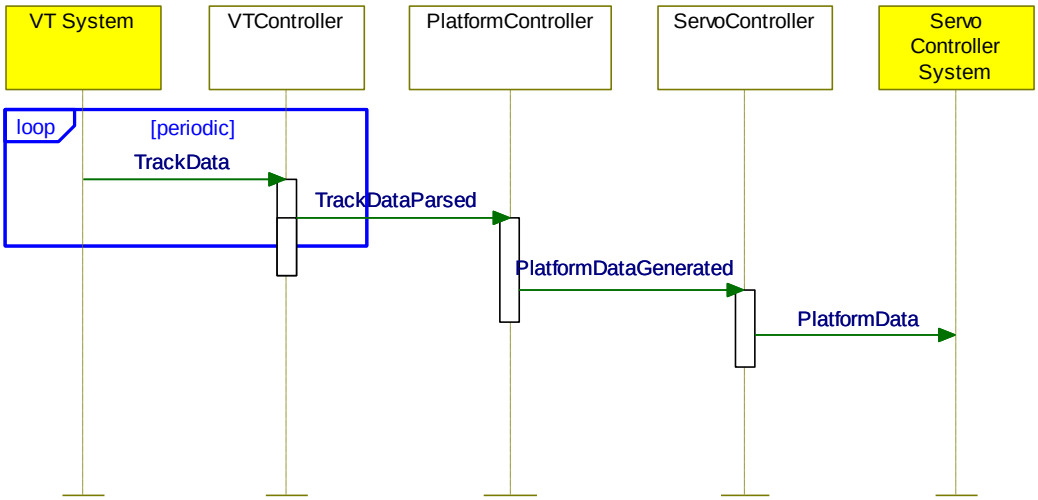

Figure 3.25 – Sequence diagram of Fire Control System AVT scenario before SSRM SPL

The AVT scenario sequence diagram before employing the SSRM SPL is shown above in Figure 3.25.

The delay from reception of the track data from VT System to the transmission of the platform data to the Servo Controller System and the CPU usage during the scenario are measured. Measurements are shown below in Table 3.15.

Table 3.15 – Measurements of the AVT scenario before SSRM SPL

Minimum Delay (ms)	**Maximum Delay (ms)**	**Average Delay (ms)**	**% CPU Usage**
0,85	1,05	0,9	72,3

During the development process of the product line, the team transformed the AVT scenario into a mission [49]. In order to isolate the VTController and ServoController classes from the AVTMissionController, they introduced two more layers into the scenario as shown below in Figure 3.26. The team introduced the TargetManager and ServoManager classes in order to support other target sources and platform controllers.

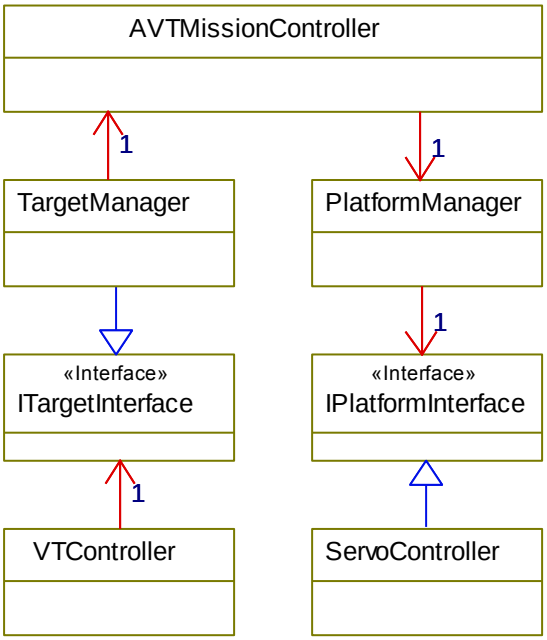

Figure 3.26 – Class diagram of the AVT mission

Since VT System provides target data periodically, there were two options in TargetManager: Either it would receive periodic updates from VTController (push strategy), or it would get target data from VTController periodically (pull strategy). In order to keep the abstraction level high, the team decided on "pull strategy". The sequence diagram of the AVT mission implemented with pull strategy is displayed in Figure 3.27.

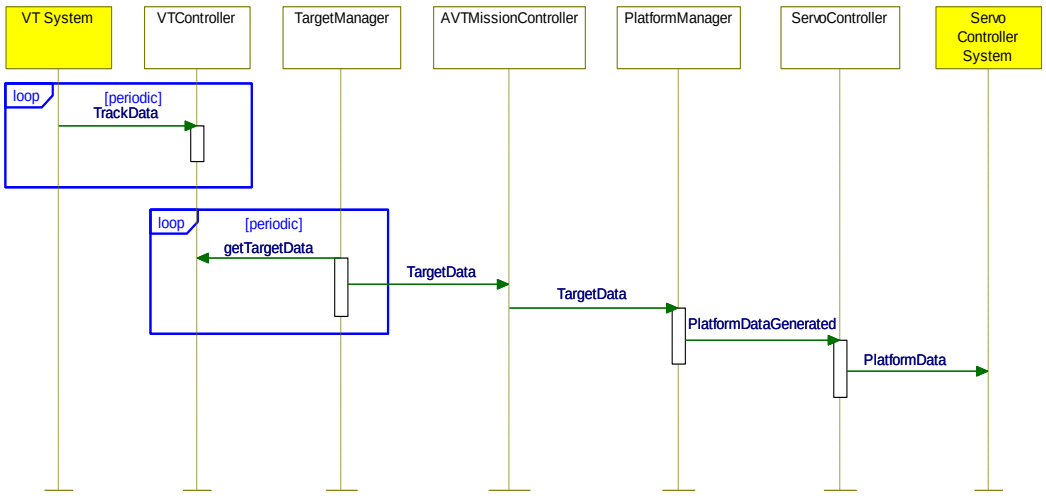

Figure 3.27 – Sequence diagram of the AVT mission with pull strategy

The delay from reception of the track data from VT System to the transmission of the platform data to the Servo Controller System and the CPU usage during the scenario are measured. Measurements are shown below in Table 3.16.

Table 3.16 – Measurements of the AVT mission with pull strategy

Minimum Delay (ms)	Maximum Delay (ms)	Average Delay (ms)	% CPU Usage
2,12	35,0	20,0	81,5

The average delay due to pull strategy was 20 ms, which is sufficient for some systems; however for systems which have high performance requirements, this delay was unacceptable. Therefore, the team implemented the "push strategy" for taking updates from VTController.

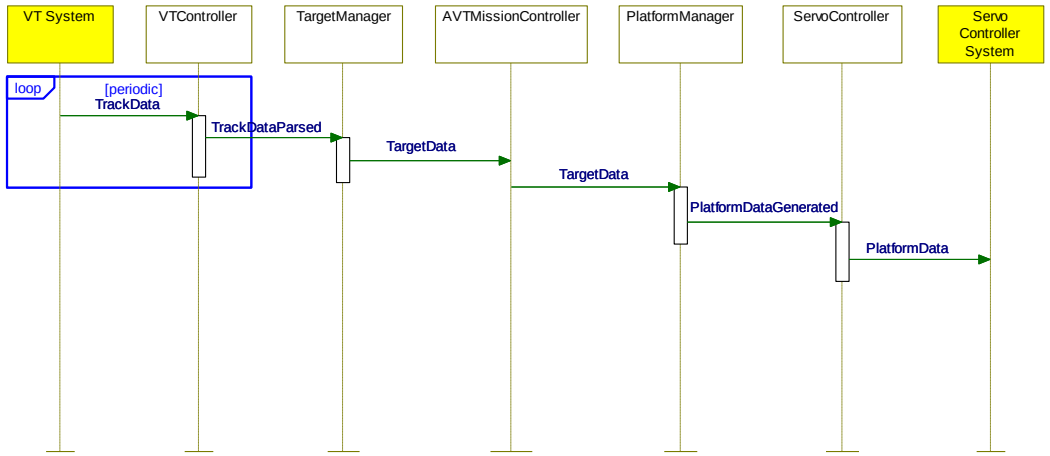

Figure 3.28 – Sequence diagram of the AVT mission with push strategy

The sequence diagram of the AVT mission implemented with pull strategy is shown in Figure 3.28.

The delay from reception of the track data from VT System to the transmission of the platform data to the Servo Controller System and the CPU usage during the scenario are measured. Measurements are shown below in Table 3.17.

Table 3.17 – Measurements of the AVT mission with push strategy

Minimum Delay (ms)	Maximum Delay (ms)	Average Delay (ms)	% CPU Usage
2,12	5,5	3,2	79,9

3.5.4 DISCUSSION OF THE PERFORMANCE MEASUREMENTS

Above, we explained three different implementation methods of the AVT scenario. The first method was before the team developed the SSRM SPL. In the second and third methods, there are

two additional layers which are due to the product line employment and in order to increase the reuse of the scenario software.

The AVT scenario delay is measured for three cases (see Table 3.15, Table 3.16, and Table 3.17), and comparison of the average delay and CPU usage are shown in Figure 3.29. According to these comparisons, we can conclude that while transforming the software into more reusable, and more abstract from the interfaces; we lose from the performance. Therefore, the developers should decide on the limit of this trade-off. Sometimes, the performance requirements allow these improvements; however sometimes performance requirements are too heavy. The second method was the most appropriate one in terms of reusability, however the developers had to perform and apply the third method. It was also more reusable compared to the first method, and it was acceptable when the performance requirements were considered.

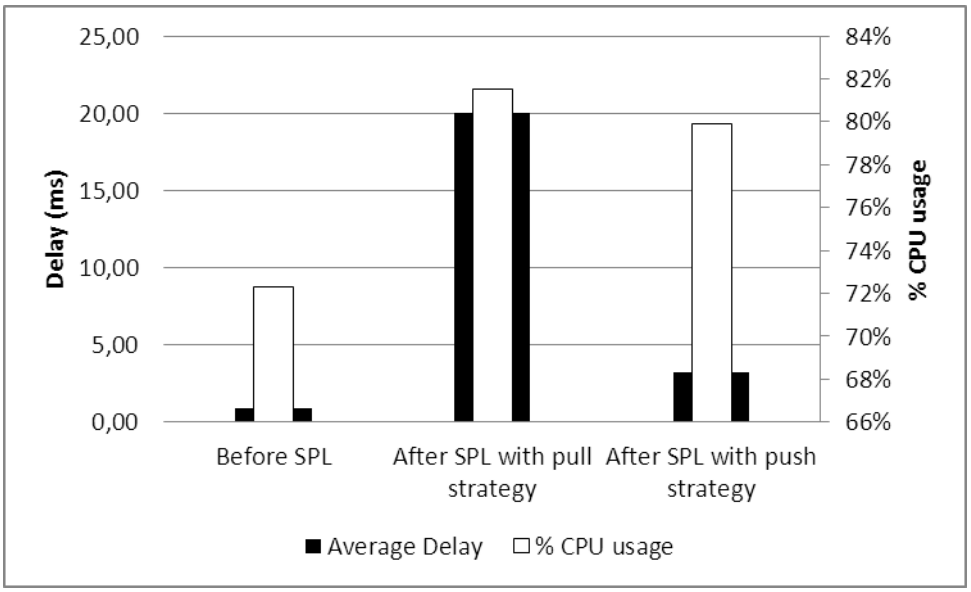

Figure 3.29 – Comparison of average delays and CPU usages of three different implementations of AVT scenario

3.6 VERIFICATION OF THE HYPOTHESES

In this section, the results of the measurements in the case studies will be analyzed, and it will be stated whether or not the hypotheses are verified.

3.6.1 HYPOTHESIS 1 (CASE STUDY 1)

Code-based Quality: The quality of software products is improved as reuse rates of the products increase.

The motivation in formulating Hypothesis 1 was the fact that we expected to observe a correlation between changing reuse rates and OO metrics. In the literature, it was argued that OO metrics are strongly related to predicting fault-proneness of software products, and similarly in the literature, the most suggested measure of software quality is the number of defect rates in a software product.

Therefore, in order to say that Hypothesis 1 is true, we have to display a correlation between changing reuse rates and OO metrics.

In this work, an experiment was designed in order to compare the software quality metrics with changing reuse rates. We have found a strong correlation between complexity metrics and reuse rate. Between coupling metrics and reuse rate, we have observed a positive relationship. With other quality metrics, we did not observe a strong relation. We observed that size metrics and inheritance metrics have similar attitudes. In Table 3.18, the results of the experiment are shown: "-" means the so called metric is not correlated to changing reuse rate, and "+" means there is a correlation observed between the so called metric and changing reuse rate.

Now, we can answer if Hypothesis 1 is true or false. The results show that some CK metrics and size metrics do not correlate with changing reuse rate: SLOC, DIT, NOC, and LCOM. However, the results also show that Coupling and Complexity CK metrics and the additional complexity metrics show a strong correlation with the changing reuse rate (see Figure 3.30).

Table 3.18 – Summary of the results of the OO metrics of the experiment

Metric acronym	Primary OO Concept	Results
SLOC	Class	-
DIT	Inheritance	-
NOC	Inheritance	-
CBO	Coupling	+
WMC	Complexity	+
McCabeCC	Complexity	+
NBD	Complexity	+
% Branches	Complexity	+
LCOM	Cohesion	-

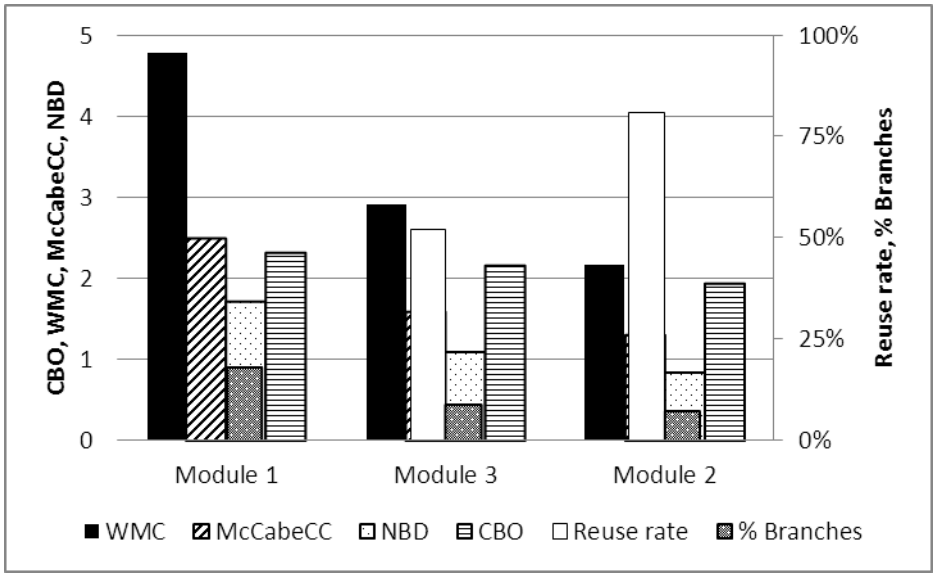

Figure 3.30 – Reduction of complexity and coupling metrics with respect to increasing reuse rate

As analyzed above, the complexity and coupling metrics are correlated to reuse rates which is consistent with the "Theoretical and empirical analysis of the object-oriented metrics" section of Chapter 2, where it was empirically displayed that the most related metrics with fault-proneness are the coupling and complexity metrics.

Consequently, the improvements in coupling and complexity metrics are appropriate to claim an increase in software quality. Therefore, we can conclude that Hypothesis 1 is verified.

3.6.2 HYPOTHESIS 2 (CASE STUDY 1, CASE STUDY 3)

Performance of Embedded Software: Performance of the embedded software products decays as reuse rate of the products increase.

In the experiment designed in case study 1, we measured and compared the performance of a message receiving and transmitting scenario in three different embedded software modules. We find a strong negative correlation between performance and reuse rate; which is consistent with the related arguments in the literature.

In case study 3, we measured and compared the performance metrics of a critical scenario of an embedded software system before and after employing a product-line approach. We observed that, the case before the product line approach was the best regarding the performance, and as the software turned into more reusable and more abstract from other parts of the software, the performance of the software decayed.

Consequently, the measurements from two different case studies display that Hypothesis 2 is verified (see Figure 3.31).

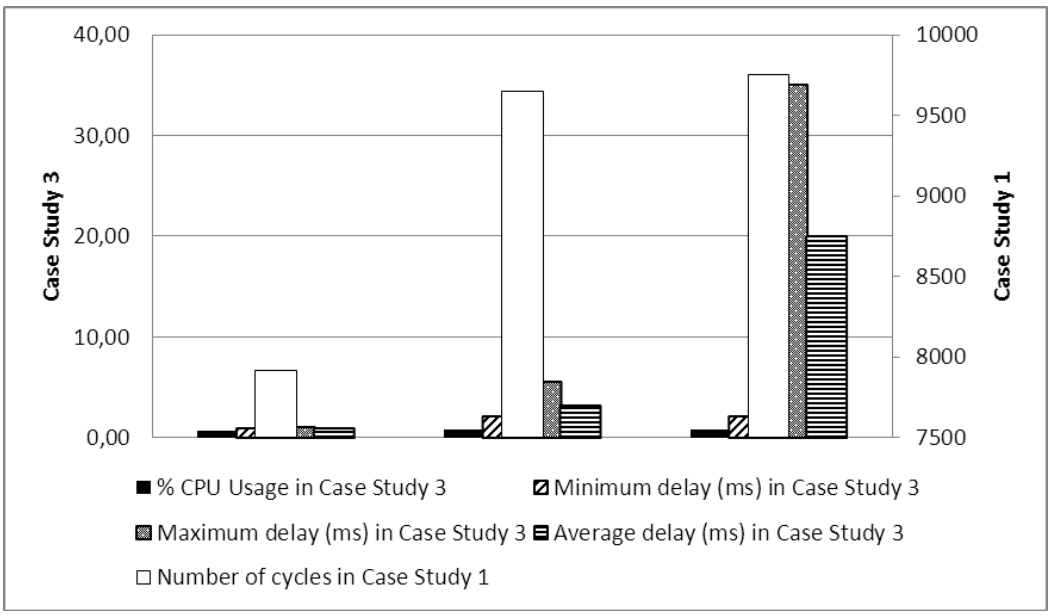

Figure 3.31 – Worsening of performance metrics as reusability increases in case studies 1 and 3

3.6.3 HYPOTHESIS 3 (CASE STUDY 2)

Fault-proneness: The number of defects detected in components decreases as these components are reused in various products.

In case study 2, measurements are taken from a product line which is used in subsequent products.

When the components are not reused, namely not used in earlier products, we observe a large number of defects in these components. Furthermore, we find that the decrease in the defect counts is independent of the product types. When the component is firstly used in product 3, again we detect a similar distribution as if the component is firstly used in product 2.

To conclude, as components are reused in several products, we observed that their defect counts decrease significantly and so their fault-proneness. Therefore, we can conclude that hypothesis 3 has been verified in this study.

3.6.4 HYPOTHESIS 4 (CASE STUDY 2, CASE STUDY 3)

Productivity: The productivity rates of products increase, as the reuse rates of these products increase.

In case study 2, productivity rates of the three products developed using the SPL approach are presented. Productivity rates are measured using the number of requirements using the requirements count and total effort. The results showed that, if the productivity is measured using the total number of requirements in the deployed product, the productivity rates improve significantly. Additionally, productivity is measured also by using the new requirements. In that case, we observed a reduction in productivity between the first and second products; and an increase in productivity between the second and third products. This situation is interpreted as an adaptation period of the product line approach.

In case study 3, we compared productivity rates of products implemented by another product line approach with increasing reuse rates. In this measurement, we also observed a positive correlation with reuse and productivity rates. However, the change of the members of the team during the development of product 3 caused a reduction in the productivity rate of this product. This situation is interpreted, similarly in case study 2, as an adaptation period of the product line.

Hence, we concluded that, if the effects of the adaptation period of the product line approach are ignored, the productivity rates improve significantly as the rate of reuse increases in a product line. Therefore, hypothesis 4 has been verified.

3.7 SUGGESTIONS TO FURTHER BENEFIT FROM THIS STUDY

In this section, discussions made in the measurements collection and hypotheses verification sections will be reviewed. By means of those analyses, suggestions regarding the reuse infrastructure and process, to improve benefits of reuse in Aselsan will be formulated.

3.7.1 USE OF REFERENCE METRICS

Measurements taken from a recently developed middleware technology used in Aselsan, were used for verification of Hypothesis 1. In these measurements, the evolution of code-based metrics with respect to changing reuse rates is examined, and improvements in both coupling and complexity metrics are observed. This study shows the importance of application of code-based metrics in software development since the goals of the so called middleware technology are remarkably verified by these measurements. Hence, software developers should incorporate these metrics into their software development processes, and before and after serious decisions on design, technology or infrastructure; the change of these metrics should be investigated.

Therefore, in order to succeed in the employment of these metrics, the software developers should select reference metrics specific to their software domain and periodically monitor the changes of these metrics.

3.7.2 AUTOMATED DETECTION OF ARCHITECTURAL EFFECTS

In order to verify Hypothesis 2, the changes of performance metrics of two different embedded systems were examined before and after introducing a reuse infrastructure in the software development process. In the first system, the above-mentioned middleware was introduced, and performance measurements are taken. Furthermore, in the lesser case, the performance of a critical scenario of an embedded software was measured after introducing the software product line. These measurements showed the negative impacts of object-oriented programming and OO concepts, therefore the reuse infrastructure, in the performance of an embedded software. However, it is well known that performance is critical for real time embedded systems. Hence, if the performance of the embedded software turns out to be below the system limitations as a result of the architectural improvements, then the developers should modify the architecture. In the second case, we observed the above-described situation. The performance of the software was insufficient for the system, after the employment of the architectural improvements due to the product line approach, therefore; the team had to update the architecture and violate some OO concepts. Henceforth, the real time embedded software developers should monitor the performance requirements after employing extensive architectural modifications; furthermore, they must update the modifications if the performance of the software eventually becomes unacceptable for the system.

Thereupon, the embedded software developers should develop methods in order to automate the process of detecting the architectural modifications which include the chance of worsening the software performance below system requirements.

3.7.3 RECORDING SOFTWARE DEVELOPMENT PROCESS DEFECTS

The defect counts of the components in consecutive products, developed by a product line approach, were measured in order to verify Hypothesis 3. Post-release defects were collected from the problem reporting system used in Aselsan; since the defects detected during the software development process are not stored in this system. This study was sufficient to show that as the components are reused in different products, defect counts of these components decay noticeably. However, we were incapable of analyzing the defects detected during the software development process since these defects are not recorded anywhere. Additionally, the severities of the defects were not indicated after they are corrected. Therefore, while analyzing the improvements of the components, it was impossible to assess their severity.

Hence, in order to improve the management of defects, and investigate the defects intensely; the severity of the defects should also be provided after being corrected. Additionally, the defects detected during the software development process should also be recorded; since the defects of the components during the development process is a key metric in order to improve the reuse infrastructure of a product line.

3.7.4 ASSOCIATION OF DEFECTS WITH DESIGN CONCEPTS

In the problem reporting system used in Aselsan, the defects were associated with the components, but they were not associated with the corresponding design concepts such as class, and method. Hence, the reasons of the defects were not thoroughly analyzed. Henceforth, in case study 3, while measuring the productivity achieved in developing the products, we were incapable of measuring the defect counts of the components reused in these products; since the defects were recorded in the problem reporting system associated with the products, but not components.

Consequently, in order to improve the management of defects, and investigate the defects intensely; the software developers should identify each defect with corresponding component, and the design concept.

3.7.5 RECORDING REWORK EFFORTS AND EFFORTS ASSOCIATED WITH REUSE

Productivity rates were measured and compared in order to verify Hypothesis 4. The total efforts of the products were measured by the business management software used in Aselsan. In case studies 2 and 3, the efforts of the products, developed by a product line method were collected. However, there was no infrastructure in order to record the rework efforts, efforts associated with reuse, post-release efforts, or maintenance efforts.

Therefore, in order to be able to measure and analyze these metrics, with changing reuse rates; these metrics should be recorded, and for this purpose the relevant infrasture should be developed.

3.7.6 EXPLICIT ACCOUNTING FOR CODE REUSE

During analysis of the productivity rates of the products, it was found that lack of the experience of the developer team was a significant factor of the declines in productivity; since, the employment of product lines requires an extra effort such as an adaptation period. Henceforth, during effort estimations of the products developed by a product line approach; the experiences of the developers, about the product line, should also be considered. Finally, the developers should also estimate and record efforts separately for reused and non-reused components, in order to analyze the impacts of reuse on the productivity rates deeply.

CHAPTER 4

DISCUSSION AND CONCLUSION

Prior to this study, several studies have been conducted on software reuse and software quality ranging from theoretical to experimental studies. These studies have shown strong evidences in the relationship of software reuse and software quality; although, in these studies, the measurement methods of the software reuse and software quality had differences. In some studies, source line of code is used to calculate reuse; however in others the number of requirements is used. For measuring software quality, mostly fault-proneness of components is used; but also when the study is not an empirical one, the researchers used specific code-based metrics to calculate and compare the quality of different software products.

In this study, we did both theoretical and empirical studies. We worked with Turkey's leading defense industry company: Aselsan's software engineering department. We aimed to examine their software projects and follow reuse and quality relations for these projects, and formulate suggestions in order to improve their reuse infrastructure and process. For this purpose, we carried out three separate case studies to investigate reuse and quality relations.

In the first case, we compared the code-based quality metrics of three software modules while two of these modules are produced by a reuse engine. In this study, we observed that complexity and coupling metrics improved, however the performance decayed as reuse rate increased.

In second and third cases, we investigated two different product lines which are actively used in many projects in Aselsan.

We were able to compare post-release defect counts of the components reused in consequent products i.e. Fault-proneness of the components as they are reused in various products, in the first product line. Furthermore, the productivity of the discussed products is also discussed, as the common components reused in these consecutive products. In this study, we observed that fault-proneness of the components decreased and the productivity rates extended as the components are reused in other products.

In the second product line, we compared productivity rates of several products as reuse rates of these products increased; and observed that as reuse rates of the products increased, the productivity rates also improved. Moreover, the worsening of the performance metrics in this product line was also observed as the reusability increased, similar to the first case.

As we expressed above, we accomplished three different case studies and measured and compared different concepts in all three cases i.e. OO quality metrics, productivity rate, defect counts, performance. We were not able to obtain all these measures in all cases. For instance, in the first study, the team was still developing the modules which we measured OO metrics and compared. Therefore, the modules were not tested, and there were no defect rates which we could collect. In the second case, we were able to obtain defect counts for each single component because the team designed their product line in such a way that each component was being tested separately and; therefore, the defect counts were specific to the components. However, the second product line was not designed in this way; therefore it was almost impossible to associate a defect and a component. Defects were entered into the problem reporting tool for each product but not for each component.

Therefore, in the third case, we could not collect defect counts of the components and use in the discussions. On the other hand, we were unable to obtain code-based metrics for the cases 2 and 3; for this reason, we used code metrics only in case 1. In short, we designed the case studies this way because we were able to obtain only these measurements.

In future studies, it would be a significant improvement if all types of measurements could be collected i.e. Code-based metrics, defect counts (separate for each component) and productivity; and if these metrics could be compared with changing reuse rates.

In this study, we were able to collect defect counts after the software development process i.e. post-release; because defects found after testing and reported by customers are kept in problem reporting system. However, it would be an improvement to consider defect counts found during the development process together with the code-based metrics. Whereas, defects found by developers are immediately solved and not recorded anywhere, therefore it seems to be practically difficult to make this improvement.

Additionally, it would be better to consider not only the number of defects, but also their severity. Although there is an additional field for problem severity in the problem reporting system used in Aselsan, we were able to reach only the defect counts. Furthermore, in our measurements, we observed that, in the problem reporting system, the defects were associated with the components in the best case. However, in order to make deeper analyses of the defects, they should also be associated with the design concepts.

In prior studies, it was shown that complexity and coupling metrics can successfully predict post-release defects. Maybe in future studies, we can explain exactly which metrics are appropriate for projects of Aselsan, by matching code-based metrics and post-release defects. Moreover, building an infrastructure in order to periodically monitor these metrics would be a serious improvement in the software development process.

Furthermore, measuring and recording rework efforts, efforts associated with reuse, post-release efforts, and maintenance efforts, in addition to development efforts, will also be a significant enhancement in the software development process, since analyses of these measures with changing reuse rates will be achievable.

Consequently, in this thesis, we have compared some software measurements such as OO quality metrics, fault-proneness, performance, and productivity with changing reuse rates in a leading defense industry company of Turkey. Up to the present time, similar case studies have been conducted elsewhere including mostly telecommunication companies; however in Turkey, there are not many case studies especially done in the defense industry. Because of that, this study is expected to constitute a promising start.

REFERENCES

[1] William Frakes and Carol Terry, "Software reuse: metrics and models," ACM Computing Surveys, vol. 28, no. 2, pp. 415-435, June 1996.

[2] W.C. Lim, "Effects of reuse on quality, productivity, and economics," IEEE Software, vol. 11, no. 5, pp. 23 - 30, September 1994.

[3] M. Morisio, D. Romano, and I. Stamelos, "Quality, productivity, and learning in framework-based development: an exploratory case study ," IEEE Transactions on Software Engineering, vol. 28, no. 9, pp. 876 - 888 , September 2002.

[4] S. Sedigh-Ali, A. Ghafoor, and R.A. Paul, "Metrics and models for cost and quality of component-based software ," in Proceedings of IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003, pp. 149 - 155.

[5] ISO/IEC, "Information technology - Software product quality - Part 1: Quality model," ISO, ISO/IEC FDIS 9126-1:2000 (E), 2000.

[6] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz, "An Empirical Study of Software Reuse vs. Defect-Density and Stability," in Proceedings of International Conference on Software Engineering, 2004, pp. 282 - 291.

[7] J.C.C.P. Mascena, E.S. de Almeida, and S.R. de Lemos Meira, "A comparative study on software reuse metrics and economic models from a traceability perspective," in Proceedings of IEEE International Conference on Information Reuse and Integration, 2005, pp. 72 - 77.

[8] Khaled El-Emam, "Object-oriented metrics: A review of theory and practice," in Advances in software engineering. New York, NY, USA: Springer-Verlag New York, Inc, 2002, pp. 23-50.

[9] S.R. Chidamber and C.F. Kemerer, "A metrics suite for object oriented design ," IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493 , June 1994.

[10] Walcelio L. Melo, Lionel Briand, and Victor R. Basili, "Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems," UM Computer Science Department, Technical report CS-TR-3395, 1998.

[11] Parastoo Mohagheghi and Reidar Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," Empirical Software Engineering, vol. 12, no. 5, pp. 471-516, 2007.

[12] Hafedh Mili, Fatma Mili, and Ali Mili, "Reusing Software: Issues and Research Directions ," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, vol. 21, no. 6, pp. 528-562, June 1995.

[13] Liesbeth Dusink and Jan van Katwijk, "Reuse Dimensions," in SSR '95 Proceedings of the 1995 Symposium on Software reusability , 1995, pp. 137-149.

[14] J. Sametinger, Software Engineering with Reusable Components.: Springer-Verlag, 1997.

[15] W. B. Frakes and G. Succi, "An industrial study of reuse, quality, and productivity," Journal of Systems and Software, vol. 57, no. 2, pp. 99 - 106, June 2001.

[16] ISO/IEC, "ISO/IEC 9126-2 : Information Technology – Software product quality–Part 2 : External Metrics," ISO, ISO/IEC TR 9126-2 , 1999.

[17] ISO/IEC, "Software engineering –Product quality – Part 3: Internal metrics," ISO, ISO/IEC TR 9126-3, 2002.

[18] ISO/IEC, "9126-4: Software Engineering - Product Quality - Part 4:Quality in use metrics," ISO, ISO/IEC WD 9126-4, 1999.

[19] J. Boegh, "A New Standard for Quality Requirements," IEEE Software, vol. 25, no. 2, pp. 57-63, March-April 2008.

[20] ISO/IEC, "Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality," ISO, ISO/IEC WD 25023, 2011.

[21] Lionel C. Briand, Jurgen Wust, John W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," The Journal of Systems and Software, vol. 51, no. 3, pp. 245-273, May 2000.

[22] Parastooi Mohagheghi, Reidar Conradi, and Jon Arvid Børretzen, "Revisiting the problem of using problem reports for quality assessment," in Proceedings of the 2006 international workshop on Software quality, 2006, pp. 45-50.

[23] M. Glinz, "A Risk-Based, Value-Oriented Approach to Quality Requirements," IEEE Software, vol. 25, no. 2, pp. 34-41 , March-April 2008.

[24] H. Al-Kilidar, K. Cox, and B. Kitchenham, "The Use and Usefulness of the ISO/IEC 9126 Quality Standard," in Proceedings of International Symposium on Empirical Software Engineering, 2005.

[25] I. Padayachee, P. Kotze, and A. van Der Merwe, "ISO 9126 external systems quality characteristics, sub-characteristics and domain specific criteria for evaluating e-Learning systems," The Southern African Computer Lecturers' Association, University of Pretoria, South Africa, 2010.

[26] F Losavioa, L Chirinos, A Matteo, N Lévy, and A Ramdane-Cherif, "ISO quality standards for measuring architectures," Journal of Systems and Software, vol. 72, no. 2, pp. 209–223, July 2004.

[27] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung, "Measuring software product quality: a survey of ISO/IEC 9126," IEEE Software, vol. 21, no. 5, pp. 88-92, September-October 2004.

[28] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe, "Comparing software metrics tools," in ISSTA '08 Proceedings of the 2008 international symposium on Software testing and analysis, Seattle, Washington, USA, 2008, pp. 131-142.

[29] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller, "Mining metrics to predict component failures," in ICSE '06 Proceedings of the 28th international conference on Software engineering , Shanghai, China, 2006, pp. 452-461.

[30] Yue Jiang, Bojan Cukic, Tim Menzies, and Nick Bartlow, "Comparing design and code metrics for software quality prediction," in PROMISE '08 Proceedings of the 4th international workshop on Predictor models in software engineering, Leipzig, Germany, 2008, pp. 11-18.

[31] Yuming Zhou and Hareton Leung, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," IEEE Transactions on Software Engineering, vol. 32, no. 10, pp. 771-789, October 2006.

[32] Linda H. Rosenberg, "Applying and Interpreting Object Oriented Metrics," in Software Technology Conference, 1988.

[33] R. Subramanyam and M.S Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," IEEE Transactions on Software Engineering, vol. 29, no. 4, pp. 297- 310, April 2003.

[34] Seyyed Mohsen Jamali, "Object Oriented Metrics (A Survey Approach) ," Department of Computer Engineering Sharif University of Technology, Tehran, Iran, 2006.

[35] M Xenos, D Stavrinoudis, K Zikouli, and D Christodoulakis, "Object-oriented metrics-a survey," in Proceedings of the FESMA 2000, Federation of European Software Measurement Associations, Madrid, Spain, 2000.

[36] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer, "Managerial use of metrics for object-oriented software: an exploratory analysis ," IEEE Transactions on Software Engineering, vol. 24, no. 8, pp. 629-639, August 1998.

[37] C. Ebert and C. Jones, "Embedded Software: Facts, Figures, and Future," IEEE Computer, vol. 42, no. 4, pp. 42-52, April 2009.

[38] Mazhar Khaliq, Riyaz A. Khan, and M. H. Khan, "An Emprical Validation of Quality Testing Model for Object Oriented Design," International Journal of Mathematical Archive, vol. 2, no. 5, pp. 635-641, May 2011.

[39] G de Souza, D A Montini, D A da Silva, and F R M Cardoso, "Design Patterns Reuse for Real Time Embedded Software Development," in Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on, 2009, pp. 1421 - 1427.

[40] P. Liggesmeyer and M. Trapp, "Trends in Embedded Software Engineering," Software, IEEE, vol. 26, no. 3, pp. 19-25, May-June 2009.

[41] Redin, Ricardo; Oliveira, Marcio; Brisolara, Lisane; Mattos, Julio; Lamb, Luis; Wagner, Flávio; Carro, Luigi, "On the Use of Software Quality Metrics to Improve Physical Properties of Embedded Systems," in Distributed Embedded Systems: Design, Middleware and Resources, Bernd Kleinjohann, Wayne Wolf, and Lisa Kleinjohann, Eds.: Springer Boston, 2008, pp. 101-110.

[42] U.B. Corrêa, L. Lamb, L. Carro, L. Brisolara, and J. Mattos, "Towards Estimating Physical Properties of Embedded Systems using Software Quality Metrics," in Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, 2010, pp. 2381-2386.

[43] R. Amuthakkannan, S.M. Kannan, V. Selladurai, and K. Vijayalakshmi, "Software quality measurement and improvement for real-time systems using quality tools and techniques: a case study," International Journal of Industrial and Systems Engineering, vol. 3, no. 2, pp. 229-256, 2008.

[44] M.F.S. Oliveira, R.M. Redin, L. Carro, L. da Cunha Lamb, and F.R. Wagner, "Software Quality Metrics and their Impact on Embedded Software," in Model-based Methodologies for Pervasive and Embedded Software, 2008. MOMPES 2008. 5th International Workshop on, 2008, pp. 68-77.

[45] J.C.B. Mattos, E. Specht, B. Neves, and L. Carro, "Making Object Oriented Efficient for Embedded System Applications," in Integrated Circuits and Systems Design, 18th Symposium on, 2005, pp. 104-109.

[46] Weishan Zhang and Stan Jarzabek, "Reuse without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices," in Software Product Lines, Henk Obbink and Klaus Pohl, Eds.: Springer Berlin / Heidelberg, 2005, vol. 3714, pp. 57-69.

[47] Soner Çınar and Volkan Şirin, "Silah Sistemleri İçin Yeniden Yapılandırılabilir Bileşenler Yoluyla Platform Bağımsız Katmanlı Mimari Tasarımı," in 5. ULUSAL YAZILIM MÜHENDİSLİĞİ SEMPOZYUMU - UYMS'11, Ankara, 2011, pp. 197-204.

[48] Ertuğrul Barak, Sezen Erdem, and Hakan Yılmaz, "TADES: Komuta Kontrol Alanında bir Yazılım Ürün Hattı Çalısması," in 3. ULUSAL YAZILIM MÜHENDİSLİĞİ SEMPOZYUMU - UYMS'10, Ankara, 2010, pp. 88-97.

[49] Evrim Kahraman, Tolga İpek, Barış İyidir, Cüneyt F. Bazlamaçcı, and Semih Bilgen, "Bileşen Tabanlı Yazılım Ürün Hattı Geliştirmeye Yönelik Alan Mühendisliği Çalışmaları," in 4. ULUSAL YAZILIM MÜHENDİSLİĞİ SEMPOZYUMU - UYMS'09, Ankara, 2009, pp. 283-287.

[50] Campwood Software. (2012, July) SourceMonitor Version 3.2. [Online]. http://www.campwoodsw.com/sourcemonitor.html

[51] Tim Littlefair. (2012, October) CCCC - C and C++ Code Counter. [Online]. http://cccc.sourceforge.net/