INVESTIGATION OF THE EFFECTS OF STRUCTURAL CHARACTERISTICS OF OBJECT-ORIENTED SOFTWARE ON FAULT-PRONENESS

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

BY

HALİT GÖLCÜK

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

MAY 2014

Approval of the thesis:

INVESTIGATION OF THE EFFECTS OF STRUCTURAL CHARACTERISTICS OF OBJECT-ORIENTED SOFTWARE ON FAULT-PRONENESS

submitted by **HALİT GÖLCÜK** 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. Gönül Turhan Sayan _____
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

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

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

Barış İyidir, M. Sc. _____
Software Engineering Dept., Aselsan

Date: **30.05.2014**

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

Name, Last name: Halit Gölcük

Signature:

ABSTRACT

INVESTIGATION OF THE EFFECTS OF STRUCTURAL CHARACTERISTICS OF OBJECT-ORIENTED SOFTWARE ON FAULT-PRONENESS

Gölcük, Halit

M. Sc., Department of Electrical and Electronics Engineering

Supervisor: Prof Dr. Semih Bilgen

May 2014, 99 Pages

This study investigates the effects of structural characteristics of object-oriented software, which are observable at the model level of the software developed by means of Unified Modeling Language (UML), on software quality, assessing quality in terms of fault-proneness. In the scope of this thesis study, real-time embedded software components developed by Aselsan, a leading defense industry company in Turkey, were analyzed. The correlation between software metrics measured from the UML models of the software components and fault-proneness metrics of those software components were presented both graphically and statistically.

Keywords: Structural software characteristics, UML metrics, Software quality, Fault-proneness, Empirical study.

ÖZ

NESNE TABANLI YAZILIMLARIN YAPISAL ÖZELLİKLERİNİN HATA YATKINLIĞI ÜZERİNE ETKİLERİNİN İNCELENMESİ

Gölcük, Halit

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Semih Bilgen

Mayıs 2014, 99 Sayfa

Bu çalışma, Birleşik Modelleme Dili (UML) kullanılarak geliştirilen yazılımların model seviyesinde gözlenebilen bazı yapısal özelliklerinin yazılım kalitesi üzerine, kaliteyi yazılımın hataya yatkınlığı olarak tanımlayarak, etkilerini incelemektedir. Bu tez çalışması kapsamında, Türkiye'de önde gelen bir savunma sanayii kuruluşu olan Aselsan tarafından geliştirilen gerçek zamanlı gömülü yazılım bileşenleri analiz edilmiştir. Yazılım bileşenlerinin UML modellerinden ölçülen yazılım metrikleri ve bu bileşenlere ait hataya yatkınlık metrikleri arasındaki ilişki hem grafiksel hem de istatistiksel olarak ortaya konulmuştur.

Anahtar Kelimeler: Yazılımın yapısal özellikleri, UML metrikleri, Yazılım kalitesi, Hata yatkınlığı, Deneysel çalışma.

To my parents

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Prof. Dr. Semih Bilgen for his guidance, advice, understanding and supervision throughout the development of this thesis study.

I am grateful to Aselsan for the resources and facilities that I utilize for this thesis study. I also want to thank my colleagues at Aselsan for their valuable support.

I appreciate Barış İyidir for the fruitful advices and discussions throughout the development of this thesis study.

I am grateful to my team leader Evrim Kahraman and my colleague Sezgin Hayırlı for their tolerance during this study.

My special thanks go to Murat Yılmaz for reviewing this thesis.

I owe Aslı Serin a debt of gratitude for her love, support and understanding throughout this study.

Last but not the least; I would like to express my special thanks to my parents Nermin and Recep Gölcük, and my sisters Nuriye, Raviye, Fecriye, Ayfer and Nilüfer 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

AHF	Attribute Hiding Factor
AIF	Attribute Inheritance Factor
CBO	Coupling Between Objects
CC	Cyclomatic Complexity
CISQ	Consortium for IT Software Quality
CK	Chidamber and Kemerer
COF	Coupling Factor
DCC	Douglass Cyclomatic Complexity
DIT	Depth of Inheritance Tree
ISO	International Organization for Standardization
IEC	International Electrotechnical Commission
KLOC	Thousands of Lines of Code
LCOM	Lack of Cohesion in Methods
MaxDIT	Maximum DIT
MaxHAgg	Maximum HAgg
MHF	Method Hiding Factor
MIF	Method Inheritance Factor
n/a	Not Applicable
NA	Number of Attributes
NAc	Number of Activities
NAgg	Number of Aggregation
NAggH	Number of Aggregations Hierarchies
NAssoc	Number of Associations
NC	Number of Classes
NCS	Number of Composite States
NDep	Number of Dependencies
NE	Number of Events
NEntryA	Number of Entry Actions
NExitA	Number of Exit Actions
NEPI	Number of Events in Provided Interfaces
NG	Number of Guards
NGen	Number of Generalizations
NGenH	Number of Generalizations Hierarchies
NM	Number of Methods
NOC	Number of Children
NOPI	Number of Operations in Provided Interfaces
NPI	Number of Provided Interfaces
NSS	Number of Simple States
NT	Number of Transitions
OOSS	Object-Oriented Software Systems
POF	Polymorphism Factor

RFC	Response for a Class
SLOC	Source Lines of Code
SQuaRE	Software Product Quality Requirements and Evaluation
SST	ASELSAN Defense System Technologies (Tr. Savunma Sistem Teknolojileri)
UML	Unified Modeling Language
WMC	Weighted Methods per Class
YMM	ASELSAN Software Engineering Department (Tr. Yazılım “ Mühendisliği Müdürlüğü)

CHAPTER 1

INTRODUCTION

In a broad sense, the quality of a software artifact can be assessed in three aspects. These are functional, structural and process aspects of quality. Functional quality is related to the functional requirements of software. In other words, functional quality describes how well software meets functional requirements. The people who mostly care about functional quality are the users of the software. Structural quality is related with the nonfunctional requirements of software. It can be evaluated by analyzing the inner structure of software. According to the Consortium for IT Software Quality (CISQ), a software component with a good structural quality should be reliable, efficient, secure and maintainable [1]. Developers of the software are mostly interested in the structural quality. The last quality aspect of software is process quality. Process quality is related with the concerns like budget, delivery time, etc. Generally, software managers and sponsors are interested in this quality aspect.

In order to measure the quality of a software component, what is expected from software in terms of quality should be clarified. Fault tolerance of a software product can be a good quality indicator in terms of reliability of software. In literature, different types of metrics are used in order to represent the fault-proneness of a software product. Defect count and defect density are the most used metrics quality metrics in terms of fault-proneness [57]. There are studies in the literature that relate design metrics of software artifact to fault-proneness empirically, and develop prediction models to predict faulty software components in a software system, such as [35], [36] and [40]. These studies, generally, used linear regression methods to build and verify the prediction model. For object-oriented software, Chidamber and Kemerer's metrics suit [2] is the design metric set in the literature most frequently used for predicting faulty classes.

As the usage of modeling languages in software engineering is becoming more and more frequent, measuring and controlling the quality of a model is becoming more significant. Unified Modeling Language (UML) is the most widely used modeling language for object-oriented software development. In the literature, there is very little work which investigates the applicability of UML design metrics for measuring and controlling fault proneness of a software product despite the importance of early prediction of fault proneness [41].

In this thesis, the association between design metrics that reflect structural characteristics of object-oriented software and fault-proneness is investigated in the context of real-time embedded software development projects carried out by Aselsan, a leading defense industry company in Turkey. The design metrics were all measured from the UML models, not from source code, of the investigated software components. Fault-proneness metrics, which are defect count, defect density, defect severity and defect correction effort, were obtained from the defect tracking tool used in Aselsan. After the correlation between design metrics and fault-proneness of the software components were presented graphically and statistically, some useful outcomes were identified about which structural characteristic of object-oriented software have an impact on the quality of real-time embedded software component in terms of fault-proneness.

The remaining chapters of the thesis are structured as follows:

In Chapter 2, a literature survey is carried out about the concepts of software quality and the software characteristics considered in the scope of the study are reviewed.

In Chapter 3, the study framework, consisting of the structural characteristics of the investigated real-time embedded software components and the fault-proneness measures to be focused on, is presented. Selected software components and software metrics for the study are described, and how the software metrics are measured from the software components and how the fault data of the software components is gathered are explained.

In Chapter 4, software metric and fault-proneness measurement results are given. Then, the association between software metrics and fault-proneness measurements of software components is established graphically and statistically.

In Chapter 5, the achievements are summarized, obtained results are discussed and limitations of the study as well as suggestions for future work are outlined.

CHAPTER 2

LITERATURE REVIEW

2.1 INTRODUCTION

In the present study, case studies were carried out on software projects developed by Software Engineering Department (YMM) of Defense System Technologies Group (SST) in Aselsan. In this department, fire control software projects are developed by modeling the software in UML and software pieces has real-time embedded nature. As the first step of the study, the related literature was investigated in order to set the background consisting of the concepts of software model quality, quality of real-time embedded software and to determine the appropriate quality metrics.

2.2 UNIFIED MODELING LANGUAGE

UML is a general purpose modeling language which is used for modeling object-oriented software systems (OOSS). The standard is managed by the Object Management Group which is a not-for-profit computer industry specifications consortium [11]. UML 2.0 has thirteen types of diagrams that are used to visualize an OOSS. These thirteen types of diagrams can be divided into three categories; structure diagrams, behavior diagrams and interaction diagrams. In software components developed at YMM, class diagrams and state machine diagrams are widely used. Class diagram falls into category of structure diagrams and state machine diagram falls into category of behavior diagrams of UML. Class diagram represents the static view and state machine diagram represents the dynamic view of software. This study focuses on object-oriented metrics which are accessible from a UML model and class diagram and state chart diagram metrics because class

diagram and statechart diagram are used widely in software projects developed at YMM and also these two types of diagrams affect the performance of software considerably.

2.3 SOFTWARE QUALITY MEASUREMENT

In order to measure the quality of a software component, meaning of software quality should be clarified. In a broad sense, quality may be defined as satisfying the functional requirements [31]. It means that if a software product could not meet functional requirements, that software product is evaluated as software with poor quality. However, in the course of time, not only succeeding functional requirements but also performing functionalities effectively, safely and productively has gained importance [32]. This means that software product should meet nonfunctional requirements as well as functional requirements. Nonfunctional characteristics of a software product can be listed as reliability, performance efficiency, security and maintainability [1].

Software products with different features should implement different nonfunctional characteristics. For example, the most important nonfunctional requirement for a statistical analysis system can be reliability; however, a banking system should give precedence security characteristic, mostly [32]. Therefore, quality requirements study should be carried out in order to set proper quality objectives for a software product. Hneif and Lee [32] conducted a study in order to improve the nonfunctional quality of a piece of software. Their approach was preventive which means that their purpose was to eliminate the defects in the development phase not in the verification phase. Their purpose was to prevent nonfunctional attributes defects by using guidelines. For selecting guidelines, they used two properties; selected guideline should have positive effect on the nonfunctional attribute and selected guidelines for a specific nonfunctional attribute should not have overlapping or conflicting relationship between each other [32]. Philips et al. [33]

conducted an empirical study in order to discover quality requirements management practices in Australian organizations. The study included 13 practitioners and 6 different companies in Australia. The study showed that quality requirements management was the most neglected part of software projects [33].

In order to have software of high quality, using appropriate software quality models meeting quality requirements is as important as defining quality requirements correctly. By use of software quality prediction models and quality metrics, empirical studies are carried out in order to validate the quality of software product [34]. In order to validate the quality model and quality metrics, number of errors detected in the software [35], [36] or maintenance cost [37] can be used. However, using maintenance cost for statistical validation is difficult [34]. The quality model that will be used can be selected using two different strategies. One of them is using generic quality models such as ISO/IEC 9126. However, these quality models are often too abstract to use [38]. The other strategy is defining your own model based on existing quality models. The second strategy is more appropriate for finding the correct quality model that meets specific quality requirements [38]. Klass et al. [38] developed an approach that can be used while adapting the existing quality models to specific quality requirements. They identified three requirements for a quality model adaptation approach which are correctness, appropriateness and efficiency. Correctness means that adapted quality model should remain conformant to its structure. Appropriateness can be explained as quality model should be adapted considering the organizational needs and capabilities. Finally, efficiency stands for the level of overhead involved in the adaptation work in relation to the benefits of applying a proven quality model.

2.4 ISO/IEC 9126 AND ISO/IEC 25010

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) has released a standard, ISO/IEC 9126, in order

to describe a quality model for all types of software products. ISO/IEC 9126 consists of four parts. The first part of the standard is ISO/IEC 9126-1 which describes a quality model including 6 characteristics and 27 sub-characteristics [13], the second part of the standard is ISO/IEC 9126-2 which determines external quality metrics [14], the third part of the standard is ISO/IEC 9126-3 which determines internal quality metrics [15] and finally the fourth part of the standard is ISO/IEC 9126-4 which determines the quality in use metrics [16]. The standard aims to develop a quality model and quality metrics which are applicable to any type of software component. The organizations or individuals can use the quality model described in ISO/IEC 9126 in order to achieve the desired quality for a software product by adopting the quality model and quality metrics according to their needs.

ISO/IEC 9126 investigates software quality in three views; namely, external, internal and quality in-use. According to the standard, there is a close relationship in these three aspects of software quality. Internal quality attributes affect external quality attributes and external quality attributes affect quality in-use attributes. In other words in-use quality of a software product depends on external quality of the product and external quality depends on internal quality of the product [13].

ISO/IEC 9126 handles external and internal quality with one quality model, and handles in-use quality with another model. As shown in Table 1, in external and internal quality model, there are six characteristics and 27 sub-characteristics which are used to measure external and internal quality of a software product. Part 2 and Part 3 of the standard give some metrics in order to measure characteristics and sub-characteristics given in Part 1 of the standard. ISO/IEC 9126-2 gives external metrics and ISO/IEC 9126-3 gives internal metrics.

Table 1 - Characteristics and Sub-characteristics in External and Internal Quality Model in ISO/IEC 9126 [13]

Functionality	**Reliability**	**Usability**	**Efficiency**	**Maintainability**	**Portability**
Suitability	Maturity	Understandability	Time Behavior	Analyzability	Adaptability
Accuracy	Fault Tolerance	Learnability	Resource Utilization	Changeability	Installability
Interoperability	Recoverability	Operability	Efficiency Compliance	Stability	Co-existence
Security	Reliability Compliance	Attractiveness		Testability	Replaceability
Functionality Compliance		Usability Compliance		Maintainability Compliance	Portability Compliance

Quality model for quality in use identifies four characteristics; namely, effectiveness, productivity, safety and satisfaction [13]. Part 4 of the ISO/IEC 9126 gives quality in use metrics in order to measure the characteristics for quality in use given in Part 1 of the standard [16].

In order to achieve desired quality for a software component, it is necessary to perform measures from all the three views of quality (internal, external and quality in use) defined in ISO/IEC 9126 quality standard. It means that it is not sufficient to perform measures using quality in use metrics in order to meet the expectations of the users of the software product because in use quality of a software product is dependent to external quality of the product and by extension external quality is dependent to internal quality of the product [13].

Although ISO/IEC 9126 quality standard is a widely accepted and referred standard in the literature, there are also criticisms about the standard.

As stated earlier ISO/IEC 9126 aims to target any type of software quality needs. Therefore, the standard needs to be customized [17]. For example, functionality is the most important characteristic of your software product and you want to measure this characteristic using the related sub-characteristics and metrics defined in ISO/IEC 9126 quality model. When performing this measure how the contribution of sub-characteristics (security, interoperability, etc.) to the functionality characteristic be evaluated is unclear [17].

Applying the defined metric measurements to the software product is difficult because the metric definitions are imprecise; therefore, they need to be redefined in some cases [18]. In [19], an empirical study was reported in which the usefulness' of ISO/IEC 9126 standard was evaluated. In the experiment, 158 participants, which were in their final year in Computer Science and Engineering, were used. A Software Requirements Specification document was given to the subjects and they were asked to produce some design documents. Using ISO/IEC 9126 quality model, subjects were asked to evaluate the quality of the design artifacts [19]. As the result

of the experiment, some difficulties were observed about using ISO/IEC 9126 while evaluating the quality of an intermediate software product. The students in the experiment found it difficult to understand many terms in ISO/IEC 9126 and also they stated that some metric definitions were overlapping [19].

The results of the applied metric measures are given in the scale between 0 and 1. It is easy to convert these results to the percentage value; however, there is not any evaluation which values of the applied metric results are good or bad [20].

The difficulties of the ISO/IEC 9126 standard may make performing quality measures impractical. Therefore organizations or individuals who decide to use ISO/IEC 9126 quality standard for evaluating the quality of a software component have to adopt the standard according to their needs in a proper way.

In 2009, ISO and IEC released a new standard family, Software Product Quality Requirements and Evaluation (SQuaRE), which is also known as 25000 series. One part of 25000 series is the ISO/IEC 25010 which describes a quality model for a software component and it is intended that ISO/IEC 25010 will replace the existing ISO/IEC 9126 quality standard [21]. ISO/IEC 25010 has some improvements over ISO/IEC 9126; for example, ISO/IEC 25010 extends the quality concept by increasing the number of quality characteristics from 6 to 8. The characteristics and sub-characteristics proposed in ISO/IEC 25010 can be seen in Table 2.

Table 2 - Characteristics and Sub-characteristics in ISO/IEC 25010 [21]

Functional Suitability	**Reliability**	**Performance Efficiency**	**Operability**	**Security**	**Compatibility**	**Maintainability**	**Transferability**
Appropriateness	Availability	Time-behavior	Appropriateness recognisability	Confidentiality	Replaceability	Modularity	Portability
Accuracy	Fault tolerance	Resource-utilization	Learnability	Integrity	Co-existence	Reusability	Adaptability
Compliance	Recoverability	Compliance	Ease of use	Non-repudia-tion	Interoperability	Analyzability	Installability
	Compliance		Helpfulness	Accountability	Compliance	Changeability	Compliance
			Attractiveness	Authenticity		Modification stability	
			Technical accessibility	Compliance		Testability	
			Compliance			Compliance	

Although ISO/IEC 25010 has some improvements over ISO/IEC 9126, overall critiques about ISO/IEC 9126 are also applicable for ISO/IEC 25010 [30]. Mayr et al. [29] proposed a quality model for embedded systems based on ISO/IEC 25010. While deriving requirements for embedded system code they could not benefit from the standard because abstraction level of the quality characteristics described in ISO/IEC 25010 was high [29]. Wagner et al. [30] carried out a study in order to reduce the gap between high level quality models such as ISO/IEC 25010, and concrete quality models. They claimed that although ISO/IEC 25010 emphasized important concepts about software quality, the standard could not be used for real quality improvement purposes [30].

2.5 REAL-TIME EMBEDDED SOFTWARE QUALITY AND UML QUALITY ASSESSMENT

Real-time embedded software has wide application area with high reliability and security requirements. Therefore, software quality is significant in systems using real-time embedded software [42]. In order to provide advantages in the real-time software development world, improving software quality, minimizing software development cost and reducing software delivery time is critical because complexity of software products and competition in the market are increasing [39]. Fault proneness of a real-time software product can be used as quality indicator of that software [40]. Kaur et al. [40] proposed that, in order to identify fault proneness of a software module, requirement metrics, code metrics and the combination of these two metrics can be used with clustering techniques. They showed the applicability of the proposal by using the real-time defect datasets from NASA software projects. In a thesis study [43], effects of reuse on the quality of real-time embedded software were investigated. In that study, metrics were collected from real projects developed in Aselsan, and defect rates of the real-time embedded software projects were considered as quality indicator.

Model driven software development is becoming more popular in software engineering industry with the passing years. UML is the most commonly used modeling language for model driven software development. YMM, too, uses UML while modeling and developing the software.

2.6 SOFTWARE QUALITY METRICS

There are many studies and derived metrics in the literature for measuring the quality of software source code like [2], [23] and [24]. In their seminal work, Chidamber and Kemerer [2] defined six object-oriented quality metrics that are known as CK metrics. These metrics are aimed at assessing the design of OOSS rather than implementation. Another important metric set is known as MOOD metrics [23]. These were defined to measure the use of object-oriented design methods such as inheritance (MIF – Method Inheritance Factor, AIF – Attribute Inheritance Factor), information hiding (MHF – Method Hiding Factor, AHF – Attribute Hiding Factor) and polymorphism (POF – Polymorphism Factor, COF – Coupling Factor). It is widely accepted (e.g. [23]) that metrics should be easy to compute, should not be tied to any particular programming language and should result in numbers which is independent from the system size. In another thesis study [44], effects of software design patterns on object-oriented software quality and maintainability were investigated. In that study, maintainability was accepted as an important quality characteristic and maintainability of the applications developed in Aselsan were measured using CK metrics and MOOD metrics. Lorenz and Kidd [24] also proposed metrics in order to measure the static characteristics of software design. Their metrics are divided into three categories; class size, class inheritance and class internal. Size oriented metrics focus on counts of attributes and operations in a class. Inheritance oriented metrics focus on the manner in which operations are reused in hierarchy class. Internal class-oriented metrics address cohesion and code-oriented issues.

With the advent of model driven software development, not only measuring the quality of source code but also measuring the quality of models has become indispensable in order to develop software with adequate quality. Many organizations use UML models for various purposes such as implementation and maintenance while developing software components or systems and these models contain large number of defects that remain undetected [25]. Lange and Chaudron [26] performed two controlled experiment in order to investigate which types of defects in UML models remain undetected and effects of these defects. The first experiment was carried out with 111 students and the second experiment was carried out with 48 practitioners. The results showed that although some types of defects were determined by most subjects, undetermined defects caused misinterpretations among the readers. By analyzing the findings from [25] and [26], it can be concluded that measuring the quality of UML models is necessary.

In [22], Lange and Chaudron proposed a quality model for UML models based upon the necessity of measuring the quality of UML models. They proposed that UML models and source code differ in the aspect of system. Abstraction level in models is higher than the source code of the system, which means UML models describe systems in a non-deterministic way. Therefore, describing the quality characteristics of UML models is necessary. Lange and Chaudron's quality model combines the quality characteristics of the model with the quality characteristics of the system.

Nugroho et al. [41] empirically showed that UML design metrics are good predictors considering the fault proneness of a class. Metrics used in the study were derived from class diagrams and sequence diagrams of a UML model. In addition to UML model metrics, several code metrics were also used which are coupling between objects (CBO) [2], McCabe's complexity [9], and lines of code. These code metrics are well-known metrics that are in relation with fault-proneness of a class [41]. The question of which of the three fault prediction models; namely, UML metric model, code metric model, and UML and code metrics model, is

effective was answered with empirical data collected from industrial projects and the metric values measured. As a result, Nugroho et al.'s study showed that combination of UML design metrics and code metrics gives best performance for predicting fault proneness.

Assessing software quality using metrics is a proven approach for improving software quality [27]. While calculating software metrics, values can be calculated manually or tools can be used. SDMetrics [28] is a tool which is used in order to calculate software metrics for the object-oriented design quality of software systems designed and implemented by UML. The tool has some metrics such as number of attributes in the class (NumAttr) or number of operations in the class (NumOps). SDMetrics also provides opportunity to define custom metrics and rules to the users.

Below, the related literature is reviewed in order to determine the metrics which would be measured within the scope of the present study to assess structural characteristics of object-oriented software.

2.6.1 CK Metrics [2]

Names and definitions of CK Metrics [2] are summarized below:

- Weighted Methods per Class (WMC): This metric is defined as the summation of complexities of the methods defined in a class. If complexities of all methods are considered to be unity, WMC is equal to number of methods in a class [2]. Another approach is to consider the complexity of a method as McCabe's cyclomatic complexity [52].
- Depth of Inheritance Tree (DIT): It is a measure of the inheritance path from the node class to the root class.
- Number of Children (NOC): It is the number of immediate child or subclasses derived or subordinated from a base class.

- Coupling between Objects (CBO): It is a measure for cohesiveness between classes.
- Response for a Class (RFC): It is the number of methods which can be called in response to a message to a class.
- Lack of Cohesion in Methods (LCOM): It is a measure for cohesiveness in a class.

These metrics were originally devised in order to measure the quality of an object-oriented design based on the source code. In the thesis study by B. Deniz [43], while collecting metrics from real projects in order to measure software quality, CK metric suit was used. In that study, metrics were collected from source code. However, there are studies, like [3], [4], [52] in the literature, which also showed the applicability of these metrics to UML models.

CK metrics are accepted as good indicators of faulty classes in a software product [45], [46], [47]. In [45], the authors validated empirically the association between some of CK metrics and defects found during acceptance testing and defects found by customers. They used three of the six metrics of CK metric suit which are WMC, CBO and DIT.

2.6.2 Class Diagram Metrics [5]

Genero et al. [5] validated the given metric set for UML class diagrams empirically. It was proposed that understandability time of class diagrams is closely related with the maintainability of those diagrams. Metric names and definitions are given as follows [5].

- Number of Associations (NAssoc): It is the number of associations in a class diagram.
- Number of Aggregation (NAgg): It is the number of aggregation relationships within a class in a class diagram.

- Number of Dependencies (NDep): It is the number of dependency relationship in a class diagram.
- Number of Generalizations (NGen): It is the number of generalization relationships in a class diagram.
- Number of Aggregations Hierarchies (NAggH): It is the number of aggregation hierarchies in a class diagram.
- Number of Generalizations Hierarchies (NGenH): It is the number of generalization hierarchies in a class diagram.
- Maximum DIT (MaxDIT): It is the maximum of the DIT values obtained for each class of the class diagram.
- Maximum HAgg (MaxHAgg): It is the maximum of HAgg values obtained for each class of the class diagram. The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves.
- Number of Classes (NC): It is the number of classes in a class diagram.
- Number of Attributes (NA): It is the number of attributes of classes in a class diagram.
- Number of Methods (NM): It is the number of methods of classes in a class diagram.

2.6.3 Statechart Metrics [6]

The behavior of classes of a system can be described by using statecharts. Therefore, complexity of statecharts contributes to the complexity of classes. In this way, statechart complexity is associated with fault-proneness of the class [48].

There are several studies (e.g. [6], [7], and [8]) in the literature that proposed and validated some metrics for UML statecharts. These studies were carried out to prove the validity of statechart metrics in the view of understandability or maintainability.

The metrics with their definitions can be seen in Table 3.

Table 3 - Statechart Diagram Structural Complexity Metrics [6]

Metric Name	Metric Definition
Number of Entry Actions (NEntryA)	It is the number of entry actions in the statechart diagram.
Number of Exit Actions (NExitA)	It is the number of exit actions in the statechart diagram.
Number of Activities (NAc)	It is the number of do/Activities in the statechart diagram.
Number of Simple States (NSS)	It is the number of simple states in the statechart diagram.
Number of Composite States (NCS)	It is the number of composite states in the statechart diagram.
Number of Guards (NG)	It is the number of guard conditions in the statechart diagram.
Number of Events (NE)	It is the number of events in the statechart diagram.
Number of Transitions (NT)	It is the total number of transitions (common transitions + self-transitions + internal transitions)
Cyclomatic Complexity (CC)	McCabe's original cyclomatic complexity metric [9] is adapted as |NSS-NT+2| where NSS is the number of simple states and NT is the number of transitions.

Common transition represents the transition of which the source and target states are different; however, for self-transition, source and target states are same. Internal transition stands for the transitions that respond an event without leaving the state.

2.6.4 Douglass Metrics [10]

In a white paper [10], Douglass proposed a metric set to measure the complexity of a UML model. Some of the Douglass metrics are similar to the well-known metrics in the literature; however, all of the Douglass metrics are intended to use with UML

models. Metric sets are grouped as Model Organizational Metrics, Requirements Metrics, Model Architectural Structural Metrics, Model Semantic Structural Metrics and Model Behavioral Metrics.

The most attractive metric in the Douglass white paper is the Douglass Cyclomatic Complexity (DCC) because, to the best knowledge of the author of the present thesis, there is no other metric which handles nesting and concurrency in a state machine in the literature. The metric is the modified version of McCabe's Cyclomatic Complexity calculated as "Edges – Nodes + 2" [9]. The definition of Douglass Cyclomatic Complexity is "Edges – Nodes + 2 + Levels of Nesting + And-States".

2.7 MEASURING COMPONENT QUALITY

Cho et al. [54] discussed the limitations of existing object-oriented software metrics in measuring the quality of software components and they produced two reasons about the inadequacy of measuring component quality of object-oriented metrics. These reasons were difference in measurement unit and insufficiency in measurement factor. These reasons came up because object-oriented metrics only focus on objects or classes; however, software components have inter class relationships.

Vernazza et al. [55] proposed a way to use CK metrics in measuring the component quality, by considering software components as group of classes, by benefitting from the properties defined in [56]. Briand et al. [56] defined some properties that software measurements, like size, length, complexity, coupling and cohesion measurements, should satisfy.

2.8 FAULT-PRONENESS MEASUREMENT METHODS IN THE LITERATURE

As a quality metric, different fault-proneness measurement types can be used. Oyetoyan et al. [57] carried out a study in order to compare different defect measures in identifying fault-proneness software components. They analyzed four defect measures, namely; defect count, defect density, defect severity and defect correction effort in the scope of the study. Defect count refers to number of detected errors in a software component, defect density is the normalization of the defect count with respect to source lines of code (SLOC), defect severity is the importance or significance of the detected errors in the software, and defect correction effort is a measurement type for the difficulty of the correction of detected error in the software. In the literature, there are studies which used these different defect measures as a quality indicator. In [58], an oversampling method in order to improve fault-proneness prediction was proposed by using number of faults. Malaiya and Denton [59] provides a model in order to estimate and assess software reliability by using defect density metric. Oyetoyan et al. [57] concluded that although defect count and defect density measures are the mostly used defect measures in the literature, defect severity and defect correction effort are also effective in identifying critical and important software components. Also, several studies [61]-[63] showed that there is a connection between size and defect density; relatively larger components in size are tend to be lower defect densities.

CHAPTER 3

THE STUDY FRAMEWORK

Software components developed by YMM are developed according to organizationally accepted and published software reference architecture. In this way, design maturity of the software components are guaranteed to some extent. However, it has been observed that quality of the software components depends on the developer because there is limited design information for the software components, except the interfaces defined, in the reference software architecture. Therefore, it was decided that in addition to the published reference architecture, specifying the structural characteristics for the software components developed as compatible with the reference software architecture would be beneficial. The purpose of the present study is to assure the quality level of software components to be developed and reveal the quality level of software components already developed. Software components developed by YMM are developed with a UML tool; therefore, it is important to collect software metrics from UML models. The results of the present study will be used to guide the software developers in order to develop software components with higher quality.

In the present study, it was accepted that fault-proneness of a software component is a good indicator of its quality. Some software components were chosen from real projects developed by YMM. Some metrics were collected from the software components, metrics and faults of software components were analyzed in order to find out which metric could be significant in evaluating fault-proneness of real-time software components designed with UML.

3.1. GENERAL INFORMATION ABOUT THE SOFTWARE DEVELOPER TEAM

Throughout the present study, we worked with a software developer team in YMM. The team develops software for fire control systems using C++ language. Fire control systems process data coming from many sensing units in order to increase the possibility of shooting the ammunition at the target [49]. The software developed by the team has real-time embedded nature. While developing software, the team uses a UML tool which has the capability of automatic code generation. The software developer team uses a reference software architecture which is called Weapon Systems Reference Architecture (SSRM) [49]. General information about the software architecture can be found in the next sub-section.

3.1.1. Reference Software Architecture used by the Software Team

The reference software architecture of the software team has been designed according to Feature-Oriented Reuse Method (FORM). FORM develops domain architectures and components by capturing commonalities and differences of applications in a domain in terms of "features" [50]. Fundamental components of the software architecture with their short explanations are as follows;

- Missions carry out a specific functionality which differs with reference to project requirements.
- Capabilities provide foundation for performing a specific mission; for example, target management, platform management. Capabilities are developed as reusable components.
- Software Manager decides that which component of the software would be active according to user requests or current situation of the software. This component of the software architecture is project specific; therefore, it is not designed as reusable.

- External Interface represents user interface, command control interface, etc. The purpose of defining such a component in software is blocking the variability in the external environment from the software.
- System Environment transfers the services of sensors and actuators to the software.
- Operating Environment provides the independence of the software from the hardware and operating system that the software is running on.

Data communication and control operations are separated in order to increase reusability and decrease maintainability effort; therefore, there are two views in the architecture; which are control view and data view. Flow of the data and control commands are also determined; pull method is used for data communication and push method is used for control operations.

3.2. SOFTWARE COMPONENTS INVESTIGATED

The software components investigated within the scope of the present study were all within System Environment Layer of the software architecture. The highest necessity to develop new software components is for the System Environment Layer for the software developer team, and also the highest fault count per software module is known to occur in System Environment Layer components. Therefore, collecting software components to be analyzed from System Environment Layer was considered reasonable and especially beneficial in the scope of the present work.

There are many software components in System Environment Layer; however, 10 components were selected from among them. The criterion in selecting software components was that the software components should be used in at least two projects which are completed and delivered to the customer. The reason for this criterion was that the software components should be tested adequately and therefore, they would be reliable components.

3.3. SELECTED METRICS

In selecting the software metrics to be measured in the present study, requirements and priorities of YMM have been kept in consideration. In order to determine which metric or metric set would be used, first, literature was reviewed and a broad set of metrics that suit the purpose of the thesis study were presented to the software leaders of the development staff of the projects from which measurement would be obtained. With the software leaders, design metrics that reflect structural characteristics of software components developed by YMM are evaluated and selected.

Metric selection together with justifications can be found in Table 4. "Metric Selection" column indicates whether concerned metric will be measured or not. "+" means the metric shall be measured, "-" means the metric shall not be measured. "Comment" column gives the reason behind the metric selection.

Table 4 - Motivation behind Metric Selection

Metric Set	Metric Selection	Metric Name	Comment
CK Metrics	+	WMC (Weighted Methods per Class)	CK Metrics are referred as good indicators in determining fault-proneness of a class (see "CK Metrics" sub-section in Chapter 2); therefore, this metric set was completely applied in this thesis study. However, metric values were gathered from the UML models rather than source code.
	+	DIT (Depth of Inheritance)	
	+	NOC (Number of Children)	
	+	CBO (Coupling Between Objects)	
	+	RFC (Response for a Class)	
	+	LCOM (Lack of Cohesion in Methods)	

Table 4 (cont'd)

Class Diagram Metrics	-	NAssoc (Number of Associations)	This metric was not measured because association relationship is generally used via ports; direct association relationship is not established in software components developed by YMM.
	-	NAgg (Number of Aggregation)	This metric was not measured because aggregation relationship is not used in software components developed by YMM.
	-	NAggH (Number of Aggregations Hierarchies)	
	-	MaxHAgg (Maximum HAgg)	
	-	NGen (Number of Generalizations)	In YMM, interface inheritance is used rather than implementation inheritance; therefore, these metrics were not measured.
	-	NGenH (Number of Generalizations Hierarchies)	
	-	MaxDIT (Maximum DIT)	In CK Metrics, there is a metric as 'DIT'. Therefore, it was evaluated that there is no need to use this metric.
	-	NDep (Number of Dependencies)	This metric was not measured because software components developed by YMM depend only to the reference software architecture.
	+	NC (Number of Classes)	We expected to establish a relationship between size and structural complexity of a software component by measuring this metric.
	+	NA (Number of Attributes)	
	+	NM (Number of Methods)	
Statechart Metrics	+	NEntryA (Number of Entry Actions)	In this thesis study, the association between structural complexity of statechart of a class and fault-proneness of that class was investigated. Therefore all of the statechart metrics defined in [6] were measured.
	+	NExitA (Number of Exit Actions)	
	+	NAc (Number of Activities)	
	+	NSS (Number of Simple States)	
	+	NCS (Number of Composite States)	

Table 4 (cont'd)

	+	NG (Number of Guards)	
	+	NE (Number of Events)	
	+	NT (Number of Transitions)	
	+	CC (Cyclomatic Complexity)	
Douglass Metrics	+	DCC (Douglass Cyclomatic Complexity)	DCC metric accounts for nesting and concurrency in a state machine which affects the complexity of a state machine.
	-	Other Douglass Metrics	Other Douglass metrics have common parts with CK Metrics, Class Diagram Metrics and Statechart Metrics; therefore, picking up only DCC metric from among Douglass metrics was considered as sufficient for the present study.

In addition to metrics, that reflect structural characteristics of object-oriented software, proposed in the literature, some metrics were considered to have association with complexity of a software component; thus, error-proneness of that software component. These metrics are related with the interface complexity of a software component, which are Number of Provided Interfaces (NPI), Number of Events in Provided Interfaces (NEPI) and Number of Operations in Provided Interfaces (NOPI).

3.4 MEASUREMENT OF METRICS

There were three possible ways for measuring metrics for the software components; these were:

- Manual metric measurement,
- Metric measurement by using a readymade metric tool,

- Automating metric measurement for the specific needs of the thesis study.

Metric measurement can be carried out by examining the UML model of the software manually. This way of metric measurement has been skipped because calculating metric values manually can be time consuming for complex software components and it is not reliable.

A tool that can be used to get metric values from a UML model came to the forefront while reviewing the literature in order to see whether measurement of metrics can be done with a readymade tool or not. This tool was SDMetrics. SDMetrics can be used with all UML modeling tools which has the capability of exporting the model to an XMI file. The tool also gives the opportunity of defining custom metrics (see "Metrics in the Literature" sub-section in Chapter 2). SDMetrics was considered to be appropriate for the needs of the thesis study; however, some difficulties have been encountered;

- XMI output of the UML design tool used by the software developer team deviated from the standard at some points. For example, default transitions in the statechart were represented as states in the XMI output. Also, internal and external transitions in a state were not distinguished at the XMI output of the tool.
- SDMetrics tool, as expected, could not measure all metrics in the scope of the thesis study; therefore, the ability of defining new metrics within the tool was attempted; however, the tool interface was not user friendly.

The UML design tool used by the software developer team supports java plug-ins which can be used to extend the product capabilities. By creating java applications, one can modify and analyze the UML model created by the UML design tool.

After the metric measurement options were reviewed, it was observed that most of the metrics could be measured by creating java plug-in applications for the specific needs of the thesis study and, for some metrics, readymade metric measurement tools could be used.

3.5 OBTAINING FAULT STATISTICS FOR THE SOFTWARE COMPONENTS

In Aselsan, a tool is used for the purpose of defect tracking. With this tool, every type of change activity associated with software development, including enhancement requests, defect reports, and documentation modifications can be managed. The defect counts regarding to the software components analyzed in the present study were collected by using this tool. However, defect records were not assigned to the software components; they were only associated with the project. Therefore, at this point, support was received from the software leaders of the projects from which measurement would be obtained, and the numbers of faults regarding the software components were gathered.

CHAPTER 4

EXPERIMENTAL WORK

In the part of the study to be presented in this chapter, metric measurements were correlated with fault-proneness metrics of the software components. Before carrying out metric measurement process, some metric definitions were needed to be clarified.

4.1 METRIC DEFINITIONS RE-VISITED

4.1.1 Number of Transitions (NT) [6]

NT metric definition is given as total number of transitions including common transitions, self-transitions and internal transitions in [6]. Although Genero et al. [51] give the same definition for NT metric, they calculate the metric in a different way; that is, they exclude the internal transitions in the calculation. In the present study, NT metric was calculated as summation of common transitions and self-transitions.

4.1.2 Cyclomatic Complexity (CC) [6]

CC metric definition is given as |NSS – NT + 2| in [6]. However, McCabe [9] gives the cyclomatic complexity definition as “edges – nodes + 2” for a connected graph. Also, Douglass [10] uses the number of states (NS) as nodes not number of simple states (NSS). Therefore, it was evaluated that taking CC metric definition as |NT - NS + 2| is more appropriate.

4.1.3 Coupling between Objects (CBO) [2]

CBO metric was formed as the summation of the coupling related metrics given by SDMetrics tool.

4.1.4 Response for a Class (RFC) [2]

RFC metric was formed as the summation of the number of messages received by the class given by SDMetrics tool as “MsgRecv”, NEPI and NOPI.

4.2 METRIC MEASUREMENT RESULTS

Metric values were mostly obtained by use of java plug-in applications for the UML tool used by the software development team and, for some metrics, readymade measurement tools were used (see “Measurement of Metrics” sub-section in Chapter 3).

All of the software metrics, except WMC, were collected from the UML models of the software components, i.e. without generating source codes of the software components. WMC metric could not be measured from the UML models because, to the best knowledge of the author of the present thesis, there is not any tool calculating WMC metric from UML model, and calculating this metric manually is very time consuming and not reliable. Therefore, a free internet search was conducted in order to select a tool to calculate WMC metric from the generated code (source code). The tool should have the feature of calculating cyclomatic complexity for each method of a class because while calculating WMC metric, methods defined at the model level were taken into consideration. So, CCM [53],

which is a tool analyzing cyclomatic complexity of C/C++, C#, JavaScript and TypeScript code, was selected.

A tool which measures LCOM metric from source code or UML model could not be found. Also the metric could not be obtained from a java plug-in application. Therefore, LCOM metric was omitted from the evaluation.

Metric measurement techniques are outlined in Table 5.

Table 5 - Metric Measurement Techniques

Metric Set	Metric ID	Metric Name	Metric Measurement Technique
CK Metrics	M1	WMC	CCM
	M2	DIT	SDMetrics
	M3	NOC	SDMetrics
	M4	CBO	SDMetrics
	M5	RFC	SDMetrics + Java Plug-in
Class Diagram Metrics	M6	NC	Java Plug-in
	M7	NA	Java Plug-in
	M8	NM	Java Plug-in
Statechart Metrics	M9	NEntryA	Java Plug-in
	M10	NExitA	Java Plug-in
	M11	NAc	Java Plug-in
	M12	NSS	Java Plug-in
	M13	NCS	Java Plug-in
	M14	NG	Java Plug-in
	M15	NE	Java Plug-in
	M16	NT	Java Plug-in
	M17	CC	Java Plug-in
	M18	DCC	Java Plug-in
Interface Metrics	M19	NPI	Java Plug-in
	M20	NEPI	Java Plug-in
	M21	NOPI	Java Plug-in

Measured metric values are listed in Tables 6 and 7. There are 10 software components analyzed in the scope of the study, and these components are named as Component_1, Component_2, etc.

Table 6 - Measured metric values for first 5 components

	Component_1	Component_2	Component_3					Component_4			Component_5		
	Class_1	Class_1	Class_1	Class_2	Class_3	Class_4	Class_5	Class_1	Class_2	Class_3	Class_1	Class_2	Class_3
M1	143	236	227	1	2	2	6	67	131	3	72	176	3
M2	0	0	0	0	0	0	0	0	1	0	0	1	0
M3	0	0	0	0	0	0	0	0	0	0	0	0	0
M4	3	1	7	2	2	2	3	22	44	2	20	49	2
M5	29	30	62	0	2	2	0	4	33	0	5	42	0
M6	1	1	5					3			3		
M7	26	41	117					48			46		
M8	35	26	61					68			50		
M9	16	24	87	1	n/a	n/a	n/a	25	13	n/a	22	13	n/a
M10	3	1	11	0	n/a	n/a	n/a	5	0	n/a	2	0	n/a
M11	29	15	93	0	n/a	n/a	n/a	25	2	n/a	26	1	n/a
M12	16	21	86	2	n/a	n/a	n/a	28	13	n/a	25	12	n/a
M13	10	14	34	0	n/a	n/a	n/a	28	8	n/a	20	13	n/a
M14	11	8	84	0	n/a	n/a	n/a	28	1	n/a	30	1	n/a
M15	26	31	105	0	n/a	n/a	n/a	39	18	n/a	36	19	n/a
M16	37	48	191	2	n/a	n/a	n/a	72	26	n/a	65	26	n/a
M17	13	15	73	2	n/a	n/a	n/a	18	7	n/a	22	3	n/a
M18	19	22	87	2	n/a	n/a	n/a	30	13	n/a	29	11	n/a
M19	4	5	8					3			3		
M20	15	15	34					18			16		
M21	14	15	32					10			8		

Table 7 - Measured metric values for last 5 components

	Component_6					Component_7	Component_8				Component_9	Component_10
	Class_1	Class_2	Class_3	Class_4	Class_5	Class_1	Class_1	Class_2	Class_3	Class_4	Class_1	Class_1
M1	180	33	48	7	1	233	55	3	3	3	117	260
M2	0	0	0	0	0	0	0	0	0	0	0	0
M3	0	0	0	0	0	0	0	0	0	0	0	0
M4	7	5	4	4	4	2	4	2	2	5	1	1
M5	44	0	20	3	0	27	16	2	2	0	26	26
M6	5					1	4				1	1
M7	79					21	22				12	34
M8	76					23	25				13	20
M9	13	7	2	n/a	n/a	4	5	n/a	n/a	n/a	6	5
M10	2	2	0	n/a	n/a	0	2	n/a	n/a	n/a	0	0
M11	34	3	4	n/a	n/a	3	3	n/a	n/a	n/a	7	5
M12	15	6	3	n/a	n/a	10	4	n/a	n/a	n/a	6	4
M13	9	1	1	n/a	n/a	7	3	n/a	n/a	n/a	4	4
M14	7	6	0	n/a	n/a	1	2	n/a	n/a	n/a	1	0
M15	19	4	3	n/a	n/a	14	4	n/a	n/a	n/a	5	3
M16	29	13	5	n/a	n/a	21	7	n/a	n/a	n/a	8	6
M17	7	8	3	n/a	n/a	6	2	n/a	n/a	n/a	0	0
M18	11	9	4	n/a	n/a	10	5	n/a	n/a	n/a	3	3
M19	7					7	4				4	6
M20	36					6	6				11	9
M21	31					21	14				15	17

Selected metrics were generally measured at the class level except Class Diagram Metrics and Interface Metrics as seen in Table 6 and Table 7. Therefore, it was required to combine CK Metrics and Statechart Metrics to measure the component quality (see “Measuring Component Quality” section in Chapter 2). In order to map CK Metrics to component metrics, the proposed method given in [55] was used. According to the method, in order to map WMC, NOC and RFC to component level, they should be summed up; in order to map DIT to component level, it should be taken maximum of DIT of individual classes inside the component; and in order to map CBO metric to component level, number of external classes coupled to the classes inside the component should be considered. For the Statechart Metrics, the same idea was used by accepting Statechart Metrics as complexity and size metrics. So, Statechart Metrics of individual classes were summed up in order to map these metrics to component metrics.

After CK Metrics were mapped to component metrics, metric measurements became as in Table 8.

Table 8 - Metric Values for the Components

	Component_1	Component_2	Component_3	Component_4	Component_5	Component_6	Component_7	Component_8	Component_9	Component_10
M1	143	236	238	201	251	269	233	64	117	260
M2	0	0	0	1	1	0	0	0	0	0
M3	0	0	0	0	0	0	0	0	0	0
M4	12	12	15	12	12	17	16	10	10	16
M5	29	30	66	37	47	67	27	20	26	26
M6	1	1	5	3	3	5	1	4	1	1
M7	26	41	117	48	46	79	21	22	12	34
M8	35	26	61	68	50	76	23	25	13	20
M9	16	24	88	38	35	22	4	5	6	5
M10	3	1	11	5	2	4	0	2	0	0
M11	29	15	93	27	27	41	3	3	7	5
M12	16	21	88	41	37	24	10	4	6	4
M13	10	14	34	36	33	11	7	3	4	4
M14	11	8	84	29	31	13	1	2	1	0
M15	26	31	105	57	55	26	14	4	5	3
M16	37	48	193	98	91	47	21	7	8	6
M17	13	15	75	25	25	18	6	2	0	0
M18	19	22	89	43	40	24	10	5	3	3
M19	4	5	8	3	3	7	7	4	4	9
M20	15	15	34	18	16	36	6	6	11	9
M21	14	15	32	10	8	31	21	14	15	17

4.3 FAULT STATISTICS OF THE SOFTWARE COMPONENTS

Errors associated with the software components were obtained with the help of software leaders of the projects from which measurement would be obtained (see “Obtaining Fault Statistics for the Software Components” section in Chapter 3). Approximately 2500 defects, which are all removed defects, were analyzed with the software leaders and it was determined that 183 of 2500 defects concerned the investigated software components. Fault-proneness metrics of the software components are given for each component in Table 9.

Table 9 - Fault-proneness Metrics of the Software Components

	Defect Count	Defect Density	Defect Severity	Defect Correction Effort
Component_1	5	0,794	14	7
Component_2	24	3,701	76	88
Component_3	64	2,678	232	206
Component_4	26	1,819	79	63
Component_5	11	0,766	34	17
Component_6	17	1,502	58	33
Component_7	15	3,95	55	22
Component_8	0	0	0	0
Component_9	2	0,53	7	1
Component_10	20	5,249	58	45

Four different fault-proneness measurement types were used as suggested by related literature (see “Fault-Proneness Measurement Methods in the Literature” section in Chapter 2). These measures, which are defect count, defect density, defect severity and defect correction effort, were obtained from the software defect tracking tool used in Aselsan. Defect count is simply the error count associated with the software component. Defect density is the error count for every thousands of lines of source code (KLOC) of the software component which means that defect density is equal to “defect count / KLOC”. Every error recorded by the defect tracking tool is evaluated in terms of severity by the test engineer who detected the related error. Test engineer evaluates the severity of error in five degree. Defect severity was obtained by making use of this data, 5 point was given to the most severe error and 1 point is given to the least severe error. In this way, defect severity was obtained as the summation of this scaled data. After test engineer recorded an error using the defect tracking tool, software developer fixes the error and enters how many hours spent in order to fix the error. As defect correction effort, this measurement was used.

4.4 CORRELATION BETWEEN SOFTWARE METRICS AND FAULT-PRONENESS OF THE SOFTWARE COMPONENTS

Before trying to correlate the calculated metric values with fault-proneness of the software components, when we looked over the metric values from Table 8, it could be seen that M2 (DIT, CK Metrics) and M3 (NOC, CK Metrics) metric measurements were not meaningful. Only Component_4 and Component_5 have a measurement as “1” and all other software components have “0” measurement for M2, and all of the measurements are “0” for M3. This means that inheritance property of object-oriented programming is not widely used in YMM. Therefore, the association between DIT and NOC metrics of CK Metrics and the fault-proneness of software components could not be determined in the scope of this thesis study; hence, M2 and M3 metrics were left out of the scope of the study.

4.4.1 Finding the Correlation between Software Metrics and Fault-Proneness of the Software Components Graphically

First, the association between calculated software metrics and fault-proneness of the software components was presented graphically. All the graphs showing the correlation between design metrics and fault-proneness metrics are presented in the Appendix section.

The numbered points in the graphs, which are in the Appendix section, represent the software components; for example, 1 represents the Component_1, 2 represents the Component_2, etc.

Figures 1 through 19 show the association between defect count and respective calculated metrics. As seen from the figures, all of the software metrics have positive relationship with the defect count, as expected. This means that structural characteristics represented by the design metrics in the scope of the study are closely related with number of errors detected in software.

Figures 20 through 38 show the association between defect density and respective calculated metrics. Metrics other than M1, M4, M7, M19 and M21 have negative correlation or do not have strong positive correlation with defect density of the software components. Indeed, as in the case for defect count, positive association between metrics and defect density was expected. The reason behind the situation might be that software components, which are in the scope of the study, have different source lines of code varying between 3771 (Component_9) and 23893 (Component_3). Larger software components tend to have lower defect densities as suggested in [62]-[64]. Therefore, it can be concluded that using defect density as a quality metric is convenient for software components which have close number of source lines of code.

Figures 39 through 57 show the association between defect severity and respective calculated metrics. As is seen from the figures, all of the software metrics have positive correlation with the defect severity, as expected. This means that structural characteristics represented by the design metrics in the scope of the present study cause more severe errors in software.

Figures 58 through 76 show the association between Defect Correction Effort and respective calculated metrics. As is seen from the figures, all of the software metrics have positive correlation with the defect correction effort, as expected. This means that structural characteristics represented by the design metrics in the scope of the present study result in raise in defect correction effort of software components.

4.4.2 Statistical Evaluation

4.4.2.1 Pearson's Correlation Coefficient

In order to correlate software metrics with the fault-proneness of the software components, first, it was decided to use Pearson's correlation coefficient [60]

because it was expected that a linear correlation between the metrics and fault-proneness of the components exists.

Pearson's correlation coefficient, which is represented as r, measures the linear relationship between two continuous variables. The coefficient also shows the direction of relationship; the values between 0 and +1 represent positive association and the values between 0 and -1 represent negative association.

The data collected should not have significant outliers and should be distributed normally in order to use Pearson's correlation coefficient. Therefore; first, outlier and normality tests were applied to the collected data. Kolmogorov-Smirnov test [60] was applied as normality test. The results of outlier and normality tests are presented in Table 10.

Table 10 - Outlier and Normality Test Results

	Outliers	**Normality**
Defect Count	Component_3	Normal
Defect Density	No outlier	Normal
Defect Severity	Component_3	Non-normal
Defect Correction Effort	Component_3	Normal

As seen from Table 10, only defect density data was appropriate for Pearson's correlation coefficient analysis. When Component_3 was excluded from the analysis, there were no outliers and all the measures had normal distribution; however, it was decided to hold Component_3 inside the analysis because Component_3, with its design metrics and fault-proneness metrics values, was significant for the analysis. Therefore, it was decided to continue to the analyses with Pearson's correlation coefficient although defect count, defect severity and defect correction effort measures had an outlier and defect severity had non-normal distribution. Also, some studies claim that Pearson's correlation coefficient is somewhat robust to deviations from normality [61]. Besides, when the analysis

were carried out by excluding Component_3, statistical significance of the results, in other words p value, was below the 95% confidence level for most of the design metrics.

The results for Pearson's correlation coefficient are outlined in Table 11.

Table 11 - Pearson's Correlation Coefficients between the Metrics and Fault-Proneness Measures

	Defect Count	Defect Density	Defect Severity	Defect Correction Effort
M1	0,530 p = 0,115	0,656 p = 0,040	0,497 p = 0,144	0,426 p = 0,220
M4	0,453 p = 0,189	0,650 p = 0,042	0,449 p = 0,193	0,327 p = 0,357
M5	0,638 p = 0,047	-0,58 p = 0,873	0,653 p = 0,040	0,579 p = 0,079
M6	0,431 p = 0,214	-0,358 p = 0,309	0,456 p = 0,185	0,409 p = 0,241
M7	0,863 p = 0,001	0,119 p = 0,744	0,871 p = 0,001	0,834 p = 0,003
M8	0,483 p = 0,157	-0,193 p = 0,594	0,472 p = 0,168	0,402 p = 0,250
M9	0,882 p = 0,001	0,003 p = 0,993	0,889 p = 0,001	0,882 p = 0,001
M10	0,808 p = 0,005	-0,126 p = 0,729	0,824 p = 0,003	0,801 p = 0,005
M11	0,825 p = 0,003	-0,056 p = 0,877	0,845 p = 0,002	0,814 p = 0,004
M12	0,878 p = 0,001	0,005 p = 0,990	0,887 p = 0,001	0,862 p = 0,001
M13	0,620 p = 0,056	-0,089 p = 0,807	0,599 p = 0,067	0,575 p = 0,082
M14	0,857 p = 0,002	-0,035 p = 0,924	0,871 p = 0,001	0,846 p = 0,002
M15	0,833 p = 0,003	-0,018 p = 0,960	0,837 p = 0,003	0,817 p = 0,004
M16	0,857 p = 0,002	-0,006 p = 0,986	0,862 p = 0,001	0,839 p = 0,002
M17	0,888 p = 0,001	0,018 p = 0,960	0,904 p = 0,000	0,882 p = 0,001
M18	0,861 p = 0,001	-0,012 p = 0,974	0,869 p = 0,001	0,845 p = 0,002
M19	0,597 p = 0,068	0,558 p = 0,094	0,629 p = 0,051	0,555 p = 0,096
M20	0,635 p = 0,049	-0,086 p = 0,814	0,644 p = 0,044	0,605 p = 0,064
M21	0,586 p = 0,075	0,252 p = 0,482	0,626 p = 0,053	0,554 p = 0,096

The correlation between the two variables is statistically significant if the p value is below 0.05 ($p < 0.05$). Table 12 summarizes the metrics which have statistically significant correlations with the fault-proneness measures of the software components with the strength of the correlation as low correlation, medium correlation and high correlation. Although there is no general rule about the classification of the strength of the correlation coefficient, Cohen [60] suggests that if the absolute value of correlation coefficient is between 0.1 and 0.3, the strength of the association is low, if the absolute value of correlation coefficient is between 0.3 and 0.5, the strength of the association is medium, and if the absolute value of correlation coefficient is greater than 0.5, the strength of the association is high.

Table 12 - Statistically Significant Pearson's Correlations

	High Correlation	**Medium Correlation**	**Low Correlation**
Defect Count	M5, M7, M9, M10, M11, M12, M14, M15, M16, M17, M18, M20	-	-
Defect Density	M1, M4	-	-
Defect Severity	M5, M7, M9, M10, M11, M12, M14, M15, M16, M17, M18, M20	-	-
Defect Correction Effort	M7, M9, M10, M11, M12, M14, M15, M16, M17, M18	-	-

Statistically significant associations between the metrics and fault-proneness of the software components all display high correlations ($|r| > 0.5$) as seen from Table 12. Metrics which are correlated with defect count and defect severity of the software components are all the same, and, for defect correction effort, only two of them (M5 and M20) are not statistically significant. It seems that M1 and M4 affect the defect density of a software component although they do not have statistically significant correlations with other fault-proneness measures; namely, defect count, defect

severity, and defect correction effort. The results of Pearson's correlation coefficient are actually consistent with the graphical display of the associations between the metrics and fault-proneness of the software components; the only difference with finding Pearson's correlation coefficient is that statistically significant correlations have been pointed out and the associations have been indicated numerically.

4.4.2.2 Spearman's Correlation Coefficient

As defect count, defect severity and defect correction effort measures had an outlier and defect severity had non-normal distribution, the metrics and fault-proneness measures were also investigated with Spearman's correlation coefficient [60], which is a non-parametric test as well, because Spearman's correlation coefficient does not require that the data do not have outliers or the data should be normally distributed. As in the case for Pearson's correlation coefficient, each of the metrics was correlated separately to the fault-proneness measures of the software components. The results for Spearman's correlation coefficient are outlined in Table 13.

Table 13 - Spearman's Correlation Coefficients between the Metrics and Fault-Proneness Measures

	Defect Count	Defect Density	Defect Severity	Defect Correction Effort
M1	0,552 p = 0,098	0,539 p = 0,108	0,559 p = 0,093	0,576 p = 0,082
M4	0,529 p = 0,116	0,717 p = 0,019	0,540 p = 0,107	0,529 p = 0,116
M5	0,596 p = 0,069	0,140 p = 0,700	0,643 p = 0,045	0,584 p = 0,077
M6	0,221 p = 0,539	-0,299 p = 0,401	0,264 p = 0,460	0,176 p = 0,627
M7	0,745 p = 0,013	0,224 p = 0,533	0,772 p = 0,009	0,721 p = 0,019
M8	0,479 p = 0,162	-0,055 p = 0,881	0,529 p = 0,116	0,430 p = 0,214
M9	0,614 p = 0,059	-0,049 p = 0,894	0,637 p = 0,048	0,590 p = 0,073
M10	0,431 p = 0,214	-0,178 p = 0,622	0,469 p = 0,171	0,369 p = 0,294
M11	0,463 p = 0,177	-0,049 p = 0,894	0,502 p = 0,140	0,445 p = 0,197
M12	0,650 p = 0,042	0,085 p = 0,815	0,686 p = 0,029	0,614 p = 0,059
M13	0,729 p = 0,017	0,213 p = 0,555	0,753 p = 0,012	0,693 p = 0,026
M14	0,413 p = 0,235	-0,207 p = 0,567	0,451 p = 0,191	0,377 p = 0,283
M15	0,620 p = 0,056	0,061 p = 0,868	0,649 p = 0,042	0,596 p = 0,069
M16	0,636 p = 0,048	0,067 p = 0,855	0,669 p = 0,035	0,612 p = 0,060
M17	0,604 p = 0,065	0,049 p = 0,894	0,639 p = 0,047	0,573 p = 0,083
M18	0,632 p = 0,050	0,067 p = 0,854	0,668 p = 0,035	0,596 p = 0,069
M19	0,389 p = 0,267	0,568 p = 0,087	0,399 p = 0,253	0,445 p = 0,198
M20	0,561 p = 0,092	-0,018 p = 0,960	0,606 p = 0,064	0,530 p = 0,115
M21	0,384 p = 0,273	0,512 p = 0,130	0,398 p = 0,255	0,427 p = 0,219

As in the case for Pearson's correlation coefficient, the correlation between the two variables is statistically significant if the p value is below 0.05 ($p < 0.05$). Table 14 summarizes the metrics which have statistically significant correlations with the fault-proneness measures of the software components with the strength of the correlation as low correlation, medium correlation and high correlation.

Table 14 - Statistically Significant Spearman's Correlations

	High Correlation	Medium Correlation	Low Correlation
Defect Count	M7, M12, M13, M16, M18	-	-
Defect Density	M4	-	-
Defect Severity	M5, M7, M9, M12, M13, M15, M16, M17, M18	-	-
Defect Correction Effort	M7, M13	-	-

Fewer metrics were correlated with fault-proneness measures of the software components by use of Spearman's correlation coefficient when compared with Pearson's correlation coefficient; however, the results were similar. For example; the metrics M5, M7, M9, M12, M15, M16, M17 and M18 were correlated strongly with defect severity of the software components according to both methods.

Two types of correlation coefficient were used in order to identify the association between design metrics and fault-proneness metrics. The first one is Pearson's correlation coefficient which measures the linear relationship between two variables, and the second one is Spearman's correlation coefficient which measures the monotonic relationship between two variables. As seen from Table 10, defect count, defect severity and defect correction effort metric measurements had an outlier and defect severity had non-normal distribution; this situation reduces the reliability to the results of Pearson's correlation coefficient. Also, monotonic relation, measured by Spearman's correlation coefficient, involves linear relation,

measured by Pearson’s correlation coefficient. Therefore, it can be concluded that results achieved with Spearman’s correlation coefficient are more reliable.

CHAPTER 5

DISCUSSION AND CONCLUSION

There are many studies in the literature which investigate the effects of structural characteristics of object-oriented software on software quality by use of metrics. A considerable part of these studies take fault tolerance of software as the quality indicator while collecting software design metrics from the source code of the software. These studies have verified a strong correlation between software design metrics and fault-proneness of software. However, rarity of the studies on the correlation of software metrics collected from UML models with fault-proneness has constituted the basic motivation of the present study.

In the present study, we aimed to prove the association between structural characteristics of object-oriented software measured in terms of UML metrics and software quality empirically by considering fault-proneness as quality indicator of software. For this purpose, we worked in Aselsan, a leading defense industry company in Turkey. The software design metrics and quality metrics were all collected from real-time embedded software components developed by YMM.

10 software components were investigated in the scope of the study. Software components used within at least two projects were selected; therefore, it was aimed that software components were tested adequately and they were reliable components. Software design metrics were collected from the UML models of the software components. Software metrics to be measured were selected after the related literature was reviewed. In addition to software metrics in the literature, three new metrics, NPI, NOPI and NEPI were proposed as interface metrics. Most of the metric measurements were carried out by constructing java plug-in applications for the UML design tool used by YMM, and ready-made metric measurement tools were used for other metrics. The defect data of the software

components were gathered from software defect tracking tool used by YMM. As fault-proneness measurement, four different types of measurement were used; namely, defect count, defect density, defect severity and defect correction effort; so, the extent of the analyses had been broadened in terms of fault-proneness.

Two methods were used in presenting the association between software design metrics and fault-proneness; namely, graphical and statistical analyses. Before the analysis, it was expected that all of the design metrics have positive correlation with the fault-proneness measures. For the cases defect count, defect severity and defect correction effort, design metrics showed positive association with the fault-proneness, as expected, at the end of the graphical analyses. However, when we looked the results for defect density, important part of the design metrics had negative correlation or did not have strong positive correlation with the fault-proneness. The reason behind this was interpreted as the fact that larger software components tend to have lower defect densities as suggested in [62]-[64].

As statistical analyses, Pearson's and Spearman's correlation coefficients were computed in order to assess the association between selected design metrics and fault-proneness of software. Expecting linear correlation between the design metrics and defect measures, Pearson's correlation coefficient was used, first. Although some defect data has outlier and non-normal distribution, it was continued with Pearson's correlation coefficient because it was decided to hold the outlier data in the analyses and some studies in the literature claimed that Pearson's correlation coefficient is somewhat robust to non-normality [61]. The results of the analyses were similar to graphical analyses. 12 of the 19 design metrics presented statistically significant correlations with the fault-proneness of software in the cases of defect count and defect severity, and 10 of the 19 design metrics showed similar results for defect correction effort. As for the graphical analyses, defect density results were weaker, in other words, the association was meaningful for only 2 of the design metrics. For Spearman's correlation coefficient, although fewer metrics

were able to be associated with defect measures of the software components as compared to Pearson's correlation coefficient, the results were similar.

The benefits of the study for Aselsan are twofold: The design metrics which were shown to be correlated with fault-proneness of software can be used to predict faulty software components. Thus, more testing effort can be spent, labor for peer review and design review can be increased and static code analyses may be attributed higher importance for the software components to be considered as error-prone, thereby improving quality achieved at release time. Also, structural characteristics that were shown to affect fault-proneness can be controlled via metric measurements in the development phase for the purpose of eliminating errors before they come up.

Limitations of the present study include the fact that only a limited number of components, all developed in the same organizational unit has been investigated. Also, software components analyzed in the scope of the present study were all real-time embedded software components. These situations restrict generalizability of our results severely. Software from other domains definitely deserves to be subjected to similar studies.

It would also be beneficial to carry out similar studies by extending the data set used. In other words, a higher number of software components should be analyzed. So, the effects of the structural characteristics of object-oriented software would be presented more explicitly. Moreover, it would be possible to achieve threshold values of the design metrics for software components with high quality in terms of fault-proneness.

Consequently, in this study, the association between some design metrics collected from UML models and fault-proneness of software was analyzed using real life project data in the context of real-time embedded software. Prior studies in the literature demonstrated the importance of code based metrics in identifying the fault-prone software components. However, the literature lacks researches which

studying on fault-proneness of software by use of UML metrics. Especially, rareness of the studies attracts attention for the statechart metrics; only a few studies analyzed the effect of statechart structural characteristics on the understandability, modifiability and maintainability [6], [7], [8]. To the best knowledge of the author of the present thesis, there is no other study which investigates the effects of statechart structural characteristics on the fault-proneness of software.

REFERENCES

[1] B. Curtis. (2012, December 4) *Non-functional Requirements Be Here* [Online]. Available: http://www.it-cisq.org/non-functional-requirements-be-here, Access date: 12/11/2012

[2] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, pp. 476-493, 1994.

[3] T. H. A. Soliman, A. El-Swesy and S. H. Ahmed, "Utilizing CK metrics suite to UML models: A case study of microarray MIDAS software," in *Informatics and Systems (INFOS), 2010 the 7th International Conference on*, 2010, pp. 1-6.

[4] N. Debnath, D. Riesco, G. Montejano, R. Uzal, L. Baigorria, A. Dasso and A. Funes, "A technique based on the OMG metamodel and OCL for the definition of object-oriented metrics applied to UML models," in *Computer Systems and Applications, 2005. the 3rd ACS/IEEE International Conference on*, 2005, pp. 118.

[5] M. Genero, M. Piattini and C. Calero, "Empirical validation of class diagram metrics," in *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*, 2002, pp. 195-203.

[6] J. A. Cruz-Lemus, A. Maes, M. Genero, G. Poels and M. Piattini, "The impact of structural complexity on the understandability of UML statechart diagrams," *Inf. Sci.*, vol. 180, pp. 2209-2220, 2010.

[7] J. A. Cruz-Lemus, M. Genero, S. Morasca and M. Piattini, "Using practitioners for assessing the understandability of UML statechart diagrams with composite states," in *Advances in Conceptual Modeling–Foundations and Applications*Anonymous Springer, 2007, pp. 213-222.

[8] J. A. Cruz-Lemus, M. Genero, M. E. Manso, S. Morasca and M. Piattini, "Assessing the understandability of UML statechart diagrams with composite states—A family of empirical studies," *Empirical Software Engineering*, vol. 14, pp. 685-719, 2009.

[9] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, pp. 308-320, 1976.

[10] B. P. Douglass, “Computing model complexity”, White paper, I-Logix, 2004.

[11] Object Management Group, Inc. (2013), *Object Management Group - UML*. Available: http://www.uml.org, Access Date: 23/02/2013

[12] H. Jung, S. Kim and C. Chung, "Measuring software product quality: A survey of ISO/IEC 9126," *IEEE Software*, vol. 21, pp. 88-92, 2004.

[13] ISO/IEC 9126 – Software and System Engineering – Product quality – Part 1: Quality model. 1999-2002

[14] ISO/IEC 9126 – Software and System Engineering – Product quality – Part 2: External Quality Metrics. 1999-2002

[15] ISO/IEC 9126 – Software and System Engineering – Product quality – Part 3: Internal Quality Metrics. 1999-2002

[16] ISO/IEC 9126 – Software and System Engineering – Product quality – Part 4: Quality in Use Metrics. 1999-2002

[17] B. Behkamal, M. Kahani and M. K. Akbari, "Customizing ISO 9126 quality model for evaluation of B2B applications," *Information and Software Technology*, vol. 51, pp. 599-609, 2009.

[18] J. Boegh, "Certifying software component attributes," *IEEE Software*, vol. 23, pp. 74-81, 2006.

[19] H. Al-Kilidar, K. Cox and B. Kitchenham, "The use and usefulness of the ISO/IEC 9126 quality standard," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, 2005, pp. 7 pp.

[20] A. Abran, R. E. Al-Qutaish and J. J. Cuadrado-Gallego, "Analysis of the ISO 9126 on software product quality evaluation from the metrology and ISO 15939 perspectives." *WSEAS Transactions on Computers*, vol. 5, pp. 2778-2786, 2006.

[21] ISO/IEC FCD 25010 Software engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Quality model and guide, 2009.

[22] C. F. Lange and M. R. Chaudron, "Managing model quality in UML-based software development," in *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, 2005, pp. 7-16.

[23] F. Brito e Abreu, "The MOOD Metrics Set," *Proc. ECOOP'95 Workshop on Metrics*, 1995.

[24] M. Lorenz and J. Kidd, "Object-Oriented Software Metrics," *Ed.Prentice-Hall, Englewood Cliffs, New Jersey*, 1994.

[25] C. Lange and M. Chaudron, "An empirical assessment of completeness in UML designs," 2004.

[26] C. F. Lange and M. R. Chaudron, "Effects of defects in UML models: An experimental investigation," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 401-411.

[27] S. N. Bhatti, "Why quality?: ISO 9126 software quality metrics (Functionality) support by UML suite," *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1-5, 2005.

[28] J. Wüst. The software design metrics tool for the UML, version 2.3. Available: http://www.sdmetrics.com, Access Date: 15.03.2013

[29] A. Mayr, R. Plosch, M. Klas, C. Lampasona and M. Saft, "A comprehensive code-based quality model for embedded systems: Systematic development and validation by industrial projects," in *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, 2012, pp. 281-290.

[30] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb and J. Streit, "The quamoco product quality modelling and assessment approach," in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 1133-1142.

[31] S. H. Kan, *Metrics and Models in Software Quality Engineering.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[32] M. Hneif and S. P. Lee, "Using Guidelines to Improve Quality in Software Nonfunctional Attributes," *Software, IEEE*, vol. 28, pp. 72-77, 2011.

[33] L. B. Phillips, A. Aurum and R. B. Svensson, "Managing software quality requirements," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, 2012, pp. 349-356.

[34] R. Lincke, T. Gutzmann and W. Löwe, "Software quality prediction models compared," in *Quality Software (QSIC), 2010 10th International Conference on*, 2010, pp. 82-91.

[35] T. Gyimothy, R. Ferenc and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *Software Engineering, IEEE Transactions on*, vol. 31, pp. 897-910, 2005.

[36] P. Yu, T. Systa and H. Muller, "Predicting fault-proneness using OO metrics. an industrial case study," in *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*, 2002, pp. 99-107.

[37] L. C. Briand, C. Bunse and J. W. Daly, "A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs," *Software Engineering, IEEE Transactions on*, vol. 27, pp. 513-530, 2001.

[38] M. Klas, C. Lampasona and J. Munch, "Adapting software quality models: Practical challenges, approach, and first empirical results," in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, 2011, pp. 341-348.

[39] L. Wang, "Component-based performance-sensitive real-time embedded software," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 23, pp. 28-34, 2008.

[40] A. Kaur, P. S. Sandhu and A. S. Bra, "Early software fault prediction using real time defect data," in *Machine Vision, 2009. ICMV'09. Second International Conference on*, 2009, pp. 242-245.

[41] A. Nugroho, M. R. Chaudron and E. Arisholm, "Assessing uml design metrics for predicting fault-prone classes in a java system," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, 2010, pp. 21-30.

[42] B. Zhang and X. Shen, "The effectiveness of real-time embedded software testing," in *Reliability, Maintainability and Safety (ICRMS), 2011 9th International Conference on*, 2011, pp. 661-664.

[43] B. Deniz, "Investigation of the effects of reuse on software quality in an industrial setting", M.S. thesis, Dept. Elect. & Electron. Eng., METU, Ankara, Turkey, 2013.

[44] T. Turk, "The effects of software design patterns on object-oriented software quality and maintainability", M. S. thesis, Dept. Elect. & Electron. Eng., METU, Ankara, Turkey, 2009.

[45] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, pp. 297-310, 2003.

[46] S. Srivastava and R. Kumar, "Indirect method to measure software quality using CK-OO suite," in *Intelligent Systems and Signal Processing (ISSP), 2013 International Conference on*, 2013, pp. 47-51.

[47] R. Selvarani, T. G. Nair and V. K. Prasad, "Estimation of defect proneness using design complexity measurements in object-oriented software," in *2009 International Conference on Signal Processing Systems*, 2009, pp. 766-770.

[48] S. Wagner and J. Jürjens, "Model-based identification of fault-prone components," in *Dependable Computing-EDCC 5*Anonymous Springer, 2005, pp. 435-452.

[49] E. Kahraman, T. İpek, B. İyidir, C. F. Bazlamaçcı, and S. 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] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin and M. Huh, "FORM: A feature-oriented reuse method with domain-; specific reference architectures," *Annals of Software Engineering*, vol. 5, pp. 143-168, 1998.

[51] M. Genero, M. Piattini and C. Calero, *Metrics for Software Conceptual Models.* Imperial College Press London, 2005.

[52] J. A. McQuillan and J. F. Power, "On the application of software metrics to UML models," in *Models in Software Engineering*Anonymous Springer, 2007, pp. 217-226.

[53] J. Blunck. Cyclomatic Complexity Calculater. Available: http://www.blunck.info/ccm.html, Access Date: 26/11/2013

[54] E. S. Cho, M. S. Kim and S. D. Kim, "Component metrics to measure component quality," in *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, 2001, pp. 419-426.

[55] T. Vernazza, G. Granatella, G. Succi, L. Benedicenti and M. Mintchev, "Defining metrics for software components," in *Proceedings of. World Multi-Conference on Systematics, Cybernetics and Informatics*, 2000, pp. 16-23.

[56] L. C. Briand, S. Morasca and V. R. Basili, "Property-based software engineering measurement," *Software Engineering, IEEE Transactions on*, vol. 22, pp. 68-86, 1996.

[57] T. D. Oyetoyan, R. Conradi and D. S. Cruzes, "A comparison of different defect measures to identify defect-prone components," in *Software Measurement and the 2013 Eighth International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2013 Joint Conference of the 23rd International Workshop on*, 2013, pp. 181-190.

[58] L. Li and H. Leung, "Using the number of faults to improve fault-proneness prediction of the probability models," in *Computer Science and Information Engineering, 2009 WRI World Congress on*, 2009, pp. 722-726.

[59] Y. K. Malaiya and J. Denton, "Estimating the number of residual defects [in software]," in *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, 1998, pp. 98-105.

[60] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences (Rev.* Lawrence Erlbaum Associates, Inc, 1977.

[61] C. J. Kowalski, "On the effects of non-normality on the distribution of the sample product-moment correlation coefficient," *Applied Statistics,* pp. 1-12, 1972.

[62] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ACM SIGSOFT Software Engineering Notes,* 2002, pp. 55-64.

[63] L. Hatton, "Reexamining the fault density-component size connection," *IEEE Software,* vol. 14, pp. 89-97, 1997.

[64] K. Moller and D. J. Paulish, "An empirical investigation of software fault distribution," in *Software Metrics Symposium, 1993. Proceedings., First International,* 1993, pp. 82-90.

APPENDIX

GRAPHS SHOWING THE CORRELATION BETWEEN DESIGN METRICS AND FAULT-PRONENESS METRICS

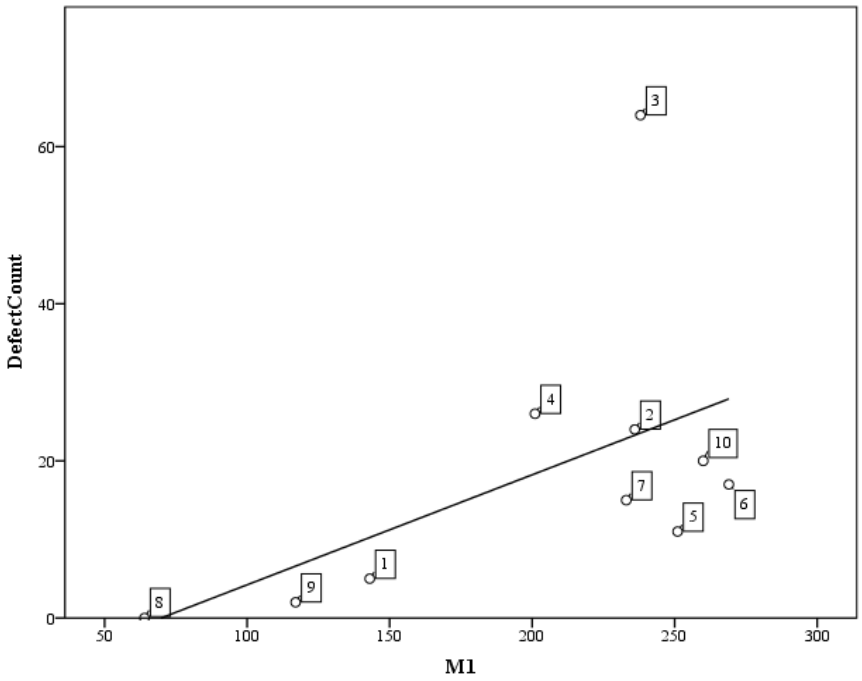

Figure 1 - Correlation between Defect Count and M1

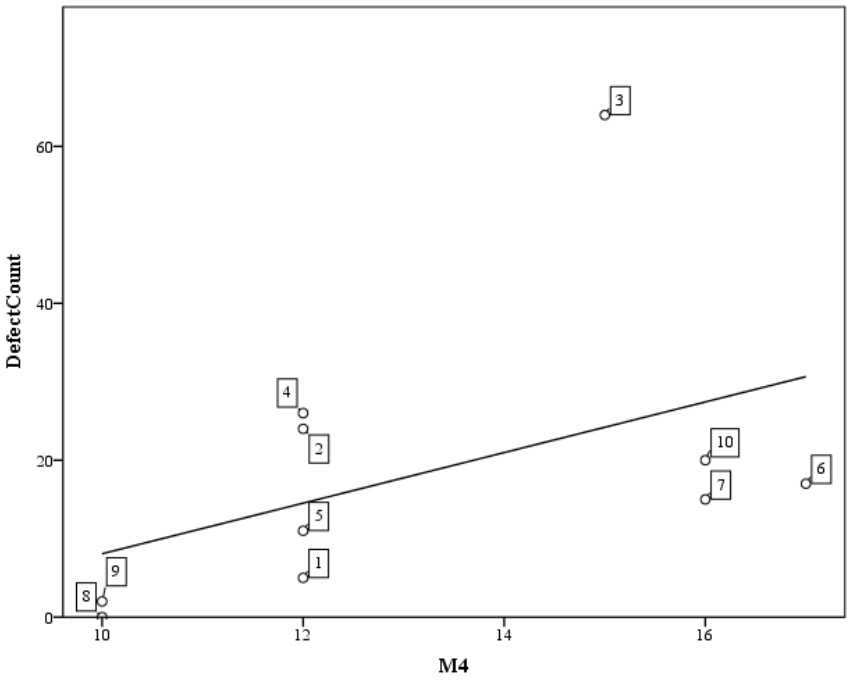

Figure 2 - Correlation between Defect Count and M4

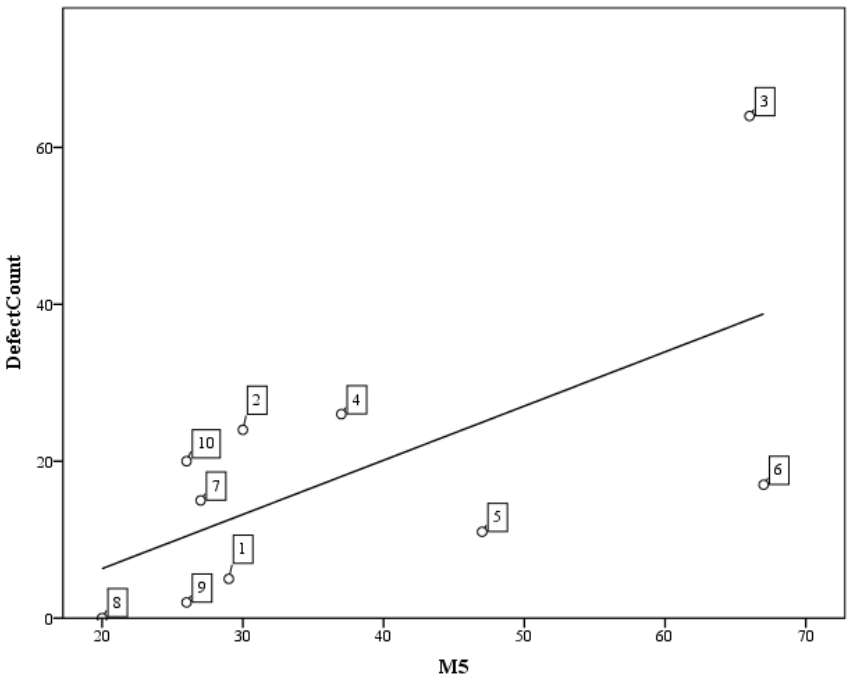

Figure 3 - Correlation between Defect Count and M5

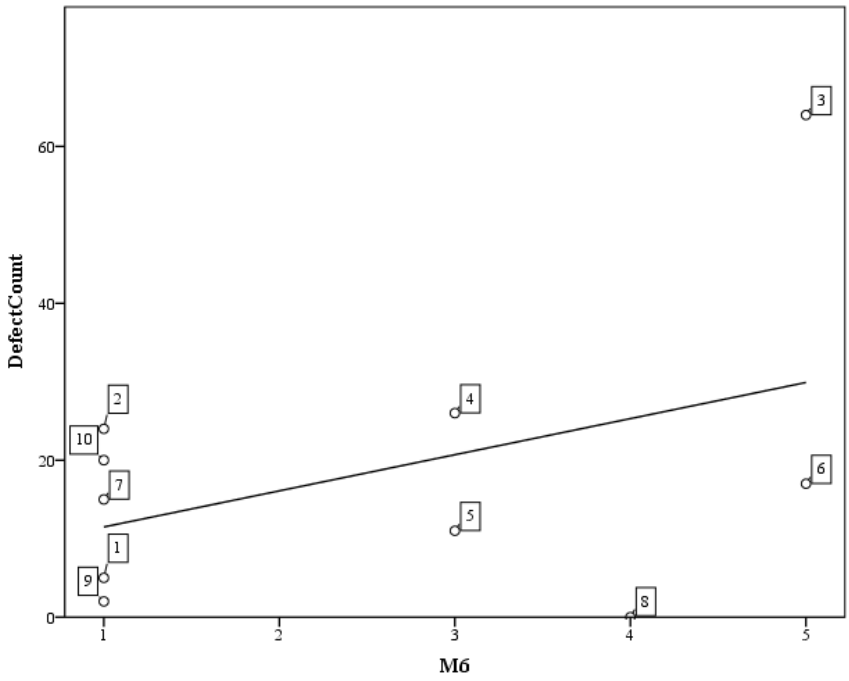

Figure 4 - Correlation between Defect Count and M6

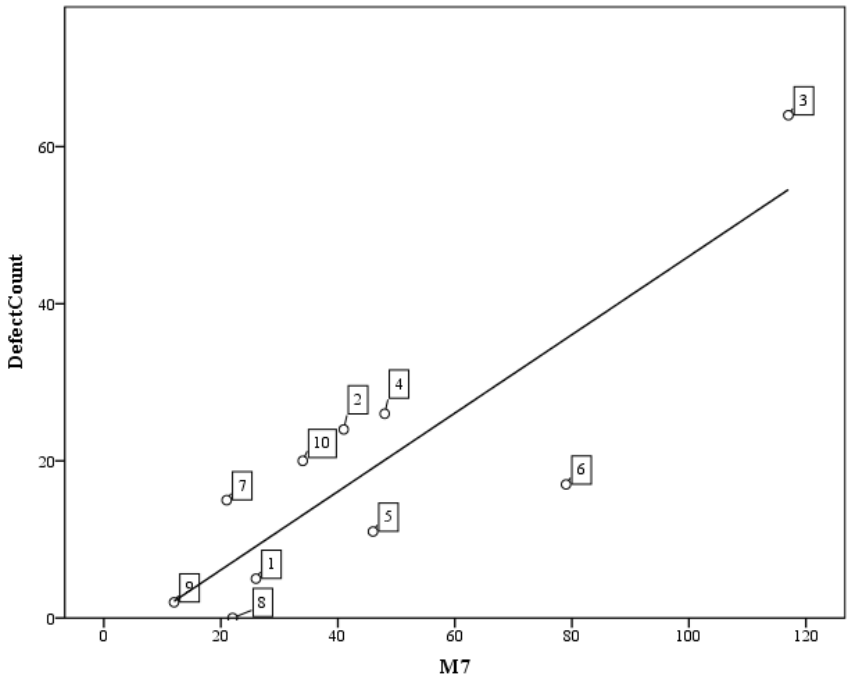

Figure 5 - Correlation between Defect Count and M7

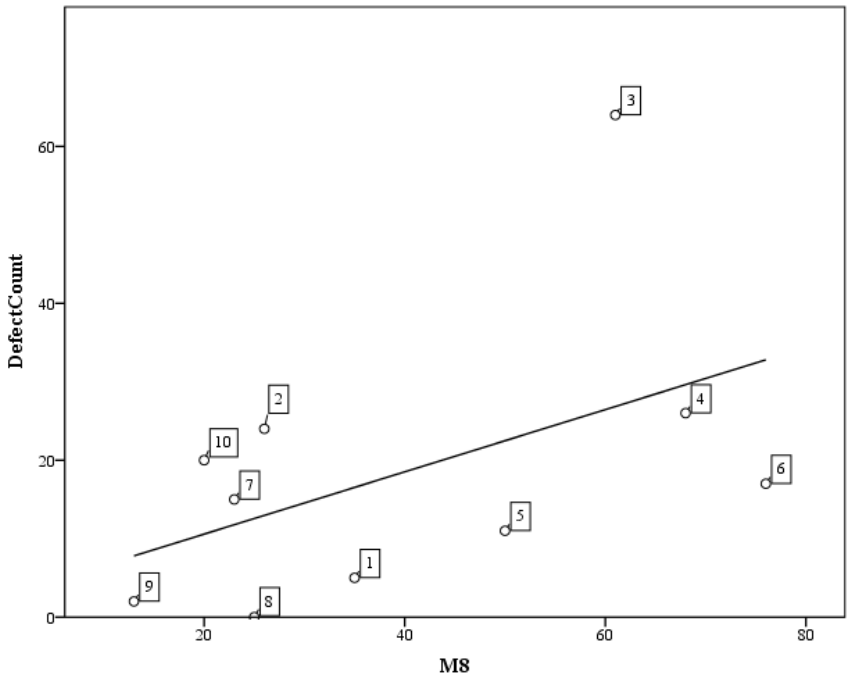

Figure 6 - Correlation between Defect Count and M8

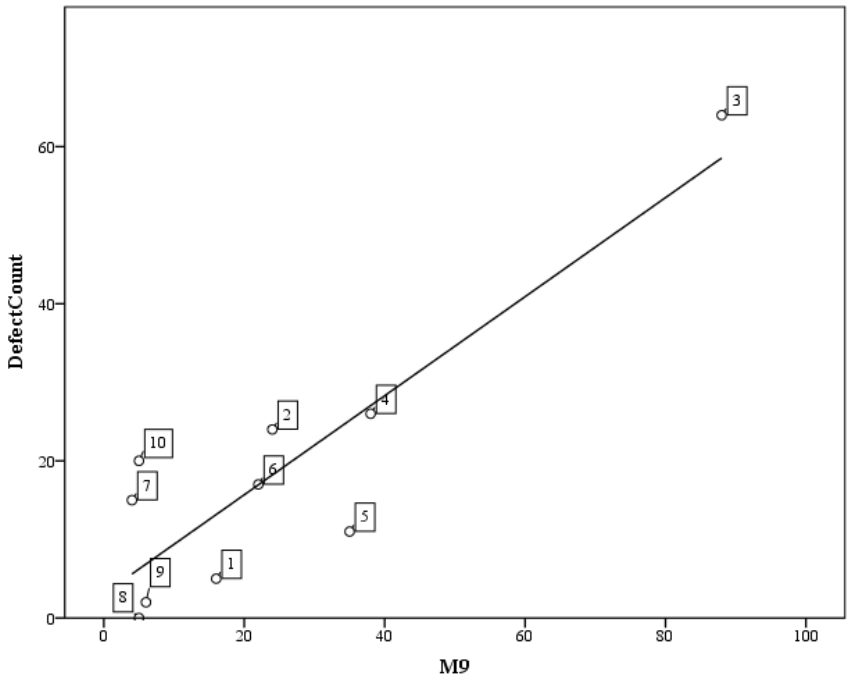

Figure 7 - Correlation between Defect Count and M9

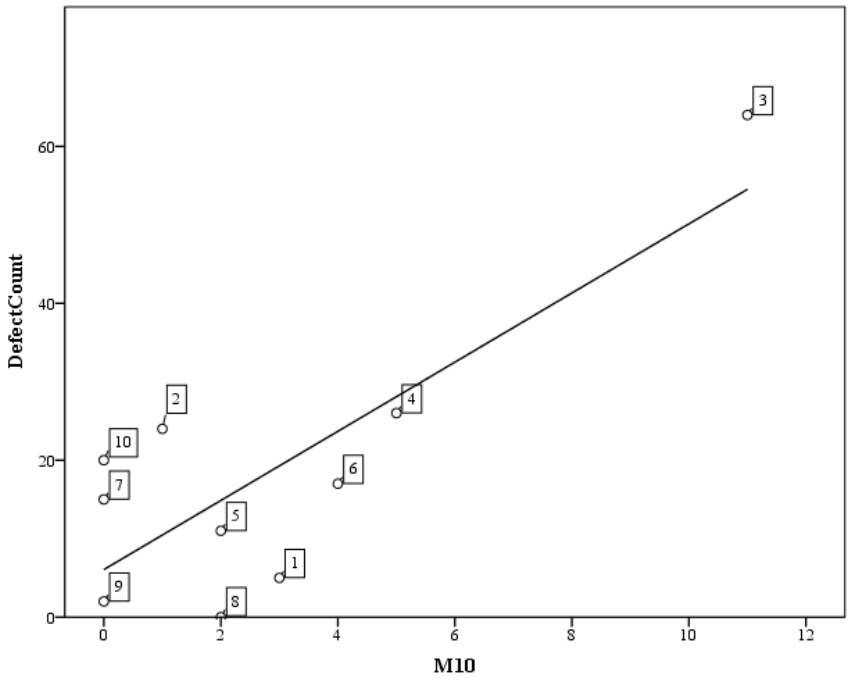

Figure 8 - Correlation between Defect Count and M10

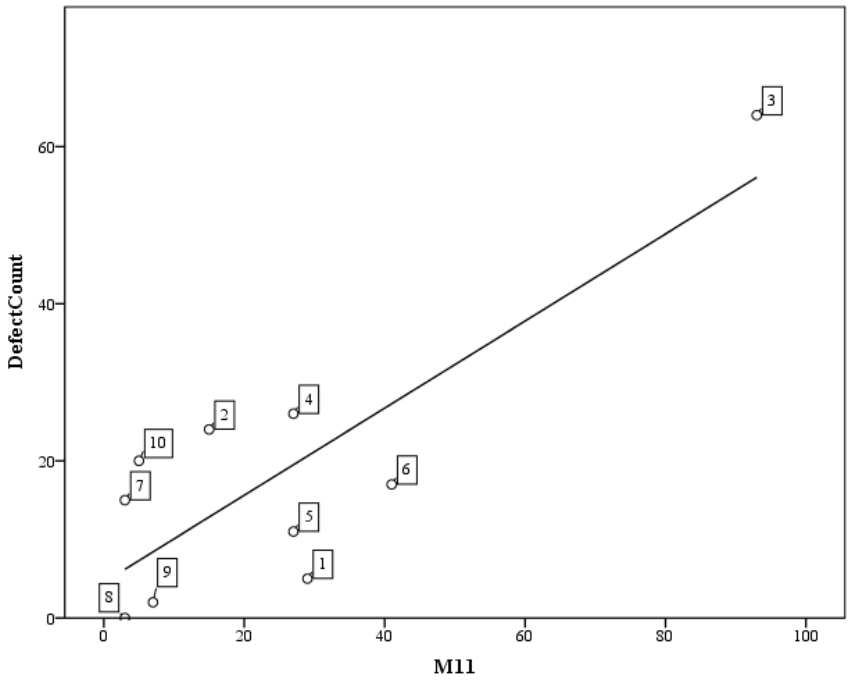

Figure 9 - Correlation between Defect Count and M11

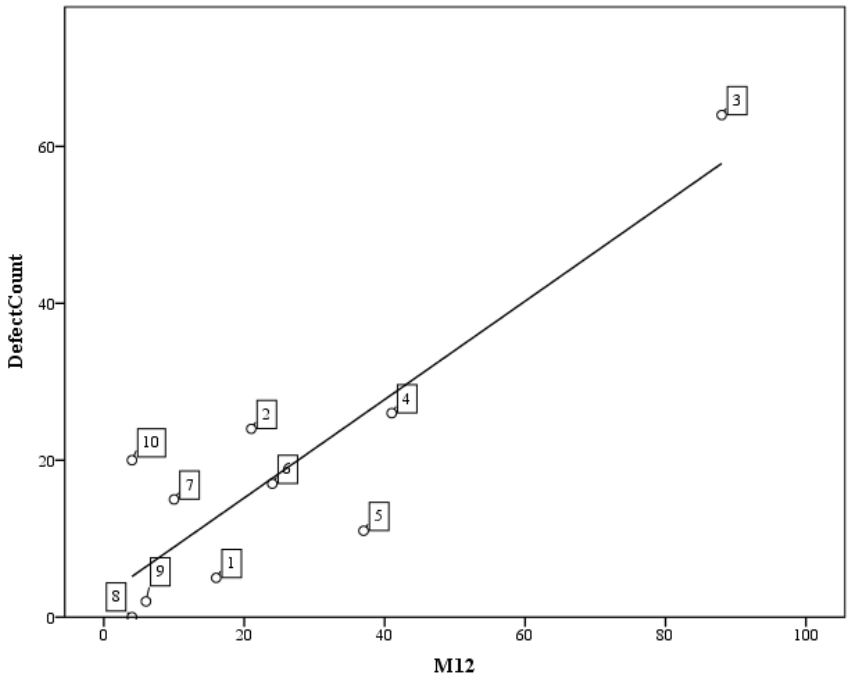

Figure 10 - Correlation between Defect Count and M12

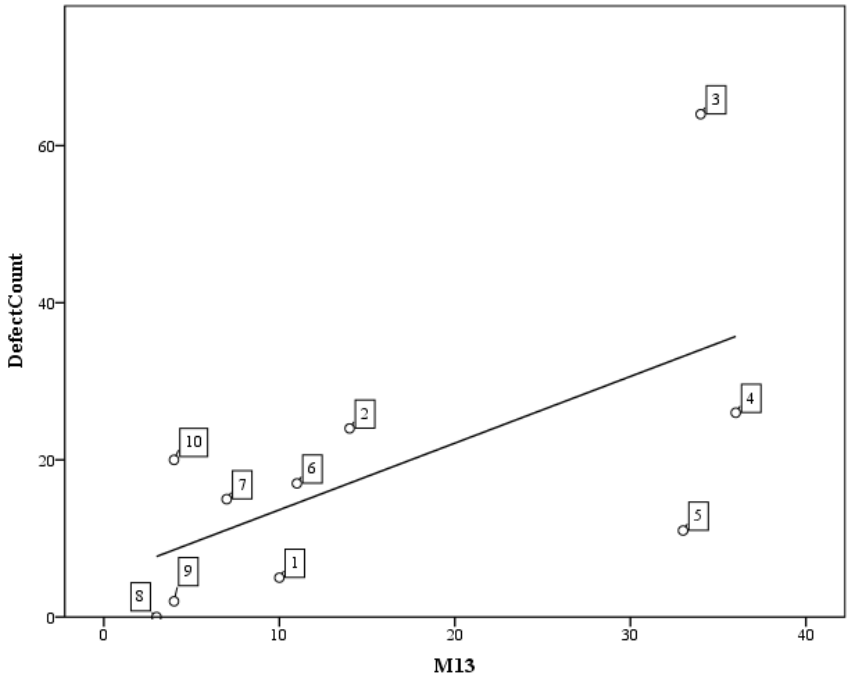

Figure 11 - Correlation between Defect Count and M13

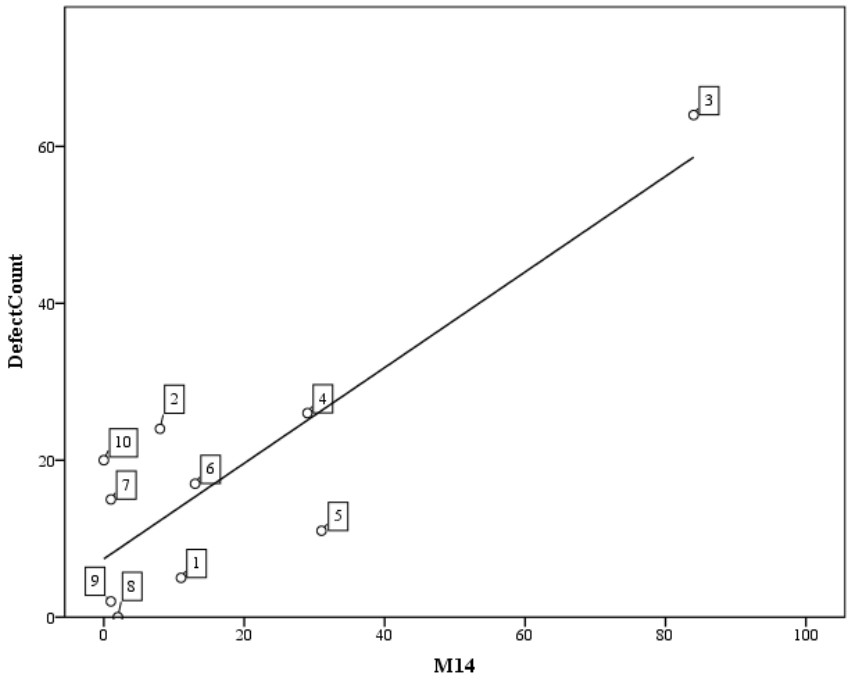

Figure 12 - Correlation between Defect Count and M14

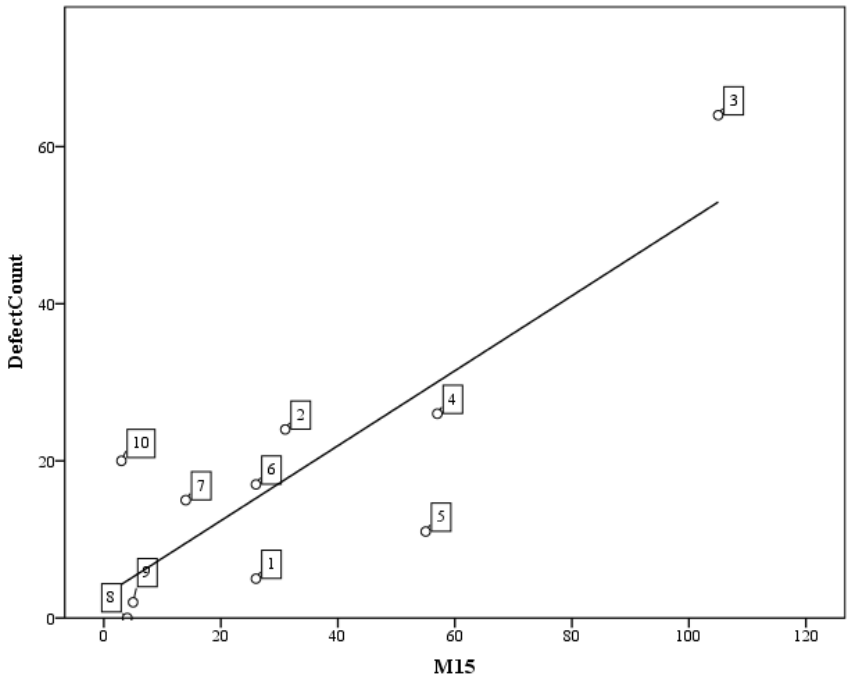

Figure 13 - Correlation between Defect Count and M15

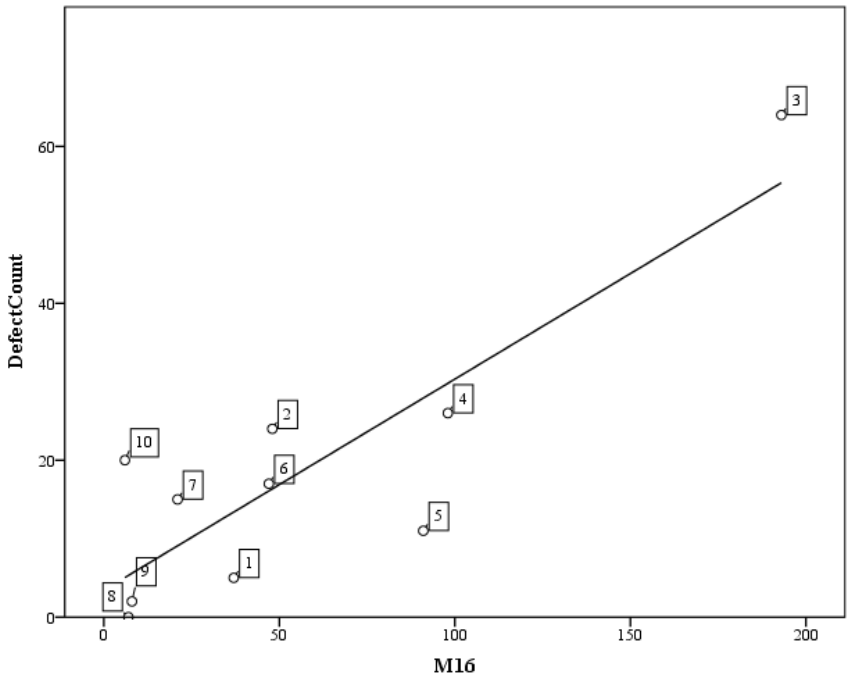

Figure 14 - Correlation between Defect Count and M16

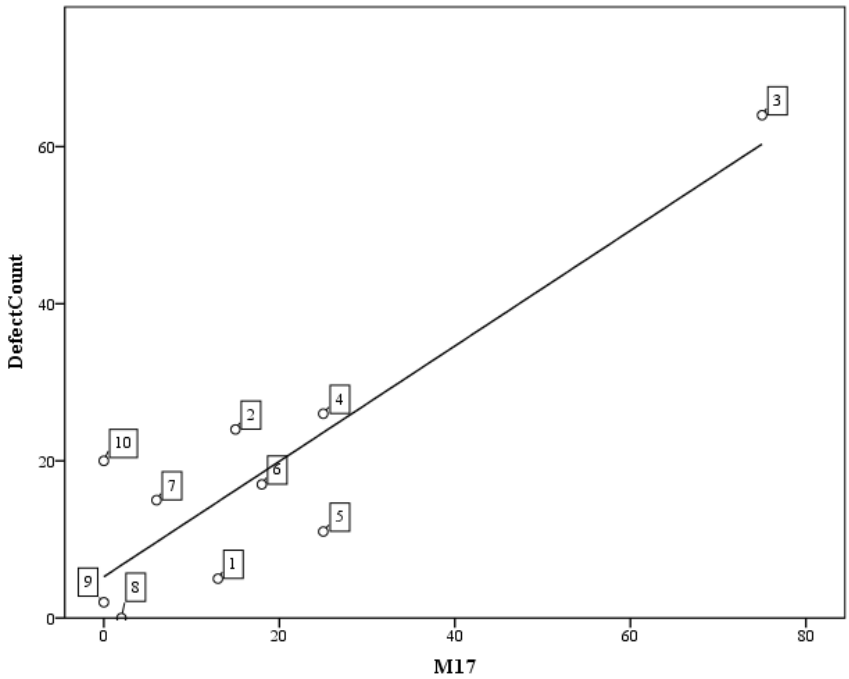

Figure 15 - Correlation between Defect Count and M17

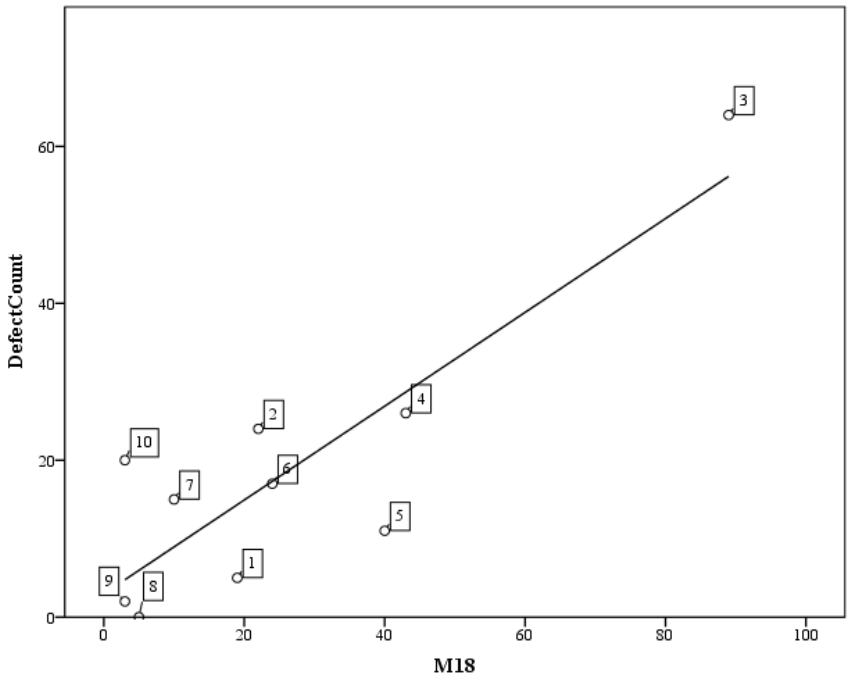

Figure 16 - Correlation between Defect Count and M18

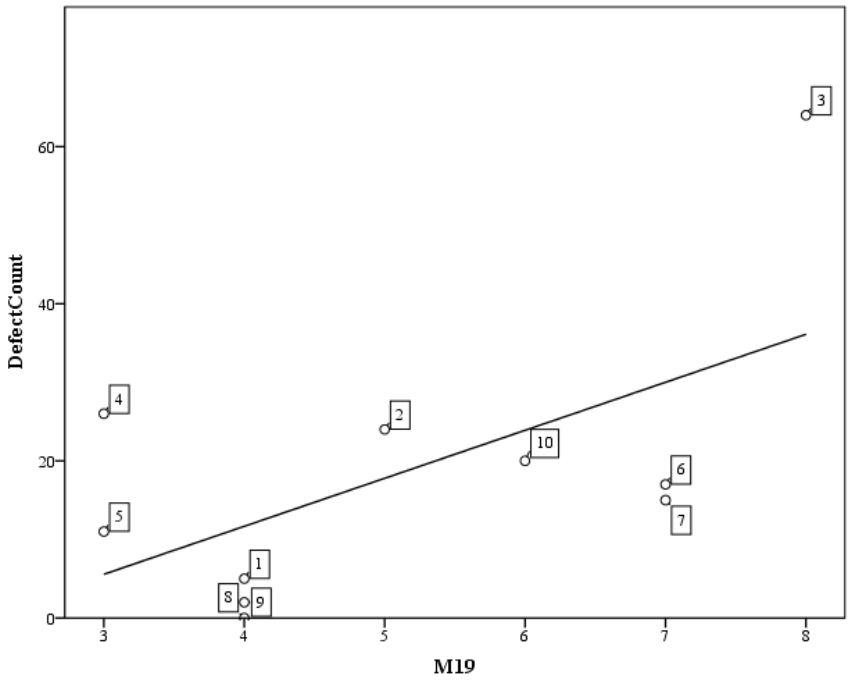

Figure 17 - Correlation between Defect Count and M19

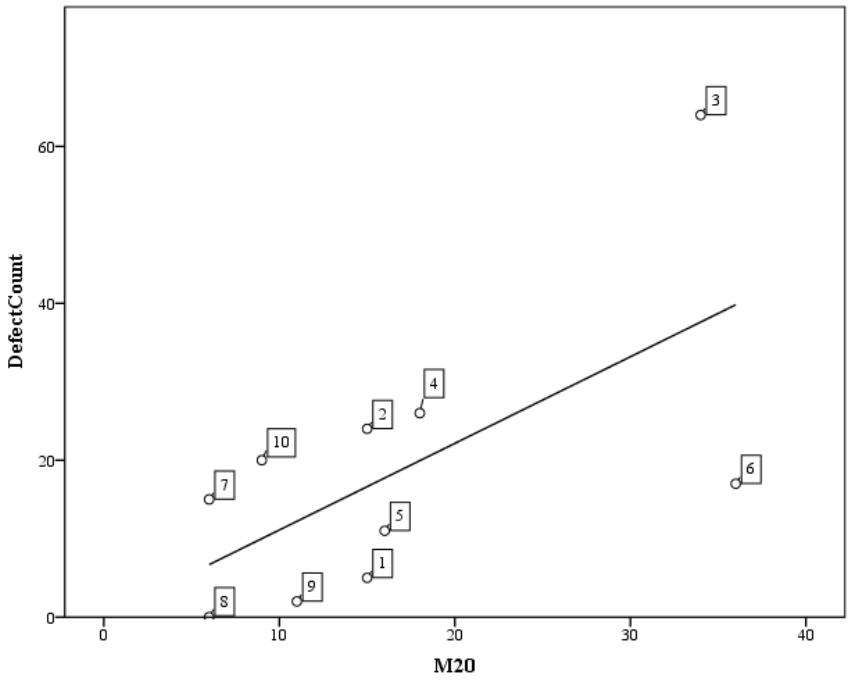

Figure 18 - Correlation between Defect Count and M20

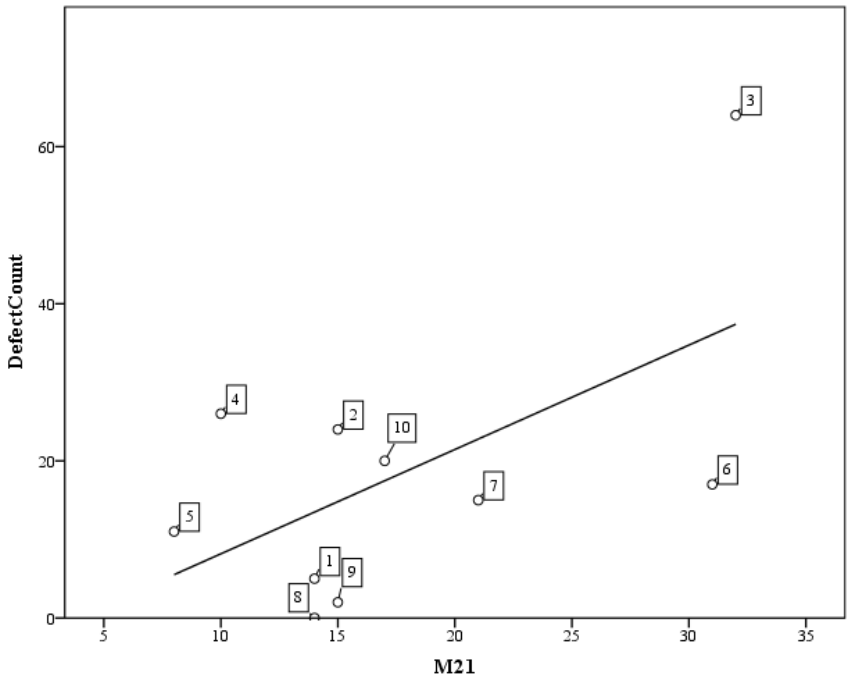

Figure 19 - Correlation between Defect Count and M21

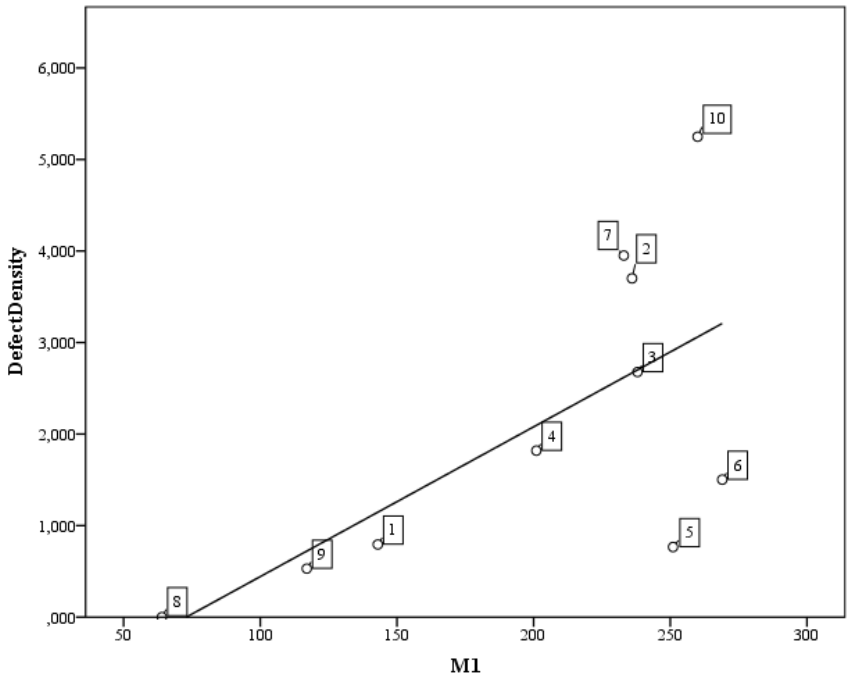

Figure 20 - Correlation between Defect Density and M1

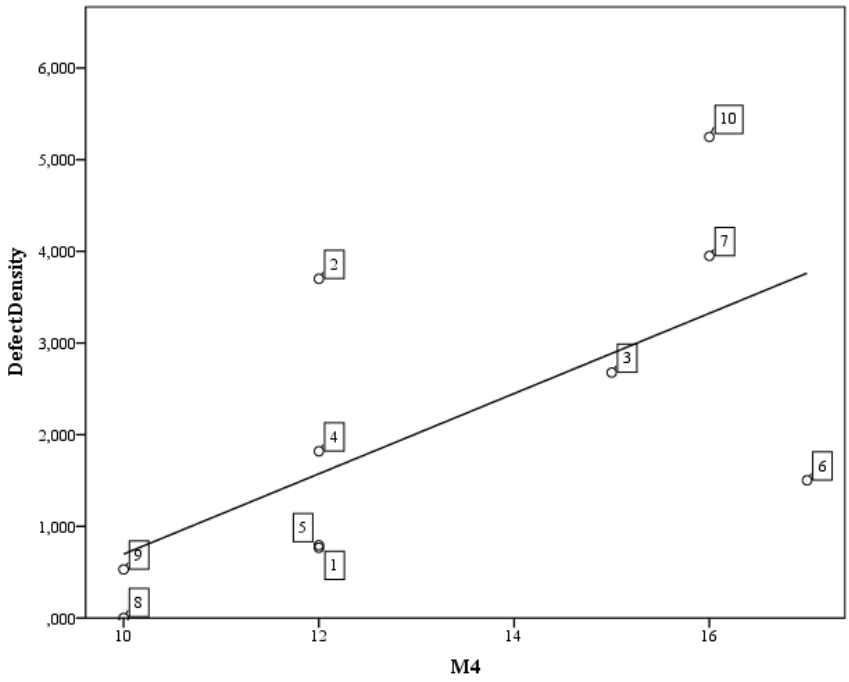

Figure 21- Correlation between Defect Density and M4

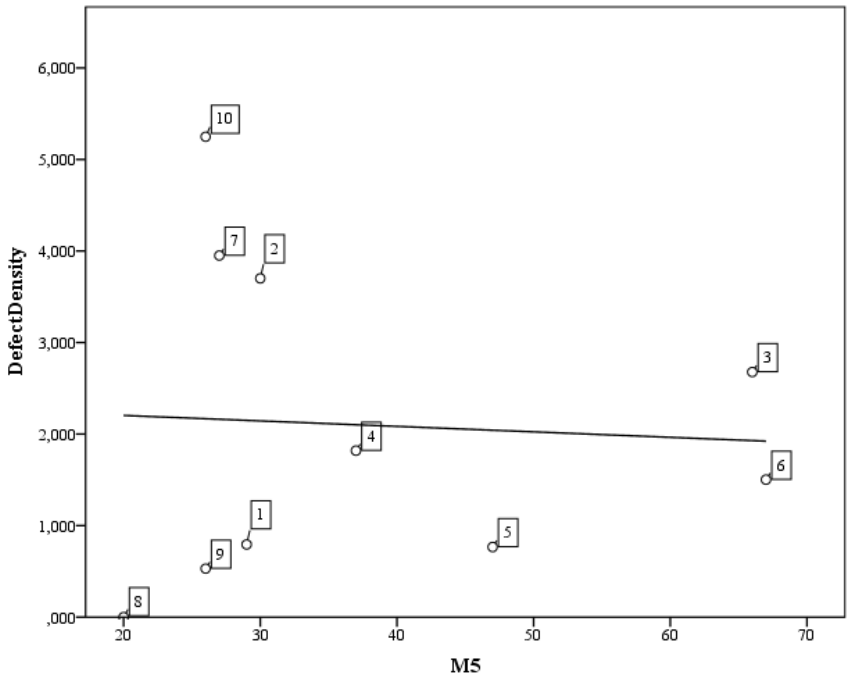

Figure 22 - Correlation between Defect Density and M5

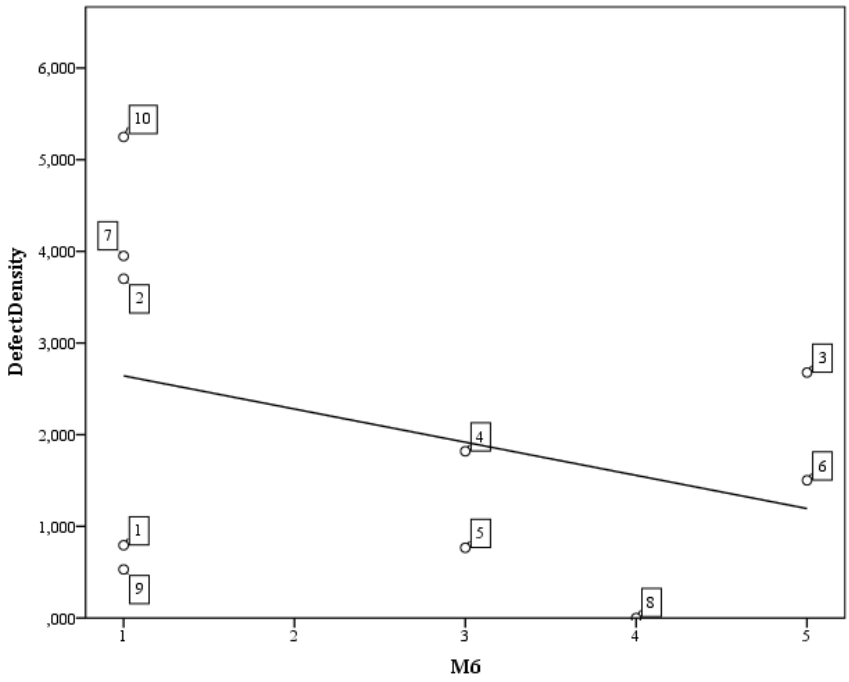

Figure 23 - Correlation between Defect Density and M6

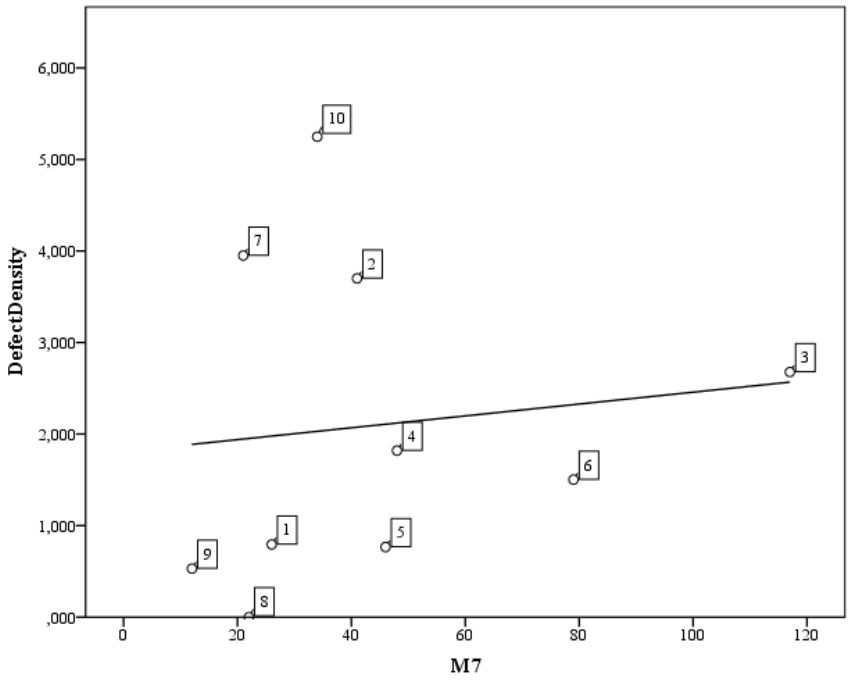

Figure 24 - Correlation between Defect Density and M7

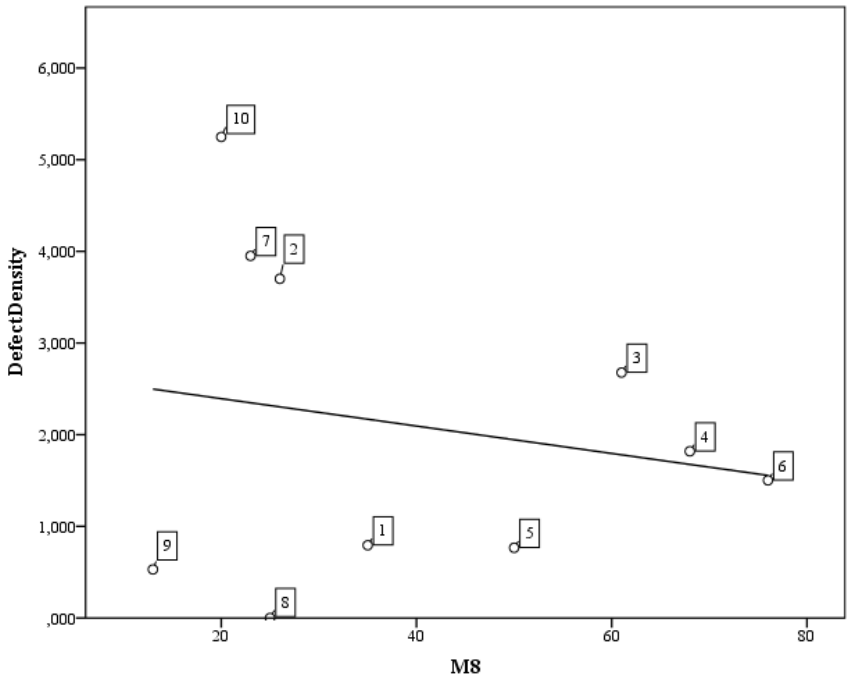

Figure 25 - Correlation between Defect Density and M8

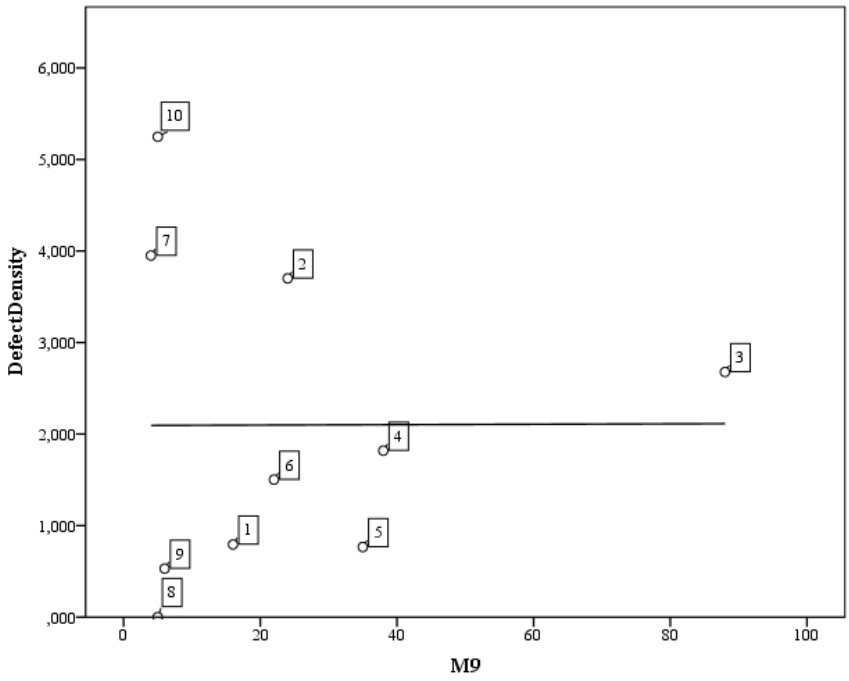

Figure 26 - Correlation between Defect Density and M9

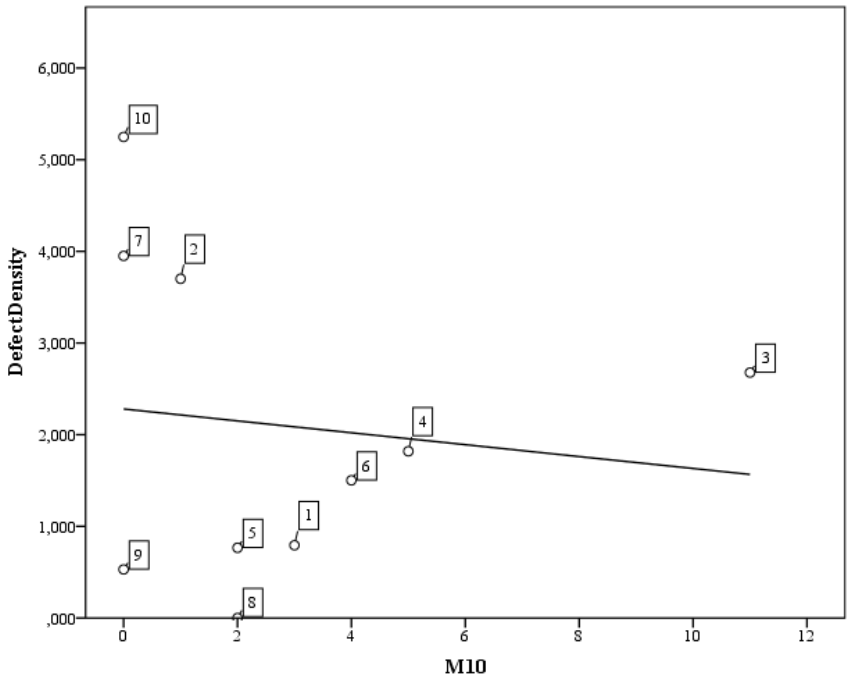

Figure 27 - Correlation between Defect Density and M10

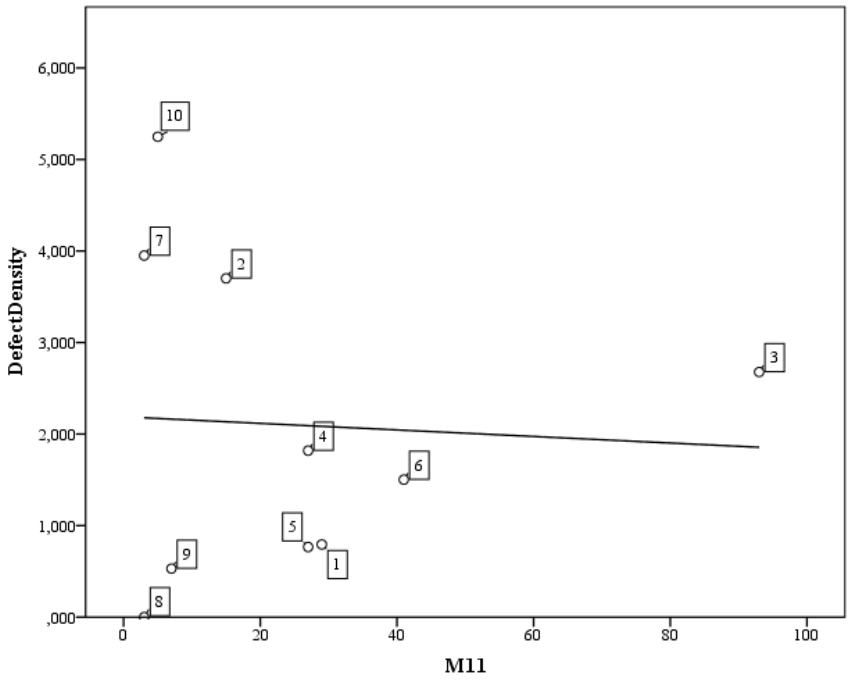

Figure 28 - Correlation between Defect Density and M11

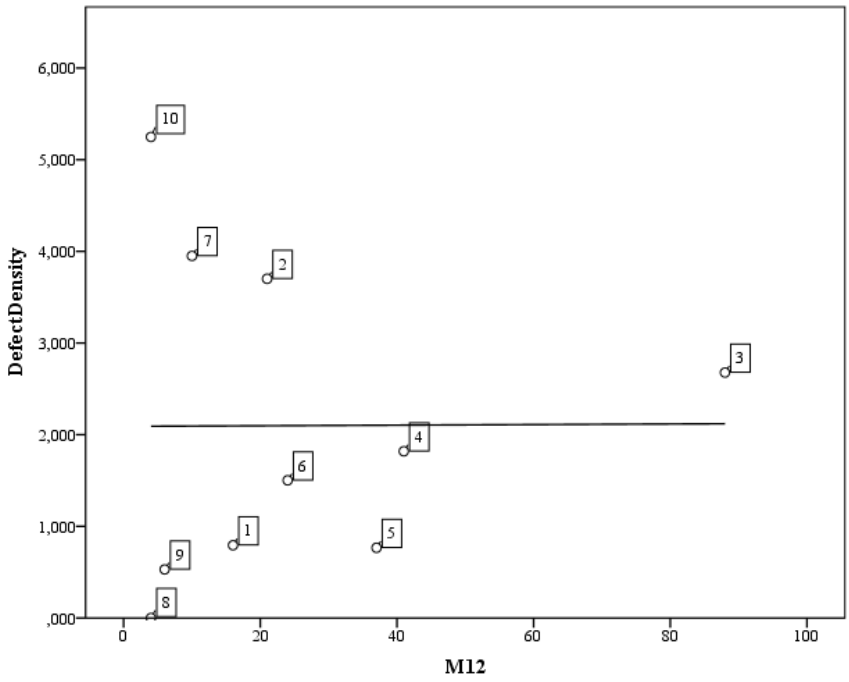

Figure 29 - Correlation between Defect Density and M12

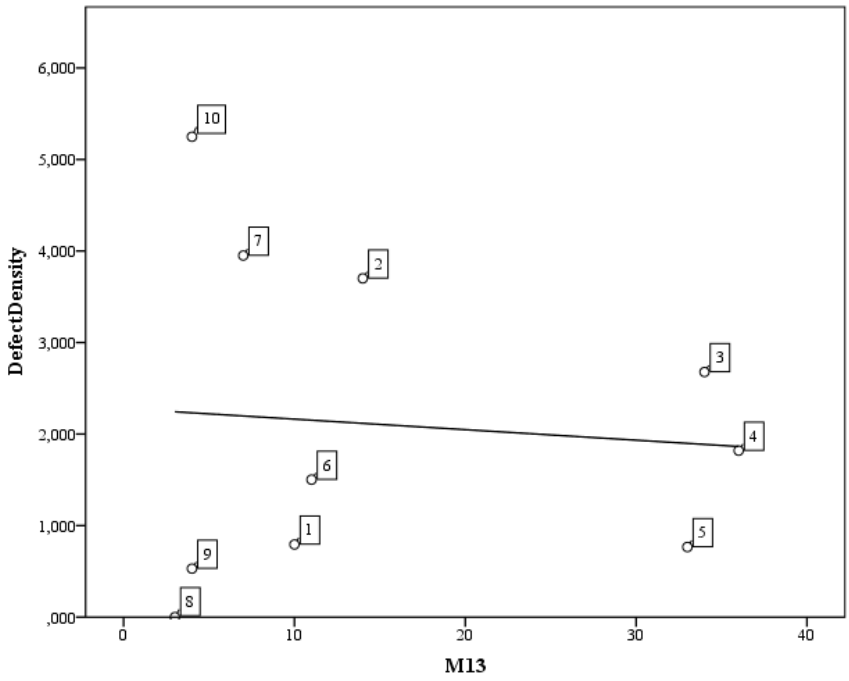

Figure 30 - Correlation between Defect Density and M13

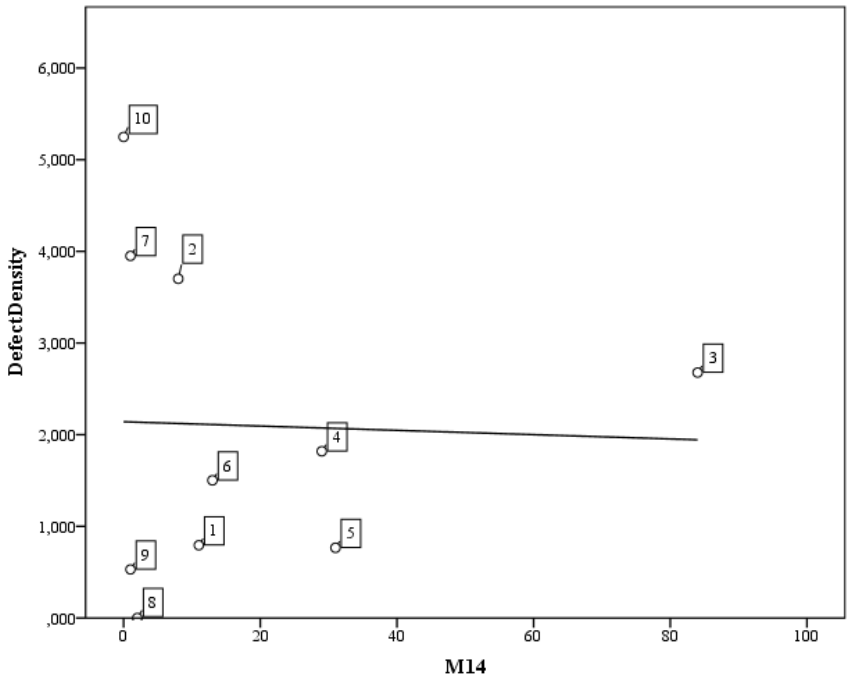

Figure 31 - Correlation between Defect Density and M14

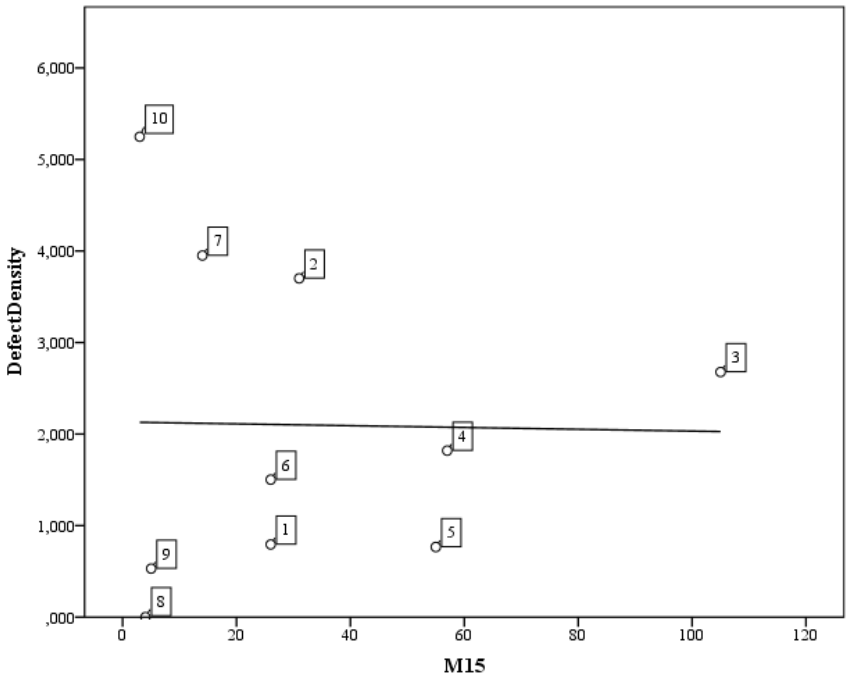

Figure 32 - Correlation between Defect Density and M15

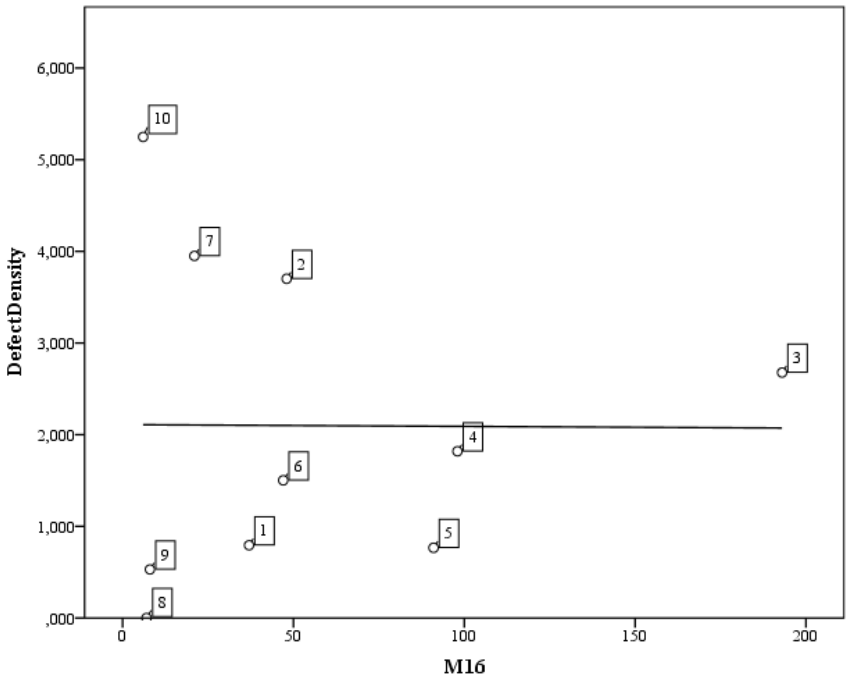

Figure 33 - Correlation between Defect Density and M16

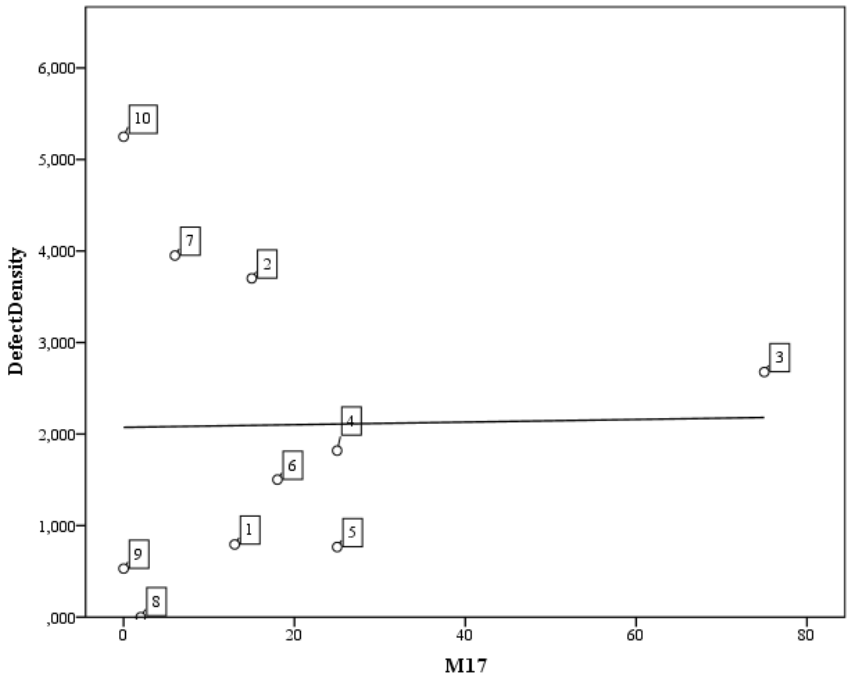

Figure 34 - Correlation between Defect Density and M17

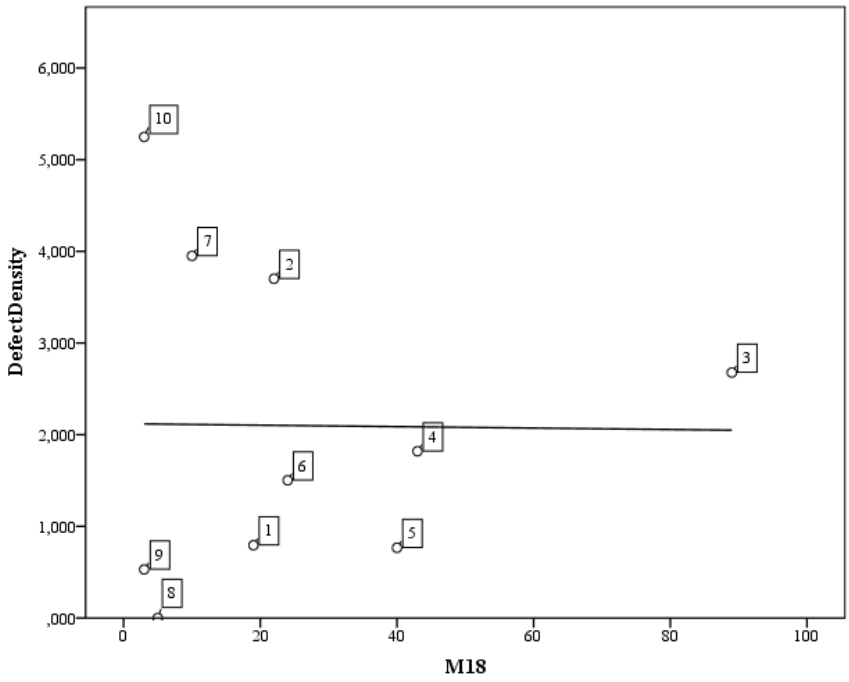

Figure 35 - Correlation between Defect Density and M18

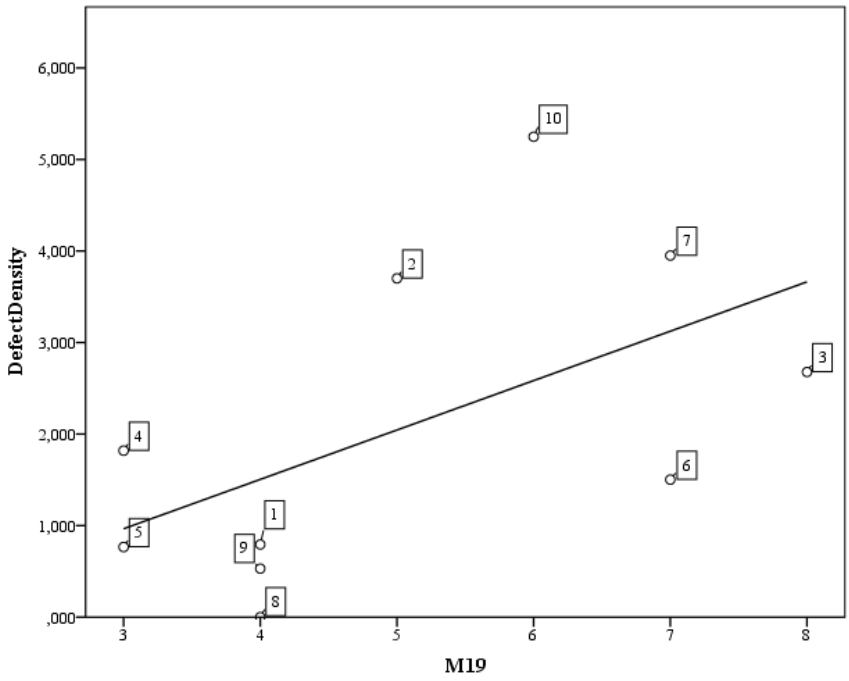

Figure 36 - Correlation between Defect Density and M19

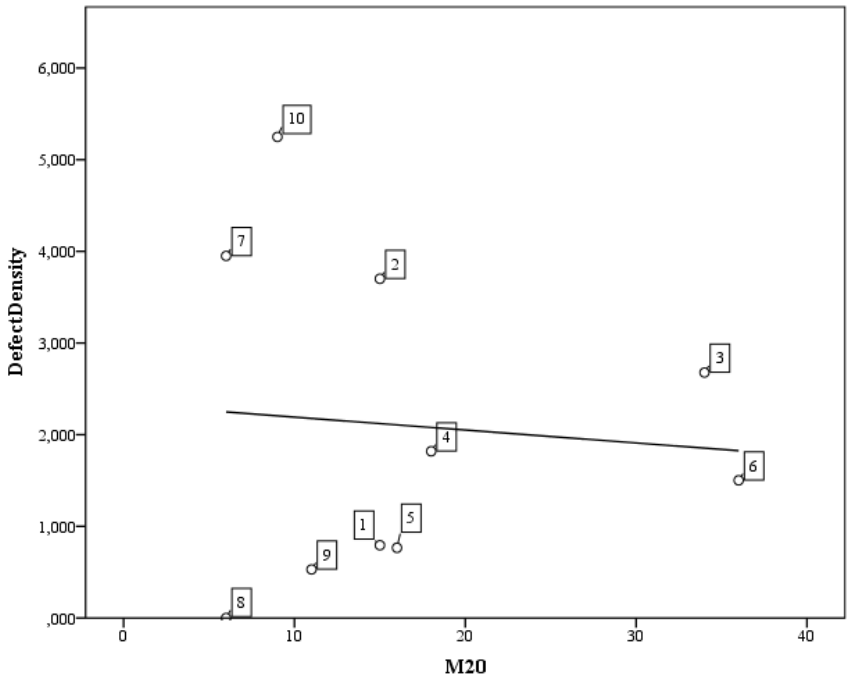

Figure 37 - Correlation between Defect Density and M20

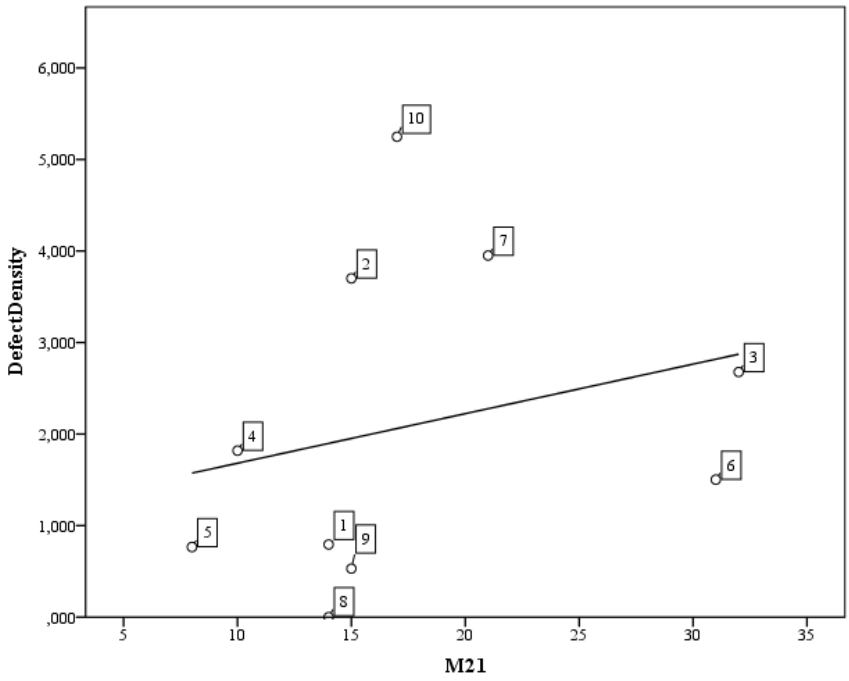

Figure 38 - Correlation between Defect Density and M21

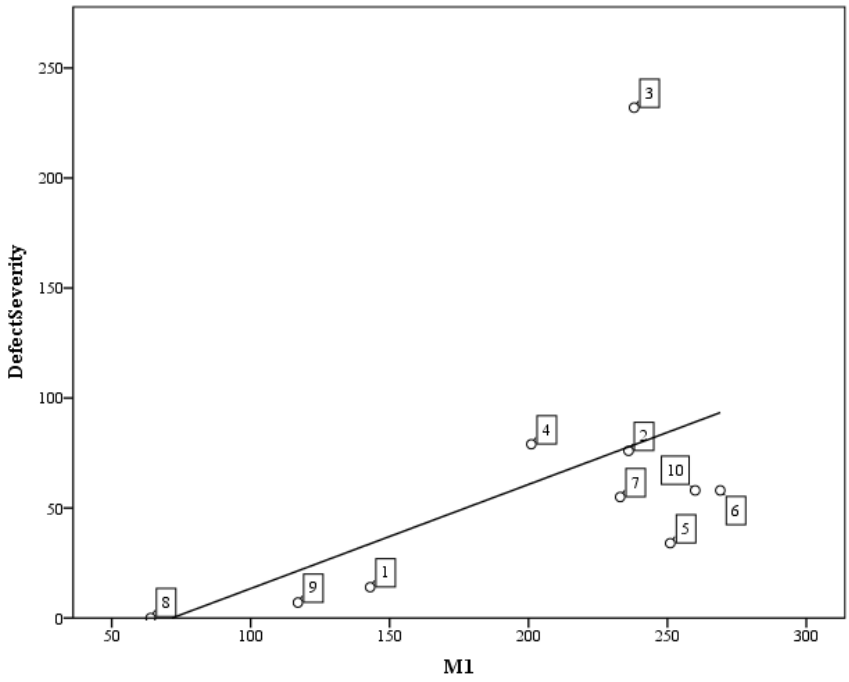

Figure 39 - Correlation between Defect Severity and M1

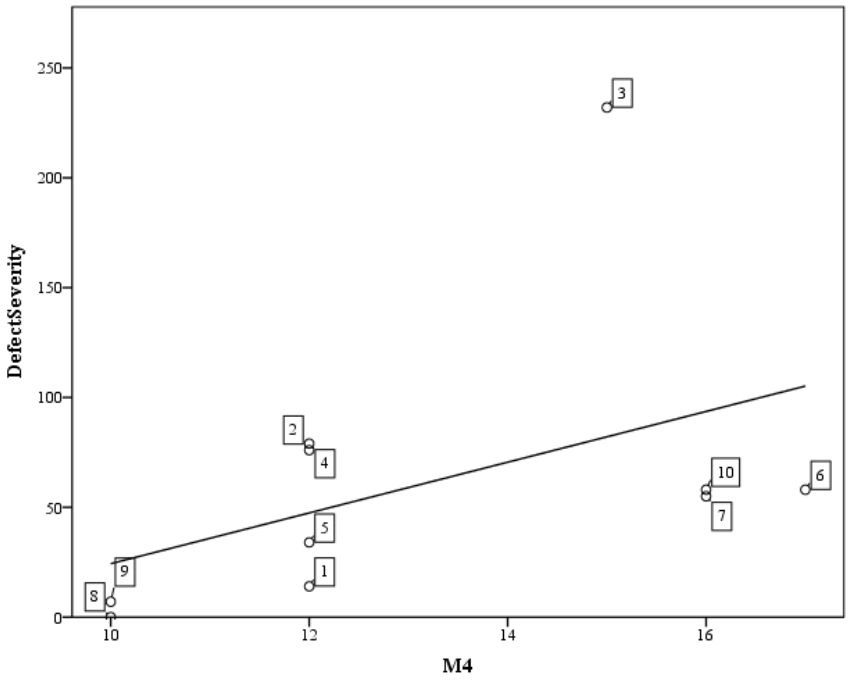

Figure 40 - Correlation between Defect Severity and M4

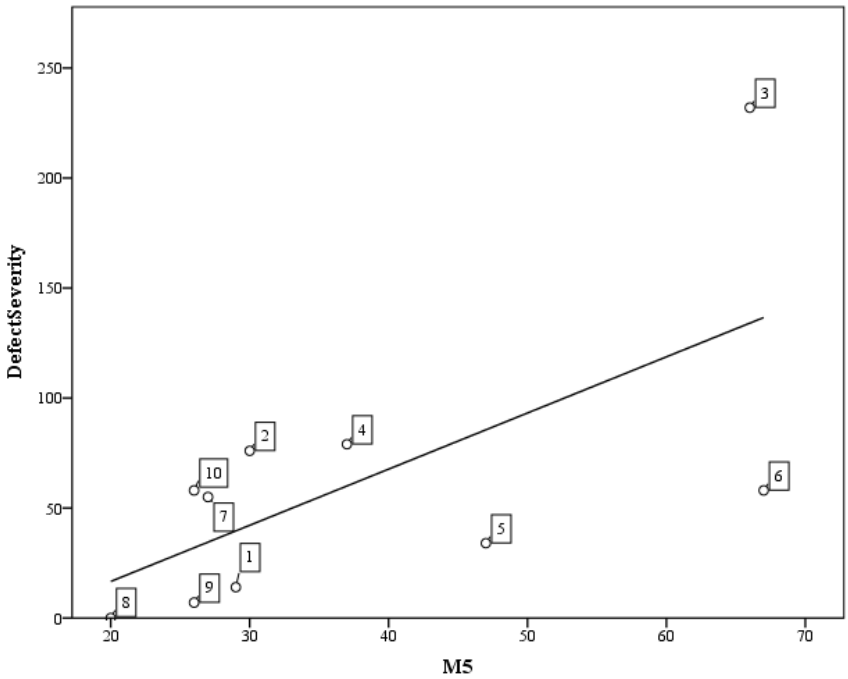

Figure 41 - Correlation between Defect Severity and M5

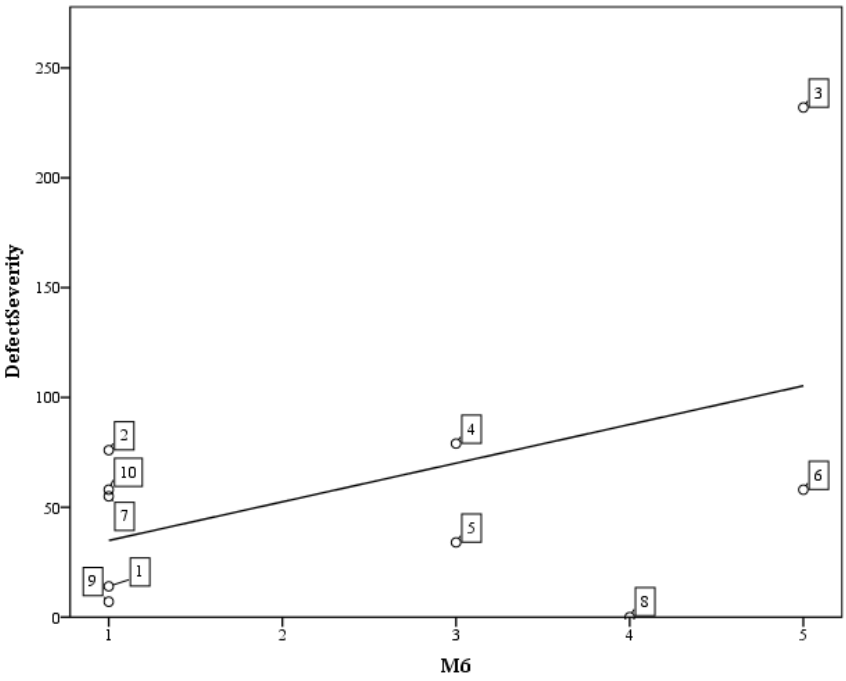

Figure 42 - Correlation between Defect Severity and M6

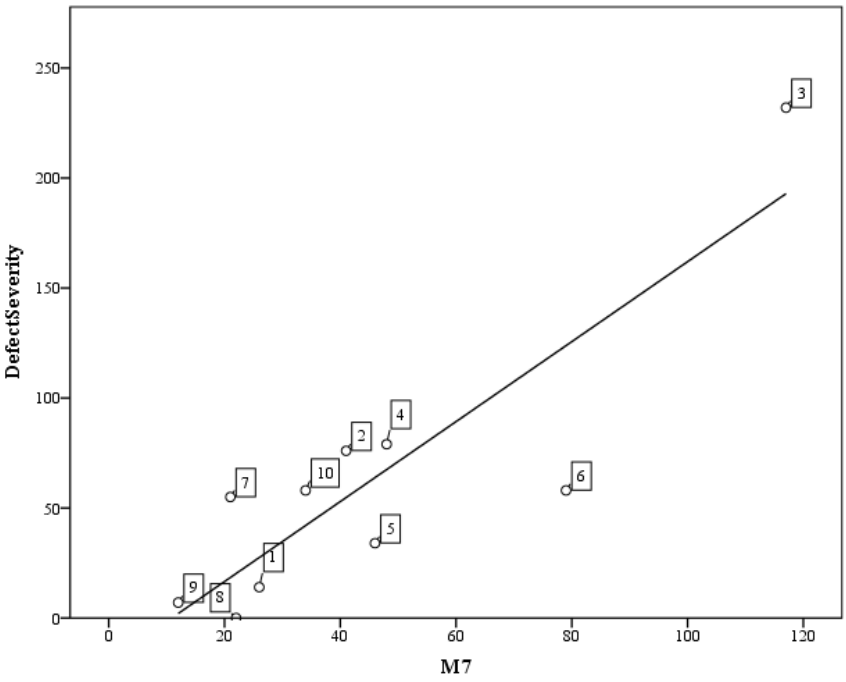

Figure 43 - Correlation between Defect Severity and M7

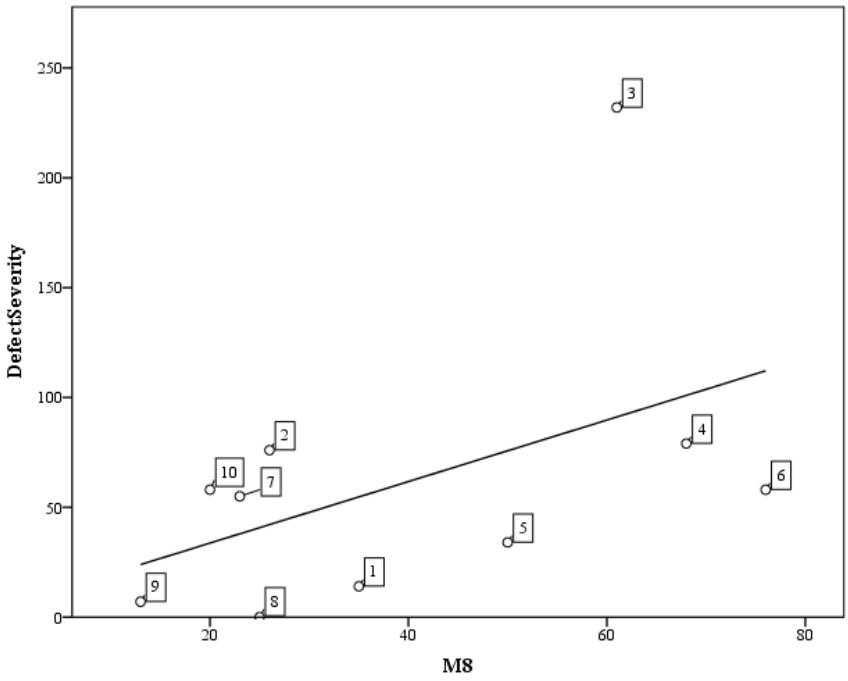

Figure 44 - Correlation between Defect Severity and M8

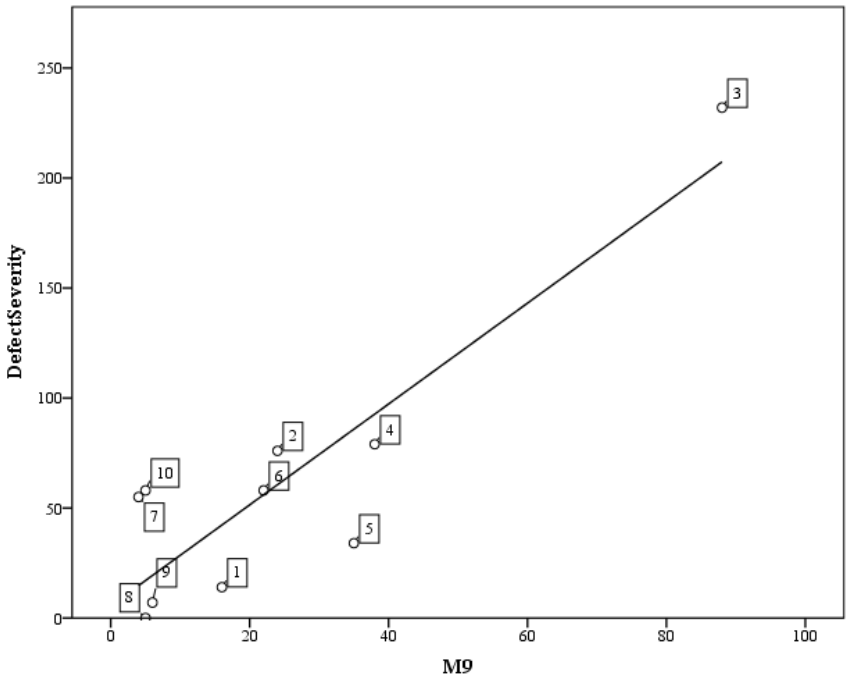

Figure 45 - Correlation between Defect Severity and M9

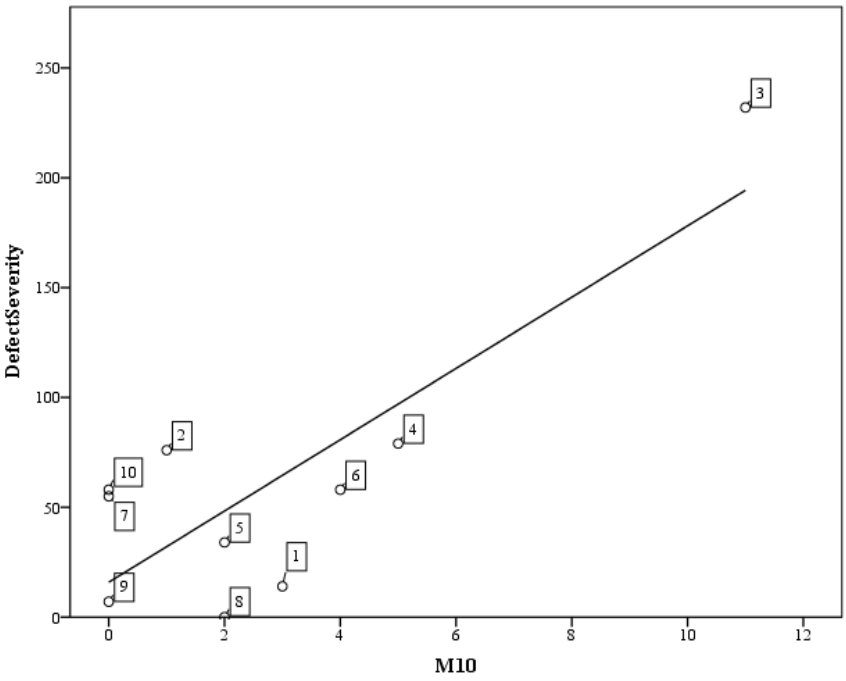

Figure 46 - Correlation between Defect Severity and M10

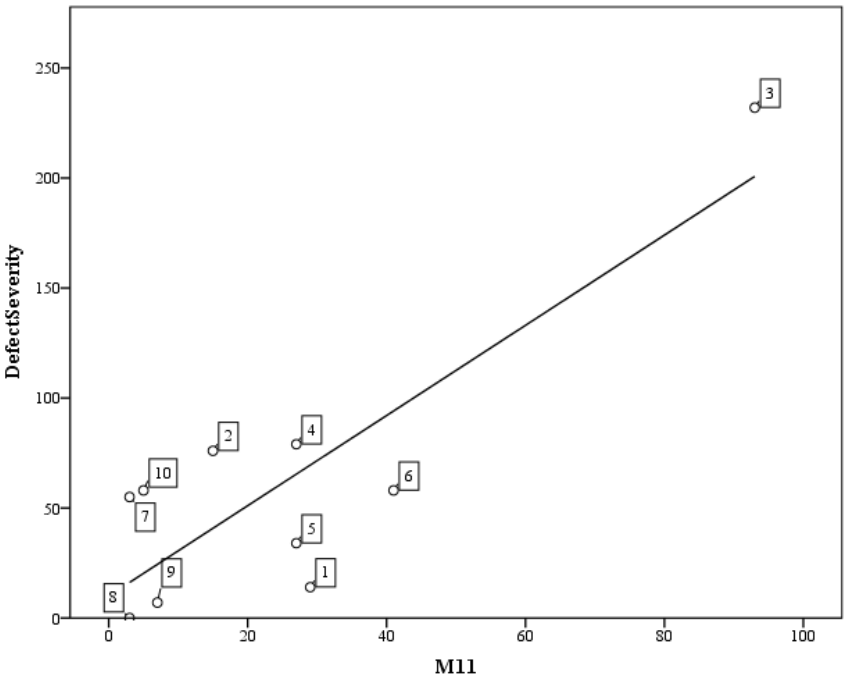

Figure 47 - Correlation between Defect Severity and M11

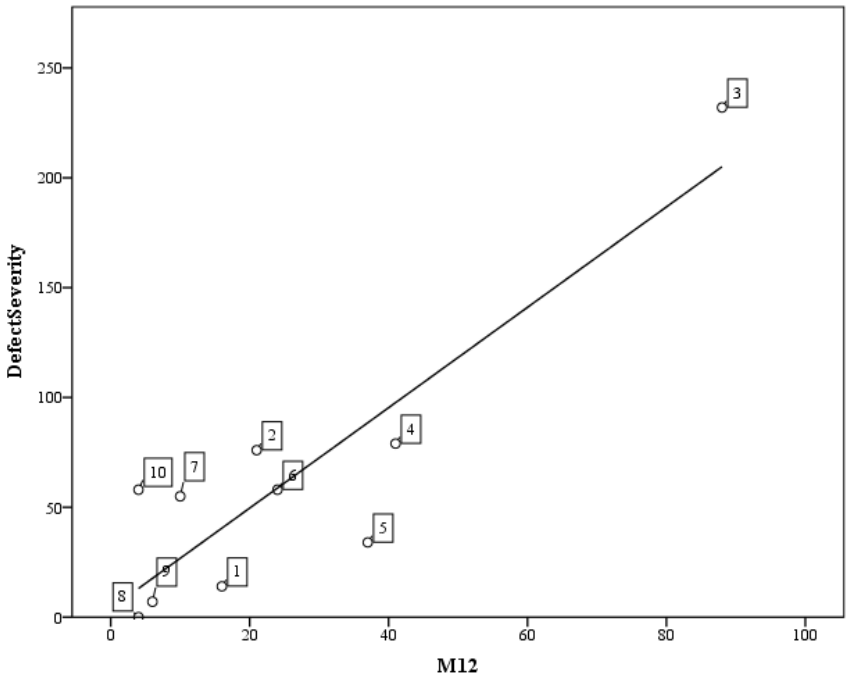

Figure 48 - Correlation between Defect Severity and M12

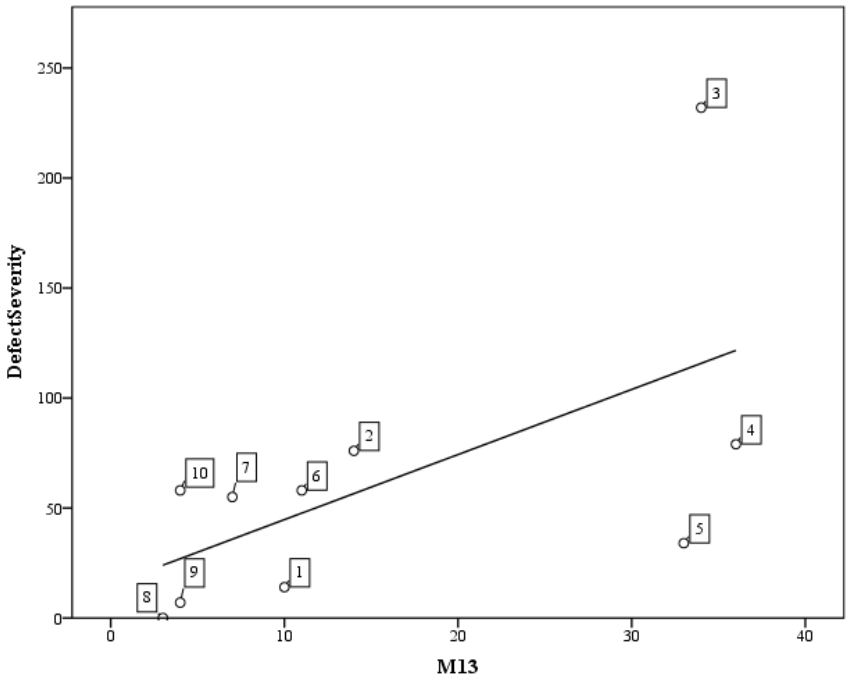

Figure 49 - Correlation between Defect Severity and M13

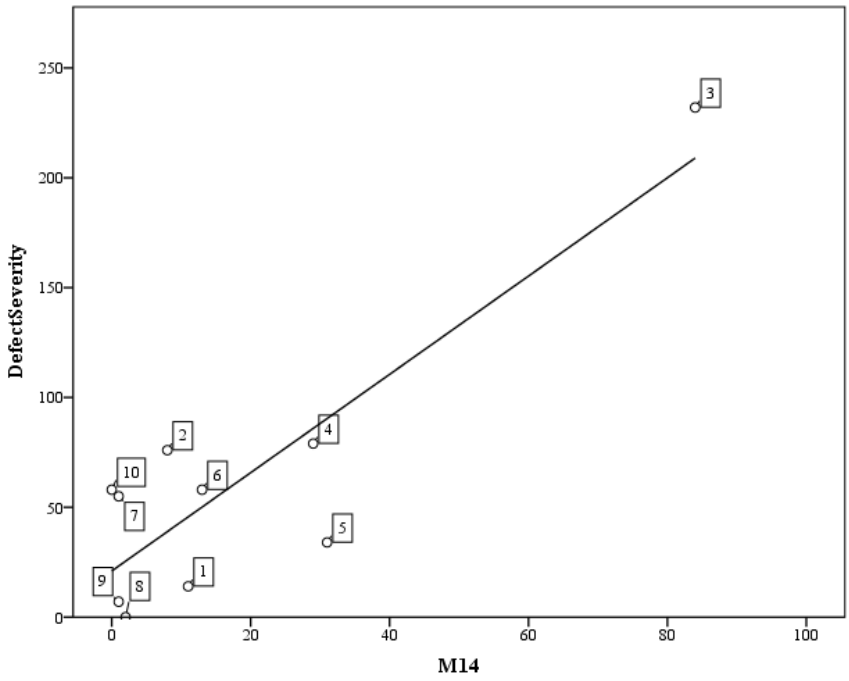

Figure 50 - Correlation between Defect Severity and M14

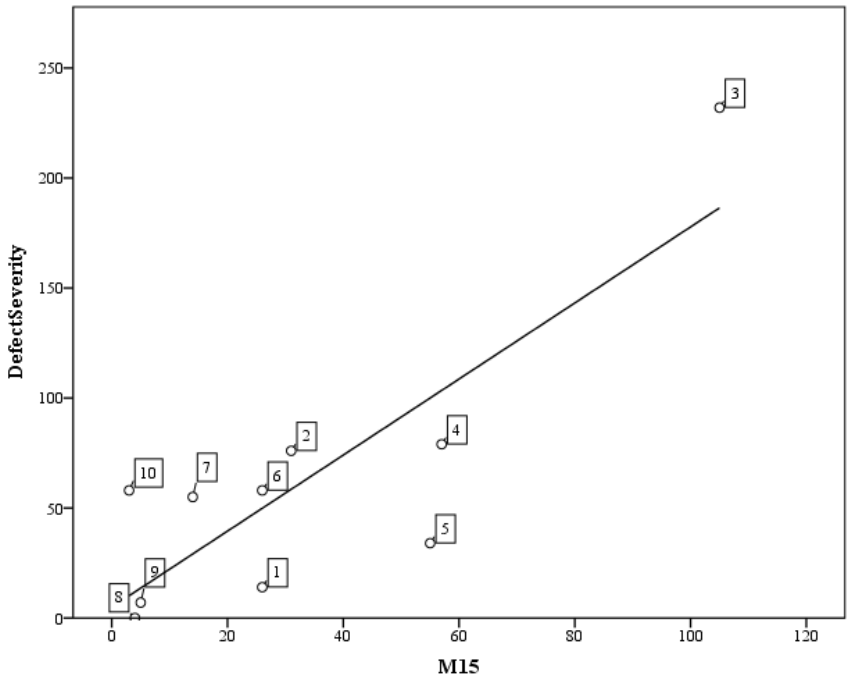

Figure 51 - Correlation between Defect Severity and M15

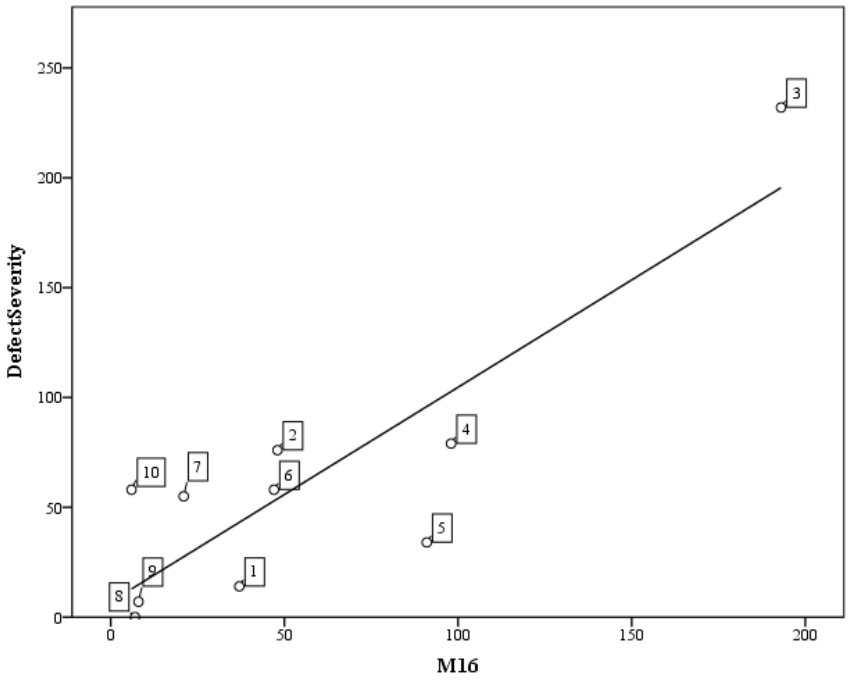

Figure 52 - Correlation between Defect Severity and M16

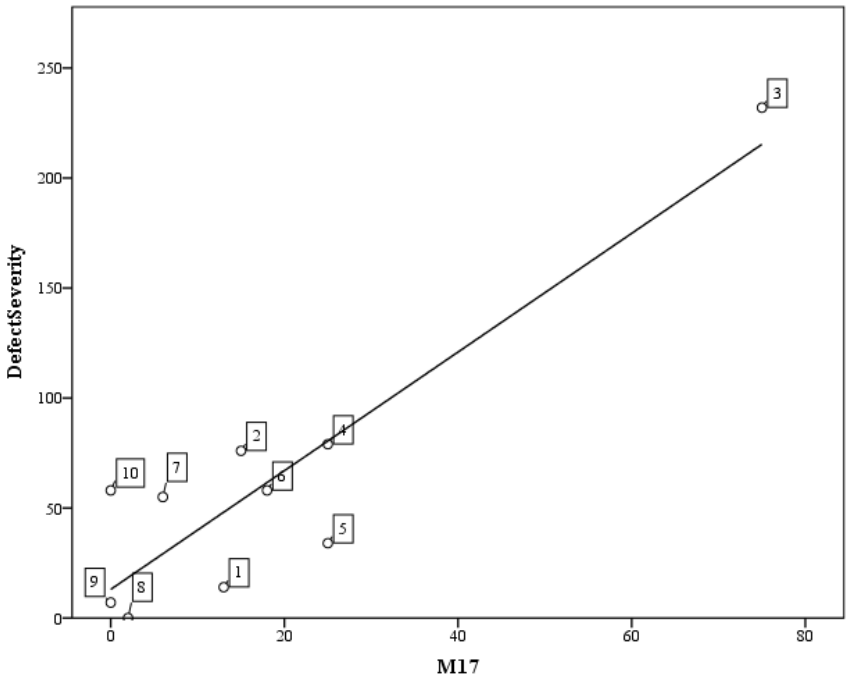

Figure 53 - Correlation between Defect Severity and M17

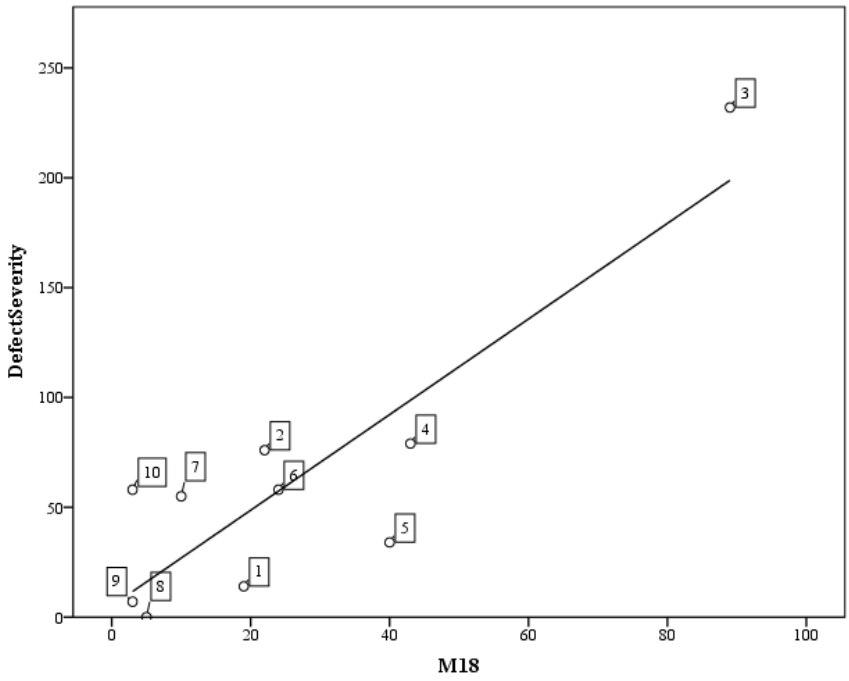

Figure 54 - Correlation between Defect Severity and M18

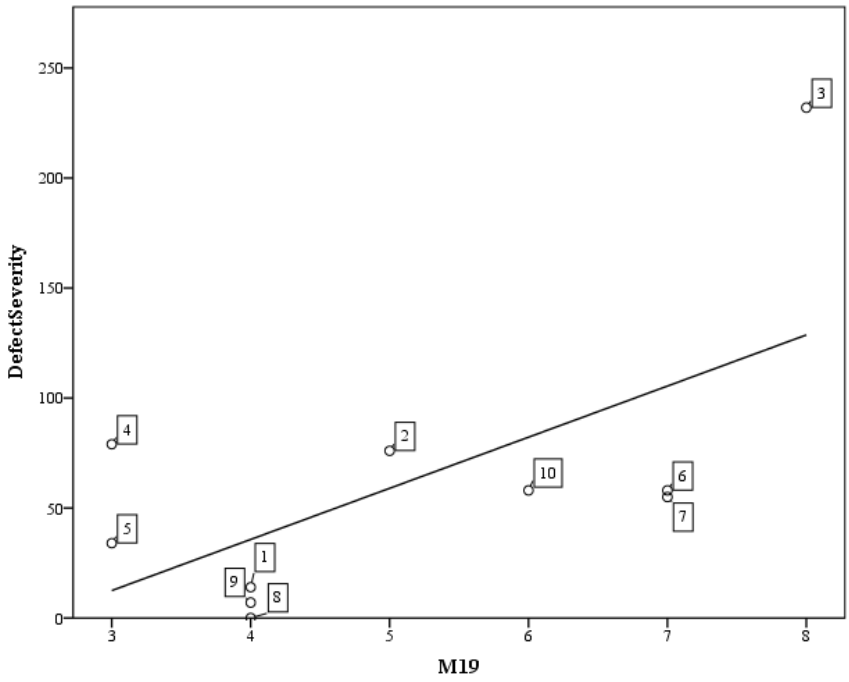

Figure 55 - Correlation between Defect Severity and M19

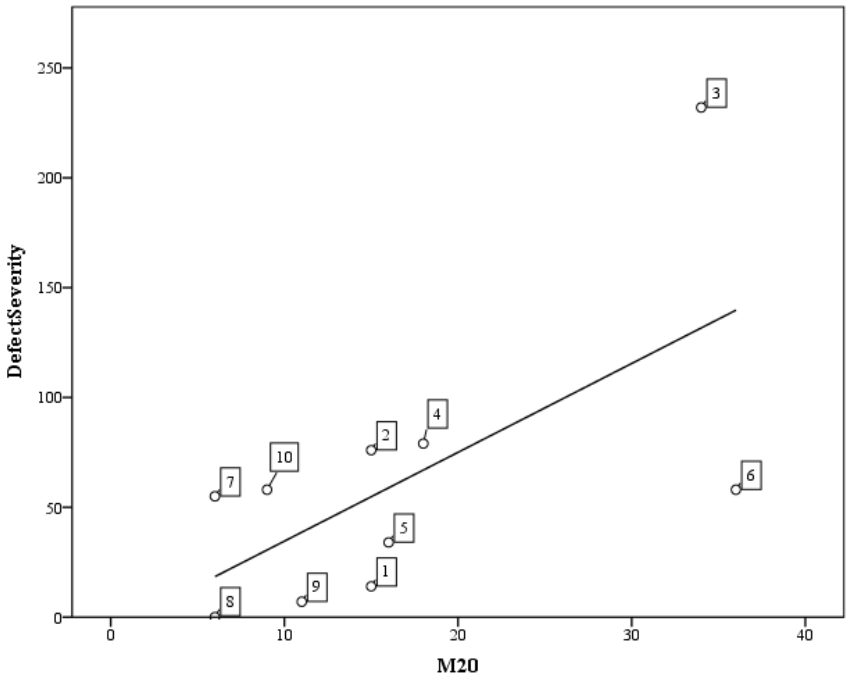

Figure 56 - Correlation between Defect Severity and M20

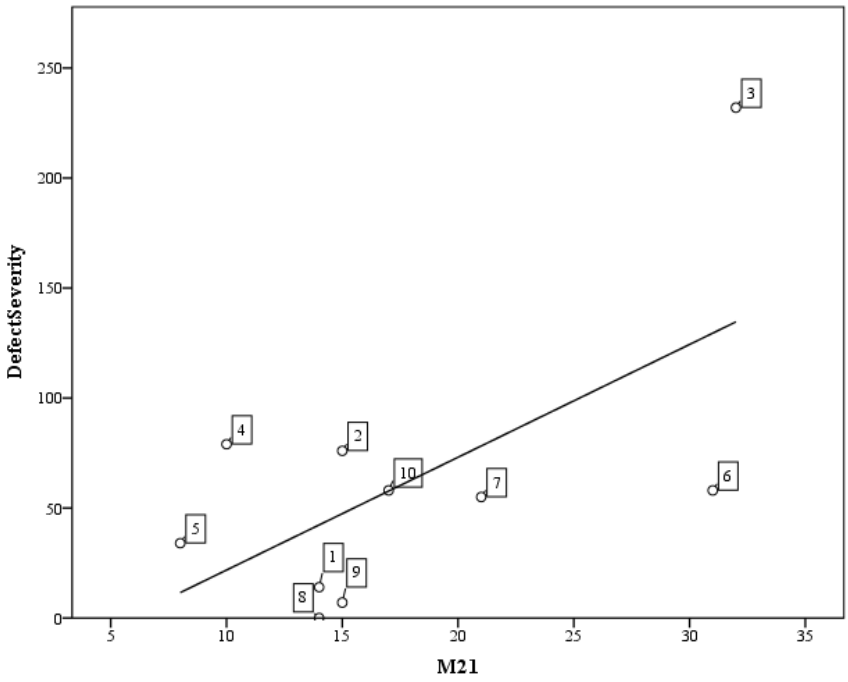

Figure 57 - Correlation between Defect Severity and M21

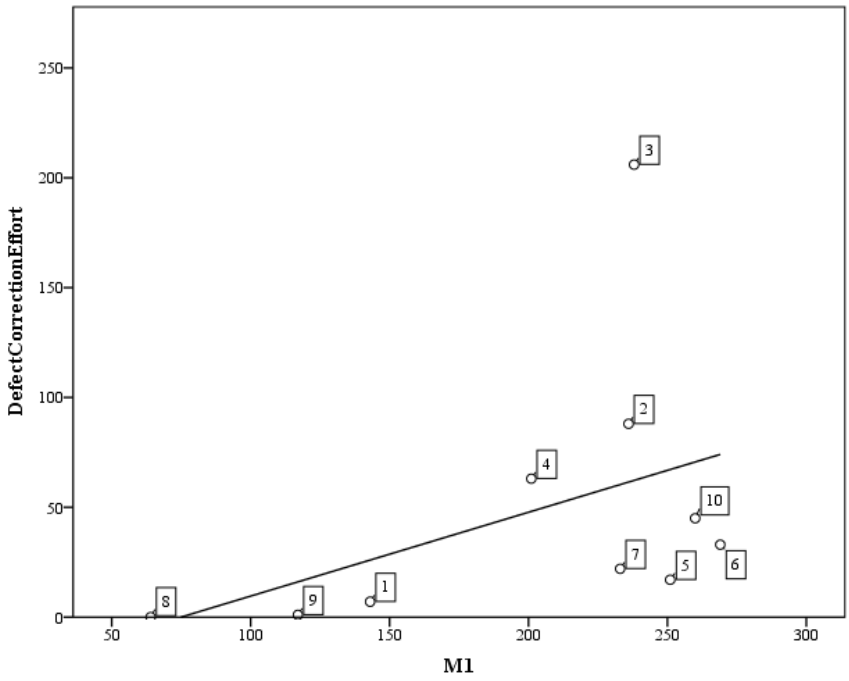

Figure 58 - Correlation between Defect Correction Effort and M1

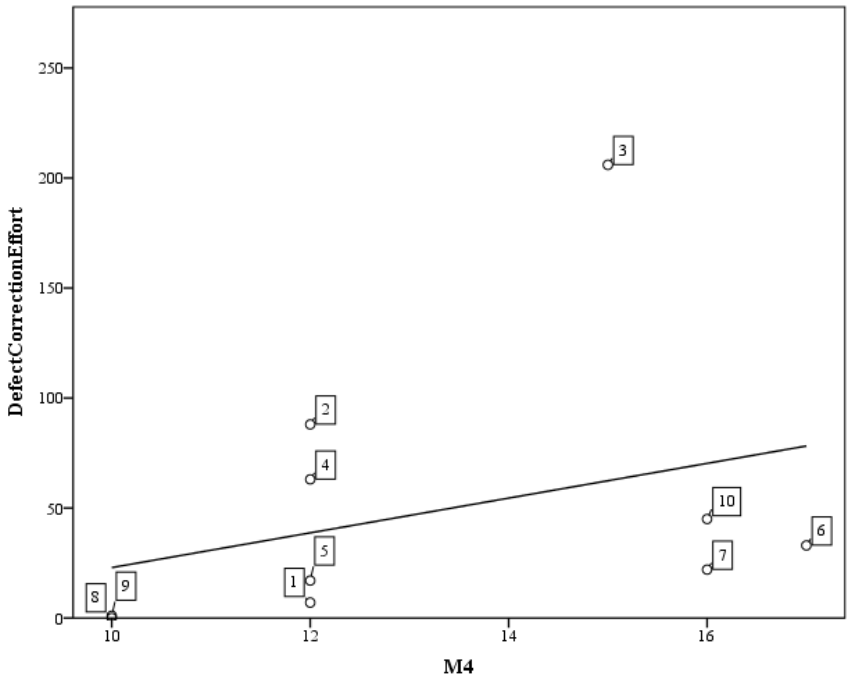

Figure 59 - Correlation between Defect Correction Effort and M4

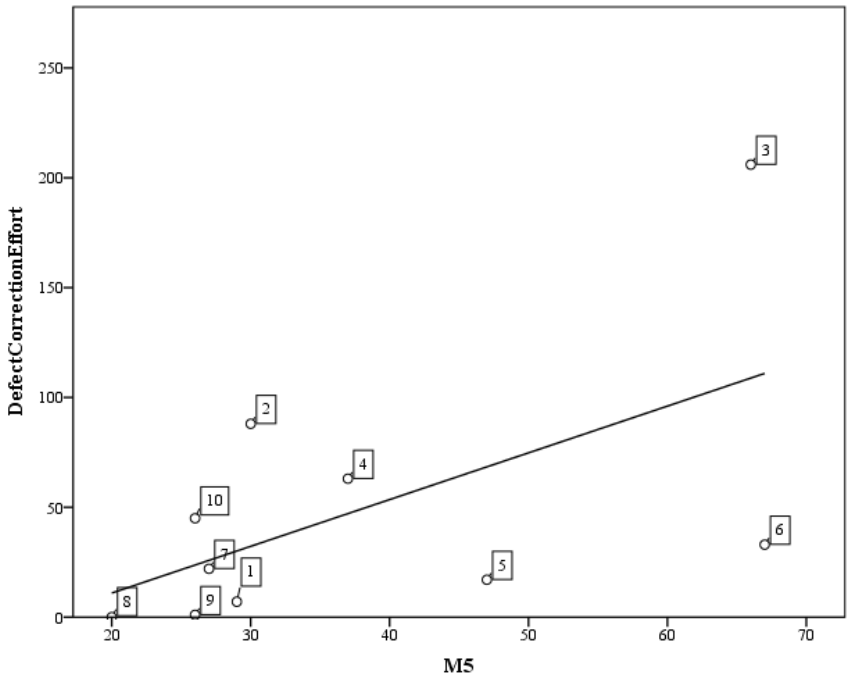

Figure 60 - Correlation between Defect Correction Effort and M5

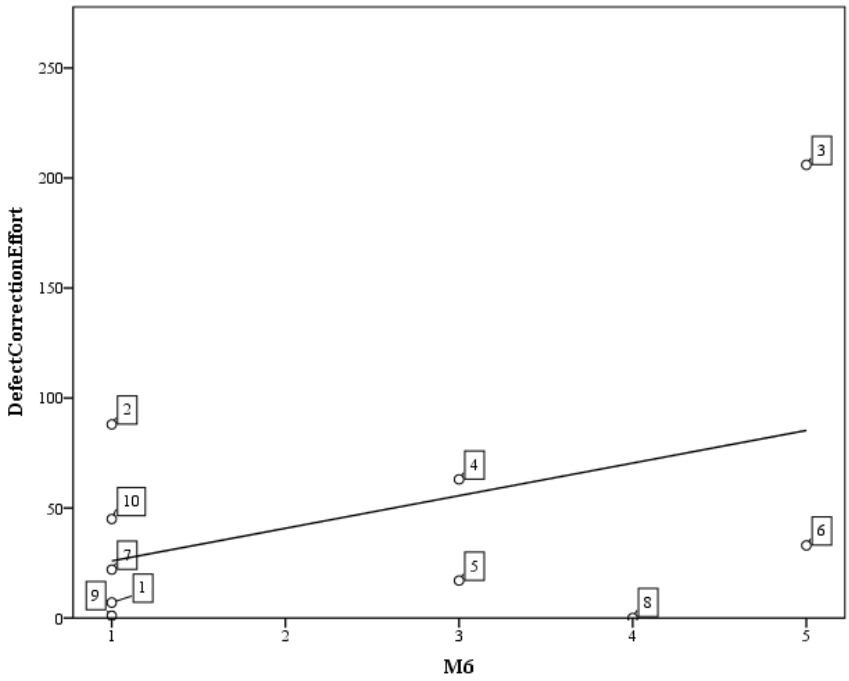

Figure 61 - Correlation between Defect Correction Effort and M6

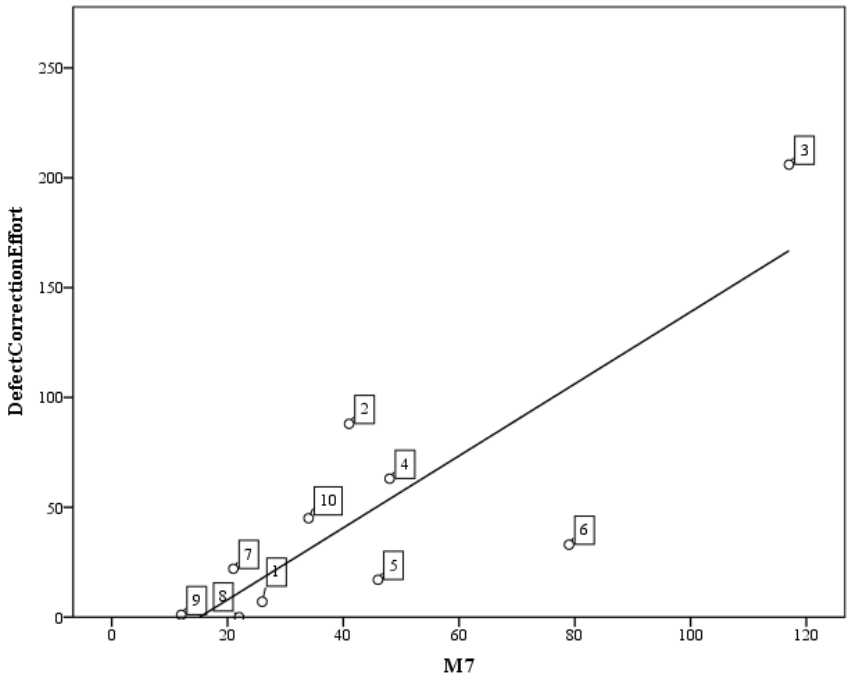

Figure 62 - Correlation between Defect Correction Effort and M7

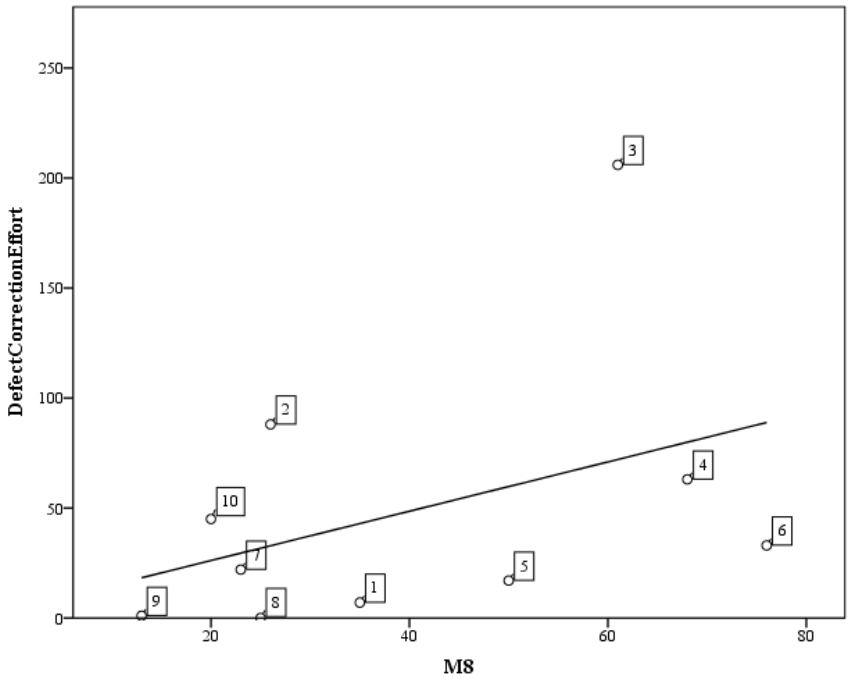

Figure 63 - Correlation between Defect Correction Effort and M8

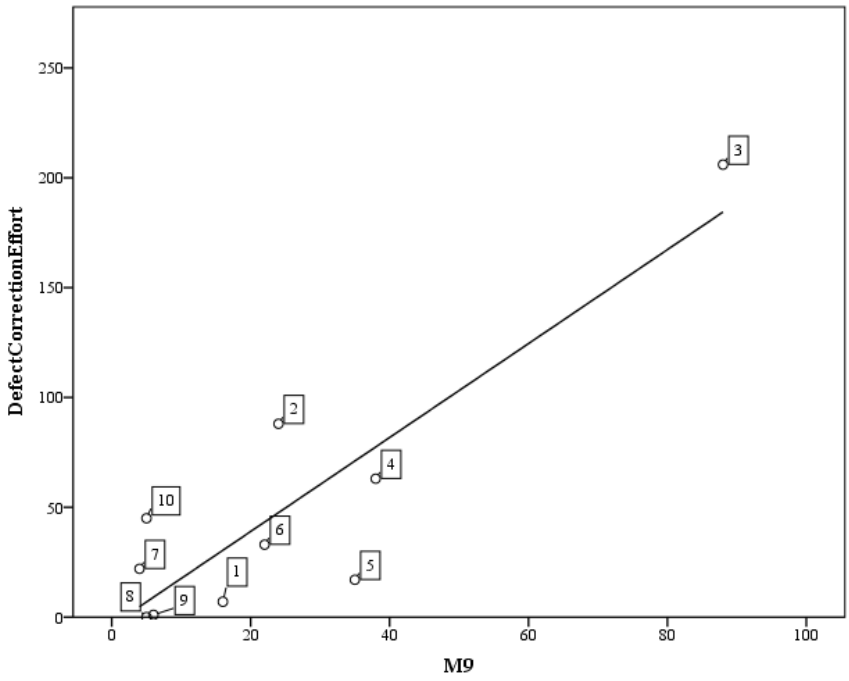

Figure 64 - Correlation between Defect Correction Effort and M9

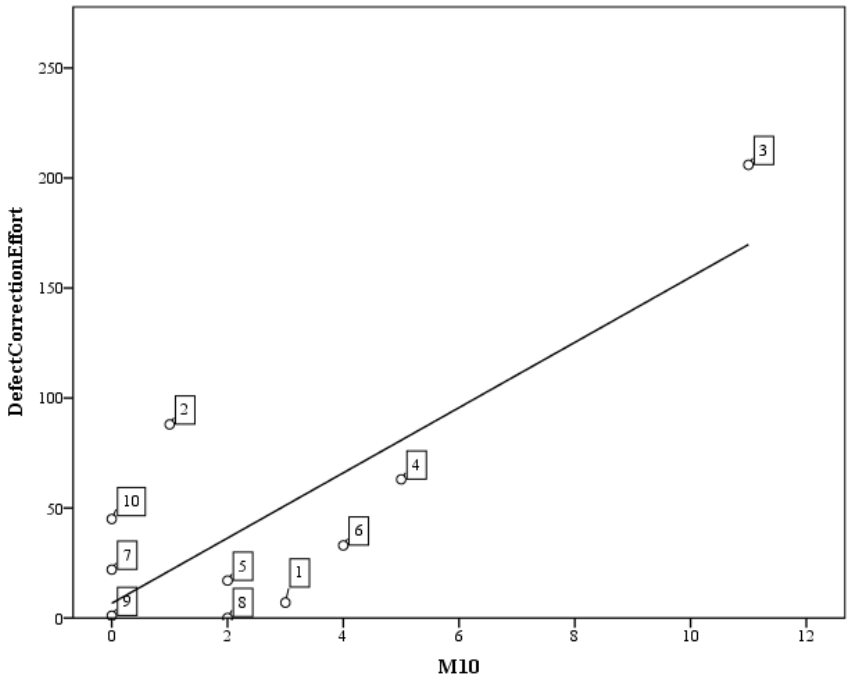

Figure 65 - Correlation between Defect Correction Effort and M10

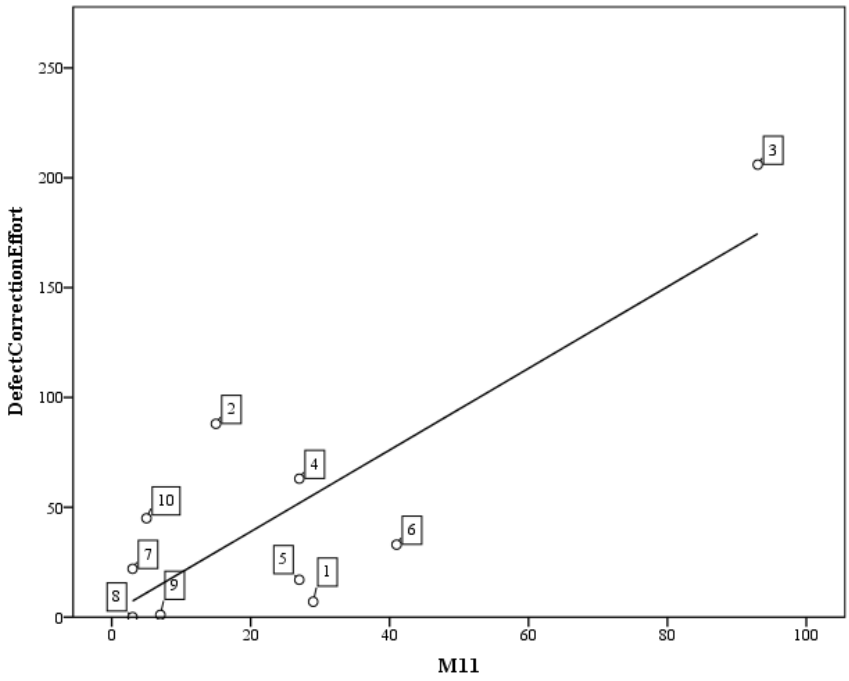

Figure 66 - Correlation between Defect Correction Effort and M11

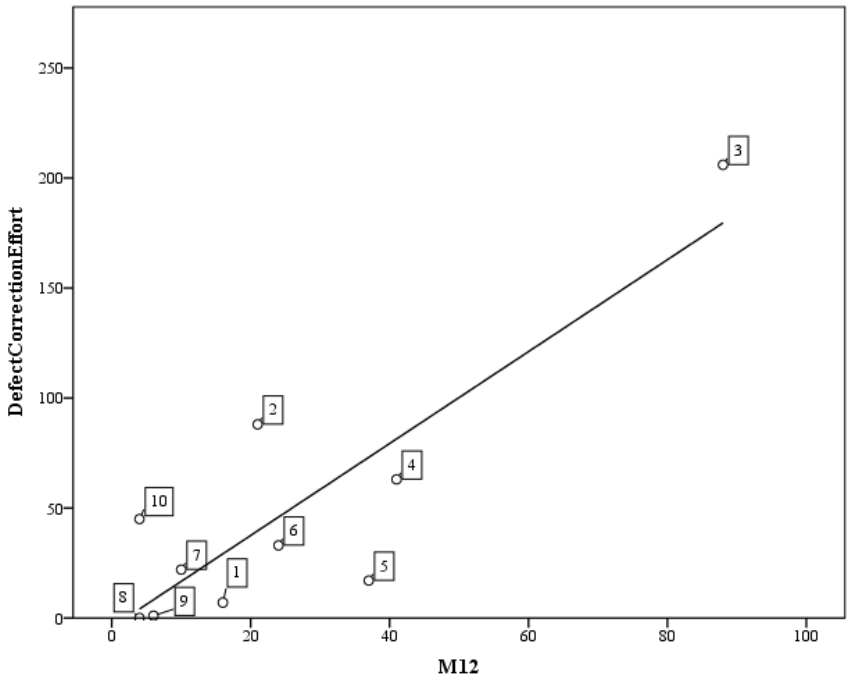

Figure 67 - Correlation between Defect Correction Effort and M12

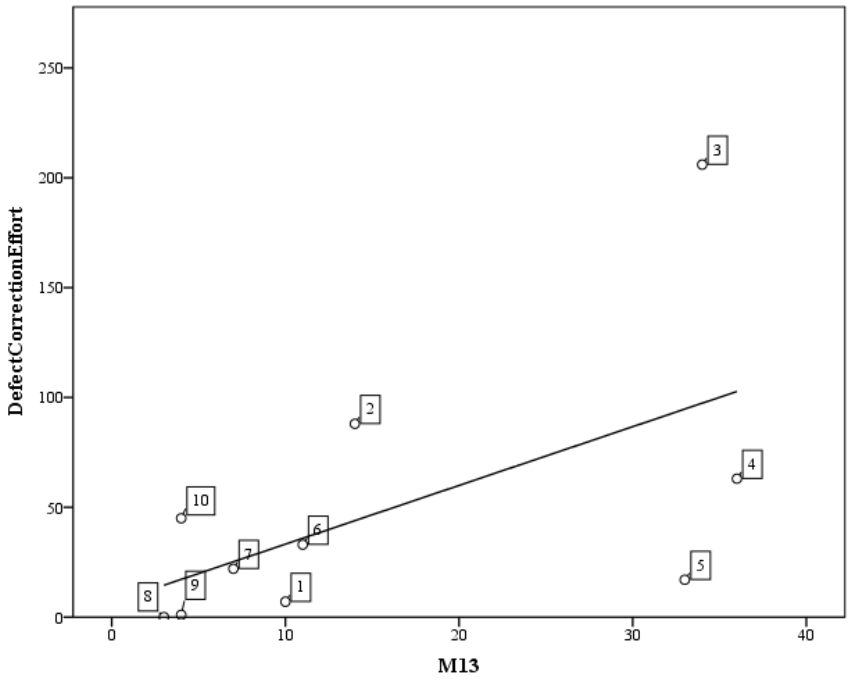

Figure 68 - Correlation between Defect Correction Effort and M13

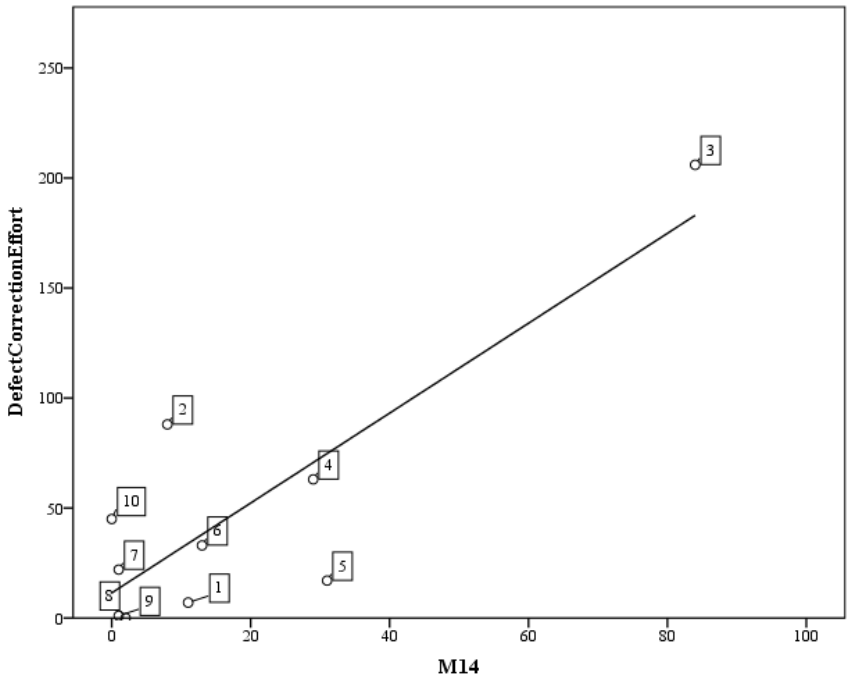

Figure 69 - Correlation between Defect Correction Effort and M14

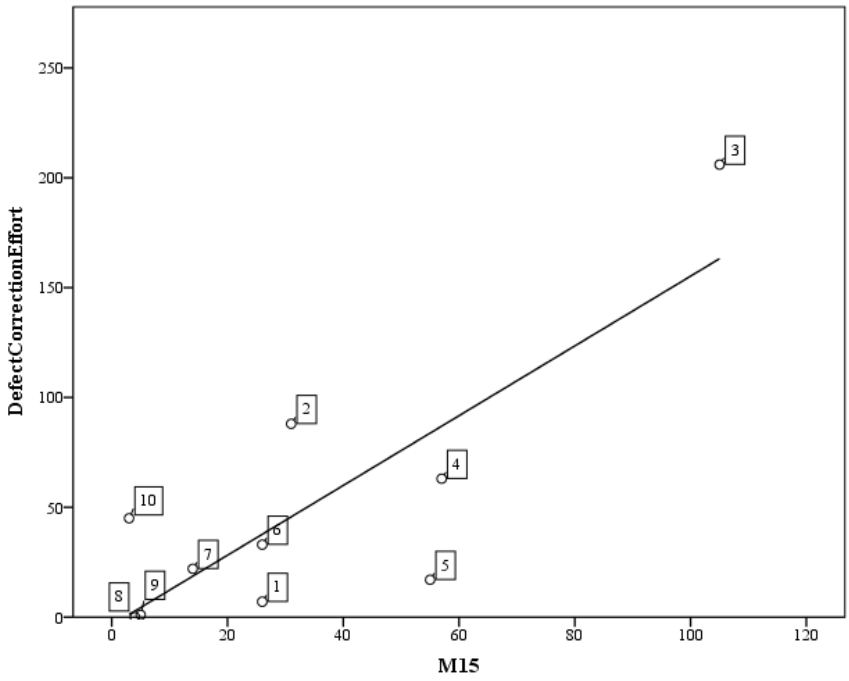

Figure 70 - Correlation between Defect Correction Effort and M15

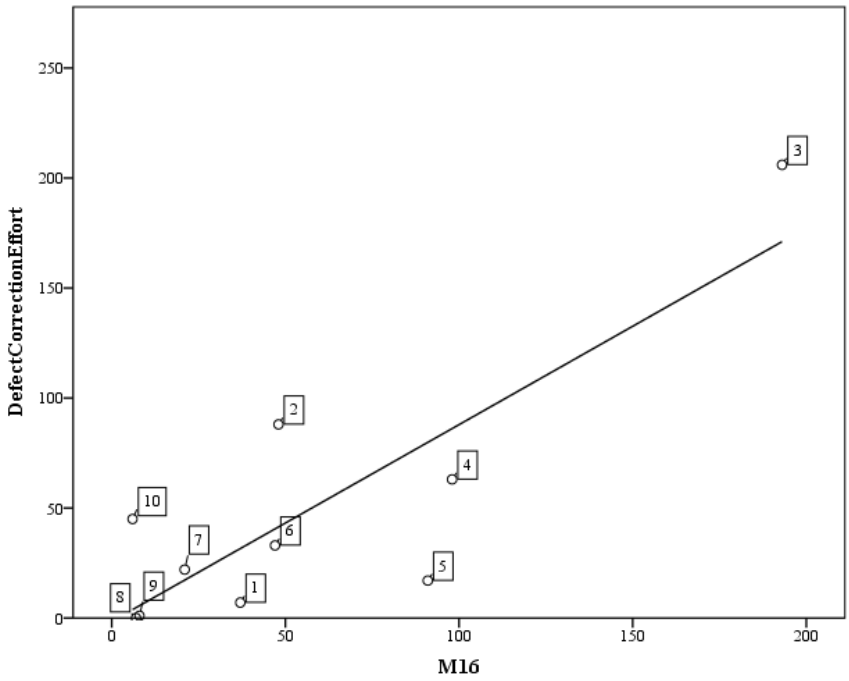

Figure 71 - Correlation between Defect Correction Effort and M16

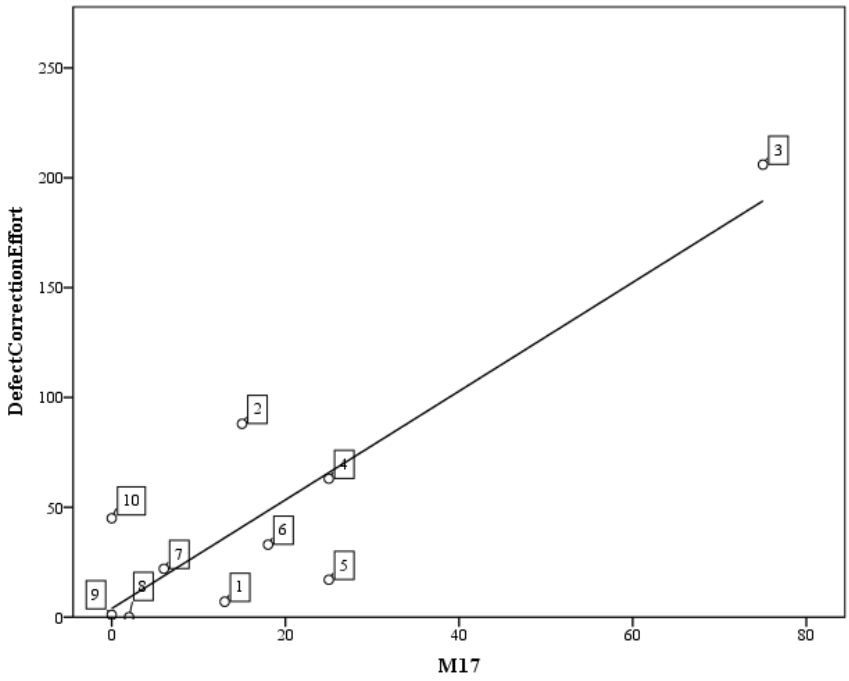

Figure 72 - Correlation between Defect Correction Effort and M17

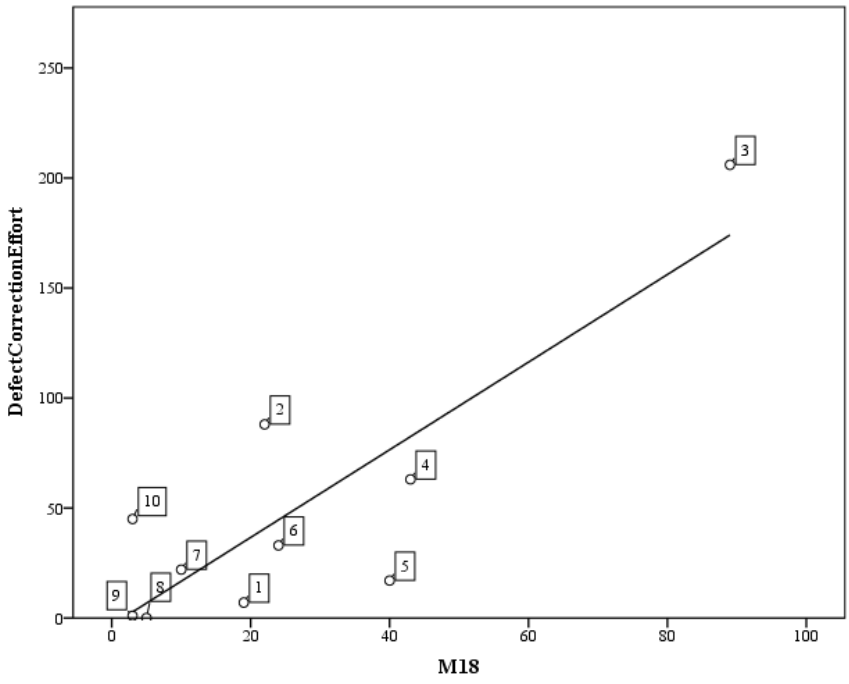

Figure 73 - Correlation between Defect Correction Effort and M18

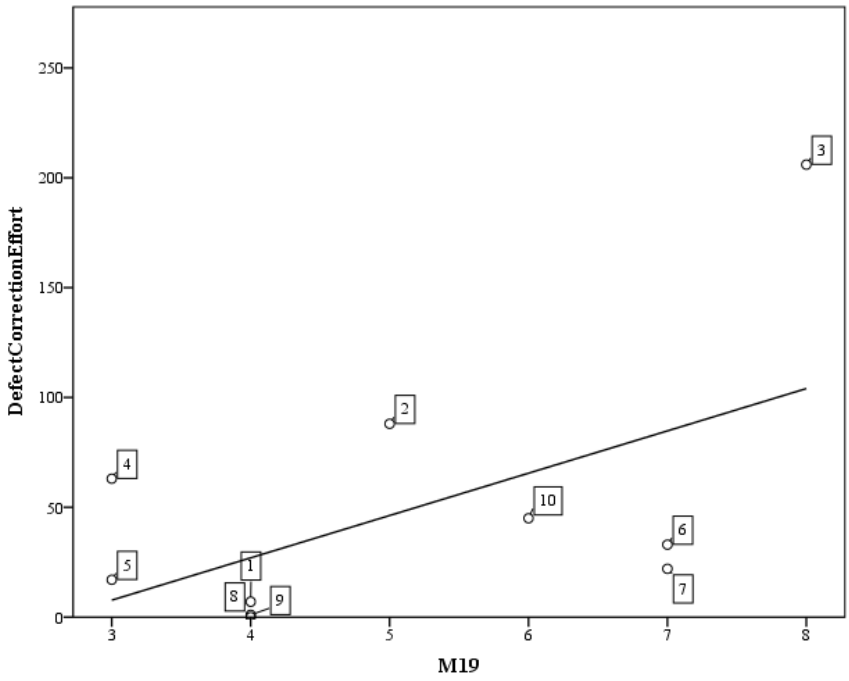

Figure 74 - Correlation between Defect Correction Effort and M19

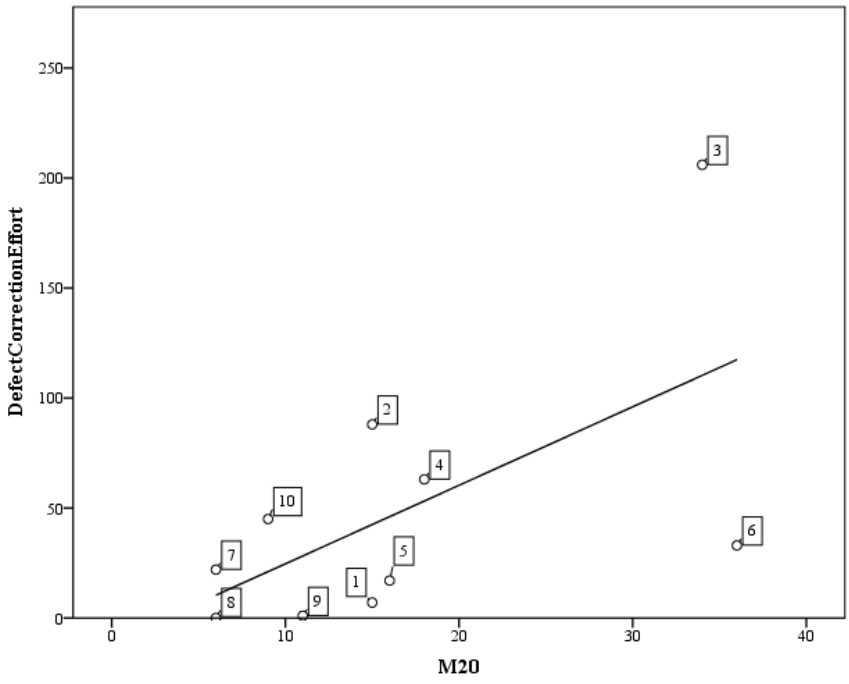

Figure 75 - Correlation between Defect Correction Effort and M20

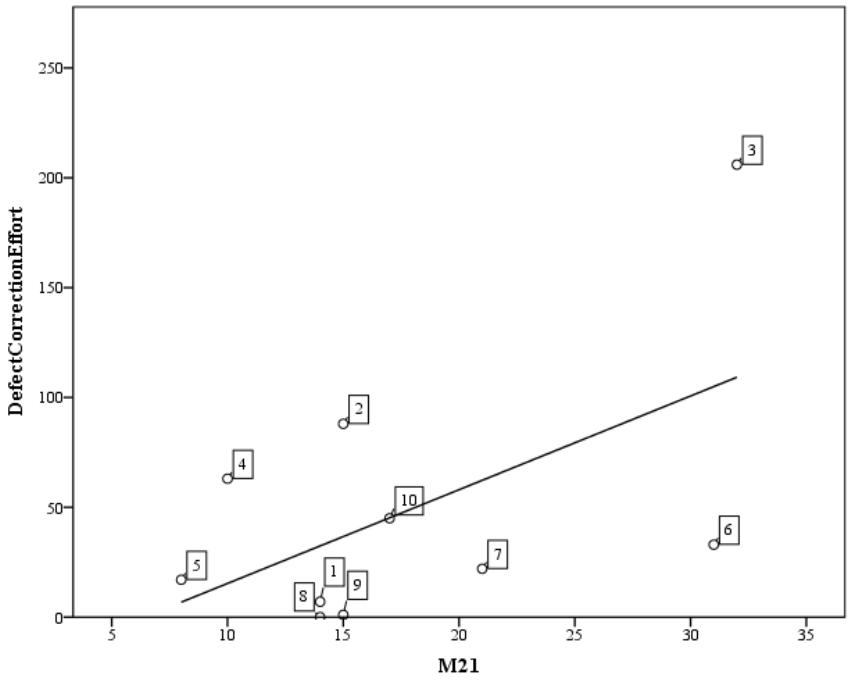

Figure 76 - Correlation between Defect Correction Effort and M21