A MEASUREMENT FRAMEWORK FOR COMPONENT ORIENTED
SOFTWARE SYSTEMS

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

BY

NAEL SALMAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

NOVEMBER 2006

Approval of the Graduate School of Natural and Applied Sciences

_____

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Doctor of Philosophy.

_____

Prof. Dr. Ayşe KİPER
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

_____

Assoc. Prof. Dr. Ali H. DOĞRU
Supervisor

Examining Committee Members

Prof. Dr. Mehmet R. Tolun (Çankaya, CENG) _____

Assoc. Prof. Dr. Ali H. Doğru (METU, CENG) _____

Assoc. Prof. Dr. Veysi İşler (METU, CENG) _____

Assoc. Prof. Dr. Ferda N. Alpaslan (METU, CENG) _____

Assist. Prof. Dr. Reza Hassanpour (Çankaya, CENG) _____

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: Nael Salman

Signature   :

ABSTRACT

A MEASUREMENT FRAMEWORK FOR COMPONENT ORIENTED SOFTWARE SYSTEMS

Salman, Nael

Ph.D., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Ali H. Doğru

November 2006, 116 pages

A measurement framework is presented for component oriented (CO) software systems. Fundamental concepts in component orientation are defined. The factors that influence CO systems' structural complexity are identified. Metrics quantifying and characterizing these factors are defined. A set of properties that a CO complexity metric must satisfy are defined. Metrics are evaluated first using the set of properties defined in this thesis and also using the set of properties defined by Tian and Zelkowitz in [84]. Evaluation results revealed that metrics satisfy all properties in both sets. Empirical validation of metrics is performed using data collected from graduate students' projects. Validation results revealed that CO complexity metrics can be used as predictors of development effort, Design effort, integration effort (characterizing system integrabiltiy), correction effort (characterizing system maintainability), function points count (characterizing system functionality), and programmer productivity. An Automated metrics collection tool is implemented and integrated with a dedicated CO modeling tool. The metrics collection tool automatically collects complexity metrics from system models and performs prediction estimations accordingly.

ÖZ

BİLEŞENE YÖNELİK YAZILIM SİSTEMLERİ İÇİN BİR ÖLÇÜM ÇERÇEVESİ

Salman, Nael

Doktora, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Ali H. Doğru

Kasım 2006, 116 sayfa

Bu tez bileşene yönelik (BY) yazılım sistemleri için bir ölçüm çerçevesi sunmaktadır. Bileşen yönelimindeki temel kavramlar açıklanmaktadır. BY sistemlerinin yapısal karmaşıklığını etkileyen etkenler belirtilmektedir. Bu etkenleri nicel ve nitel karakterlerini tanımlamaya yonelik ölçütler tanımlanmaktadır. BY karmaşıklık metriğininin gereklerini karşılayan özellikler tanımlanmaktadır. Ölçütler iki kere değerlendirilmektedir: İlk değerlendirmede bu tezde tanımlanmış olan özellikler kulanıldı. İkinci değerlerndirmede ise Tian ve Zelkowitz tarıfından tanımlanmış olan karmaşıklık ölçüt özellikleri [84] kullanıldı. Değerlendirme sonuçları, her ikisindeki özelliklerin gerçekleştirildiğini ortaya koymaktadır. Ölçütlerin görgül geçerliği yüksek lisans öğrencilerinin projelerinden toplanan veriler üzerinde gerçekleştirilmiştir. Geçerlik sonuçları BY karmaşıklık ölçütlerinin aşağıdaki parametreleri kestirim maksadı ile geliştirme sürecinde kullanılabileceğini ortaya koymaktadır: Tasarım çabası, entegrasyon çabası (sistem entegre edilebilirliğinin karakterizasyonu), düzeltme çabası (bakım yapılabilirlik karakterizasyonu), FP (sistem işlevselliğinin karakterizasyonu), ve programcı üretkenliği. Otomatik ölçüt toplama aracı gerçekleştirilmistir ve özgül bir BY modelleme aracı ile entegre edilmiştir. Ölçüt toplama aracı otomatik olarak sistem

modellerinden karmaşıklık ölçütleri toplamakta ve öngörü kestirimini buna göre gerçekleştirmektedir.

Anahtar Kelimeler: Bileşen yönelimi, Karmaşıklık, Yapısal karmaşıklık, Ölçüt , Ölçüt Otomasyonu.

To My Parents, My Wife, and My Children with Love

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE

CHAPTER 1

INTRODUCTION

Computers are every where; transportation, education, medical, governmental, and several many other fields nowadays are highly dependent on computer systems [68]. A Computer system is mainly composed of a hardware subsystem and a software subsystem. The well functioning of a computer system is dependent on the well functioning of both its software and hardware subsystems. While the steps of building efficient hardware systems remain beyond the scope of this research, we will focus on evaluating features of software system that can lead to the production of efficient and cost-effective software.

Development of software systems starts with system specification, proceeds with design which mainly comprises building models of the real world and approaching system complexity by decomposition. After that comes implementation of the specified models and system building blocks using a programming language that has constructs supporting the specified models. The last step in the development process is integrating or unifying the implemented and tested system building blocks. The key to approaching system complexity, managing performance, security, maintainability, and other important system features is decomposing the system into smaller units or modules which will in turn be the system building blocks [81]. Use of Abstraction, as the key to the identification of system building blocks or components, has been of great interest to software developers since the early days of software development. The earliest work started with process abstraction which was not powerful enough to build large and complex programs.

Then appeared the data processing view, emphasizing function abstraction that receives inputs when called, does processing in its body and yields a value as output [74, 81]. Later, and more extensively, the object-oriented (OO) approach appeared and introduced a different view of abstraction which encapsulates both data and functions into its fundamental building block "the class" which is a collection of objects, and hides information from its clients. The class abstraction allows building large and complex systems as hierarchies of objects [74]. Most recently, the component oriented (CO) system development approach with software component as its principal building block. The aim of breaking the system into smaller units (functions, classes, or components) is to manage the complexity in the systems following the widely known rule "divide and conquer". While the traditional approach focuses on functions, the OO approach focuses on data and the CO approach focuses on structure [34].

1.1 Component Oriented Software Engineering

"Reuse, reuse, and more reuse until finally you can develop large software systems by integrating already available components rather than writing code from scratch". This is the main objective of the so many research centers, software development organizations, and software customers as well. Everyone involved in the software system development process looks forward to having software systems more rapidly built and are more efficient.

Building software systems by integrating already available components has not been successfully used before the 1990s [83]. Commercials-Of-The-Shelf (COTS) components, Component Based Software Engineering (CBSE), and Component Oriented Software Engineering (COSE) are all terms referring to the new and rapidly growing approach of software development that mainly focuses on building large, and efficient software systems mainly benefiting from reusability and composition rather than code writing. Although the terms CBSE and COSE are used interchangeably most of the time in the literature they are different in their

entirety. The difference between these two terms is similar to the difference between Object Based and Object Oriented development approaches. Dorgru and Tanik [34] well-described the difference between CBSE and COSE as follows: CBSE focuses on using pre-built components while the whole system can be modeled using OO methodologies. That means CBSE considers components only at the system integration phase. On the other hand COSE requires that all stages of the development process must be component oriented. COSE suggests that the analysis and design stages of the system must be CO in order to successfully apply the idea of "build by integration rather than code writing". In this respect, traditional and object oriented paradigms fall into prescriptive category where the idea is to write code; all the leading phases are geared towards organizing the way how code will be written. Component Oriented development considers integration rather than code writing. Also, Requirements and Design stages are supported with abstract and practical concepts that correspond to components, rather than prescriptive structures such as classes, objects, or data/control structures. Component Based development is a hybrid approach where code writing is supported as well as the incorporation of components; pre-coding stages, however, are prescriptive namely Object Oriented. The component related activities in such an approach are more bottom-up, concentrating on the "wiring level" techniques for the composition of components to the hybrid system. The Component Orientation as supported in this study and the guiding references differ mainly in the integration view, and the promotion of component concept as the fundamental building block, in all phases. The component and its abstract level representations are the focus of modeling hence rendering the concept as a consistent structure from requirements to run-time, as classes are for Object Orientation.

Clements [24] listed the main advantages of applying component technologies as:
- Reduced development effort
- Increased reliability and efficiency.
- Increased flexibility and many alternatives offered to choose from.

Clements [24] also discussed the main issues and difficulties in applying COSE as:

- Lack of standards describing ways of communications between components coming from different environments.
- Component architectures and infrastructures should be identified
- Customers can receive no version support and evolution can be limited

Computer science and/or engineering departments in many universities initiated research which shares a common objective of maximizing reusability and minimizing code writing. The software industry practices related to components can be put in two categories: While several market leading software development organizations intensified their work on component platforms and component technologies, other software development organizations focused on producing and marketing components.

Microsoft is one of the leading organizations in creating component wiring technologies. Microsoft first created COM wiring technology which then improved to COM+, DCOM, and ActiveX/OLE infrastructures, and lastly provided a wider support for components in its .NET CLR framework which adds the interoperation of COM+ and Windows platform access services [59 and 83]. Sun is another market leading software developing organization that has very important contributions to the component technology. Although some java applets can be sold as separate components, it is difficult to generalize that on all applets [83]. The major contribution of Sun Java™ to the component technology was the introduction of the java bean (a bean is really a component) technology [49].

The Object Management Group (OMG) -a non-profit organization- is also working on defining and developing infrastructure for the interoperability of objects at all levels. OMG developed the Common Object Request Broker Architecture (CORBA®), which was then refined in CORBA 1.1 and CORBA 1.2. CORBA2.0 marked a significant improvement over CORBA 1.x by enabling client portability. CORBA infrastructure became mature only after the release of CORBA 2.0 which was then followed by several successive version labeled as CORBA 2.x where each version enhanced the version preceding it. Finally, OMG released CORBA3.0

which marked a significant improvement on all previous versions and provided the most support for component technology [62, 83].

On the other side, several software development organizations started developing and marketing software components. Although several such organizations exist, we will just list few such organizations. Our selection is pure subjective and does not have an implication about quality, cost, technology, or any other aspects related to the components developed or the developing organization:

1. Component Source: founded in 1996, produces components that serve several disciplines such as accounting, data mining and databases, speech recognition, image processing, CAD, web services, editing and word processing tools, and several other disciplines [26].

2. Dev Direct: founded in 2003, Dev Direct is marketing components from a variety of disciplines that work on almost all platforms. Dev Direct is an intermediary between publishers and customers (software developers) [32].

The fast growth in the interest in CO software development in both academic and business cycles is due to the several advantages it provides for building large and efficient software systems. CO focuses mainly on integrating already available components. Among the advantages of using component oriented software development are: 1) economic necessity and saving in development costs, 2) providing higher quality software, and more adaptable systems [9, 34, 75, and 83].

1.2 The need for Software metrics

Tom Demarco formulated the following about measurement *"You can not control what you can not measure"* [38]. Tom De Marco best summarized what Lord Kelvin (1824-1904), formulated about measurement *"When you can measure what you are speaking about, and express it into numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: It may be the beginning of*

5

knowledge, but you have scarcely in your thoughts advanced to the stage of science" Quoted from Zuse's website [95].

Metrics can be collected at different stages of the software development process. What metrics to collect and when to collect them is an issue that still does not have an agreed-upon answer yet. Hellerman supports that metrics are needed to compare among different alternatives [45]. Software metrics have proven to be an essential actor in the software development process and they are essential to have a successful software development environment [58]. The most widely cited viewpoints about the role(s) of software metrics can be summarized as follows:

Metrics can be used to build prediction models: El Emam [35] demonstrates that metrics can be relied on to build predictions about software errors. He also found that prediction models of errors based on metrics have error rates of about 9% while savings in maintenance costs reaches up to 42%. Mendonça and Basili [58] emphasized that software measurement is needed to characterize, control, predict, and improve software development, management, and maintaining processes.

Metrics help managers to make decisions: A lot of organizations (Software Engineering Labs at NASA and HP) use metrics to make managerial decisions related to resource distribution, cost estimations, and building defect and productivity models [10].

Use of metrics increases productivity: It has been noticed that productivity in software development is dropping with a rate higher than that in any other industry; it is estimated to have dropped by 10% from the year 1990 to the year 1995. The most important reason behind this is the lack of benchmarks to make comparisons. Productivity is found to be highly affected by the degree of connectivity which is a measure of coupling [7].

Metrics can decrease software defects: Grady in his book about software metrics [40] presented a generalized discussion of software metrics. He stated that good metrics programs can decrease software defects by 50-75%. Grady, discussed

some of the widely known metrics such as *Cyclometic complexity* [57] and concluded that it is hardly worth the effort since it is derived from the code. He suggested that metrics that are collected before the code is ready can be of higher value. Defects found during inspection before the code is complete cost less than one tenth of those found after the system delivery and furthermore affect negatively the reputation of the organization. This follows the famous saying *"Prevention is better than cure"*.

Metrics can be used as quality indicators: Schneidewind defines a metric as a function whose inputs are software data, and its output is a numerical value that describes the degree to which software possesses does or does not possess a given quality aspect [73]. Blundell et al. [12] related software metrics to different quality factors and found that design metrics can be used to: 1) evaluate current levels of software design qualities within the project, 2) decompose the problem into an acceptable set of components, and 3) identify the critical parts of the design [12]. Basili and his colleagues [10] provided empirical evidence that object-oriented design metrics can be used as quality indicators.

Metrics <u>may not</u> accurately build prediction models: Fenton and Neil describe two classes of software metrics; classical metrics that describe software attributes using numbers and the other group of metrics which are used to predict external features like cost and quality of software. The authors also stressed that one of the major problems in the field of software measurement is the weak links between industry practices and current research [37]. The authors also claim that complexity and size measures are not enough to predict software defects accurately. This viewpoint has been proven to be inaccurate through empirical researches including the one described in this thesis.

1.3 Motivation

As new technologies appeared, engineering approaches to utilize them lagged in the history. OO methodologies appeared years after OO languages. Similar phenomenon is valid today, for CO approaches. There are many component-based techniques and methods, and even some claimed to be Component Oriented.

Nevertheless, they fail to be fully CO. Although this avenue may be crucially important, a complete orientation towards components is missing that suggests a paradigm shift in software development.

The metrics-based tools proposed in this dissertation claim to be instrumental in enabling software development strategies of the possible near future. The targeted orientation demands substantial improvement over existing software development practices. For the new paradigm, that is "Build by Integration," components only appeared as enabling technologies. Including many component related approaches, existing methodologies suggest different pre-coding activities for the similar goal: code development. The CO understanding supported in this dissertation, however, promotes development through integration of existing code components rather than code writing, which is almost denied as part of the lifecycle.

Software Engineering has evolved with duplicating the hard engineering discipline experiences in its infancy (such as through Waterfall Lifecycle). It progressed through peculiar practices owing to the different nature of the intangible software artifact. Before maturity, software practices are again turning back to hard engineering disciplines for possible exploitation of the proven successes. Component ideas, Software Product-line Engineering, and even Automated Software Factories are example concepts for such a trend.

Given the size and complexity of the software systems in an ever-expanding market of demands, it is already very difficult and it may soon become impossible to generate code, one line at a time for practical sizes that are tens of millions of lines. Other engineering disciplines are also moving to a higher-level of component-based integration due to the trend toward value-added chains where every company only contributes with its core-competency artifact – what it is best at. Considering the risky business of code development and also the fact that all kinds of algorithms have been coded before, the charm of re-inventing the wheel further reduces. Of course for some cases code can be developed; we did not discard teaching multiplication after the invention of hand-held calculators. Software

industry needs urgently to move forward for less risky and bigger product assembly in shorter time periods.

That is why it is believed that the near future for software engineering will be dominated with the Build by Integration paradigm. To comply, the component orientation approach that solely models components, their abstractions, and integration is adopted. This orientation is a new understanding that not only leverages on component technologies, but also introduces a more natural design cognition that could be applied without components. Components are handy because they make it possible to divide and conquer the problem definition on "structure" basis, being supporting technologies that are developed with the consideration of integration. Structure, being the most suitable one among the three fundamental design dimensions that are data, function, and structure, makes it easier to design bigger systems. The suitability is due to its being closest to a tangible nature, when compared to data or function: they correspond to pieces of code that is already functioning. So for complex problems now we have modules that play very well along divide and conquer strategy. COSE also suggests this decomposition should be hierarchical, based on Simon [113] design principles.

It is also possible to follow the COSE methods to develop code without components. Of course, more benefit is expected for the cases where fine specification of the code is not to be manipulated. A kind of an architectural look to the holistic view should be maintained where the system is modeled as a decomposition in a structural hierarchy. The procedural details of modules (that replaces components in this case) can be left to any existing approach: OO or even traditional. The immediate suspicion about the validity of such a paradigm shift could be concerning the nature of the new model: This way, systems have to be suitable for viewing them as looser coupled networks of "code islands." However, for complex systems such a view is a necessity, reminding us about the fundamental design principle: cohesion. Keeping in mind the real difficulty in traditional development being the integration, and even in modern approaches composing huge systems, rather than coding the integrals of a defined module, this new approach seems to be the answer.

Most of the component related methods can be classified to be placed in the "wiring level." In other words, relatively lower-level technologies are devised for the easier integration of components that are already defined for protocols that support even run-time integration. The missing view is the one that should guide the developers once a huge system is requested. There are very few academic studies that suggest components as an orientation rather than being OO and allowing components to be accommodated. They however, miss the holistic view, and the simplicity that comes with the persuasion that code is not to be developed. Unfortunately, there has been minimal improvement in the literature, after the introduction of the idea in 2003 due to the difficulties in testing the paradigm. A big software company has to accept to employ the methodology that is yet experimental, for a huge project. Also, a complete test can only be possible after the availability of a matured set of components in the application domain. However, it is hoped that the component technologies and the demand in the software industry will both develop in the direction that will enable a similar methodology to come to practical life.

The metrics and measurement mechanisms proposed in this dissertation are important because they support a radically different way of developing software. If the industry adopts Build by Integration, COSE related methodologies will be very important, together with many dimensions of a methodology, metrics and measurement being among the important ones. This study is being accompanied with other theses work in an effort toward defining different directions of such approaches. As a summary, the mechanisms will make it possible to estimate and measure various process and product properties related to software development, where a software system is viewed as:

1. A decomposition of structural abstractions,
2. Connections among abstract and physical components, and
3. Integration of physical components.

So far, existing metrics approaches have been proposed for function (traditional) and data (object oriented) centric software models only. Component related work

is still mostly Object Oriented and available metrics tools are a derivation of related OO techniques.

1.4 Outline of the Thesis

The rest of the thesis is organized as follows: Chapter 2 presents a survey of the literature of software engineering, software metrics, and metrics evaluation approaches. Chapter 3 introduces a layered approach for quantifying component oriented software system using metrics collected from the design documents of these systems. Chapter 4 describes a new set of CO complexity metrics properties. The metrics presented in chapter 3 are evaluated and validated in Chapter 4. Chapter 5 discusses the need for metrics programs automation. Also, an automatic metrics collection tool is introduced. Chapter 6 presents a summary of the most significant concluding remarks, comparison of the obtained results with results obtained in similar works, and potential extensions of the current study.

CHAPTER 2

BACKGORUND INFORMATION

2.1 Foundations of Software Engineering

Software engineering is the engineering discipline that focuses on methods, techniques, and procedures for building large and complex software system in a cost-effective manner [38, 68]. The term software engineering was first used in 1968 in a NATO conference that was held to discuss what was known as the software crisis [81]. The need for software engineering emerged after the introduction of computer systems embodying integrated circuits. Informal methods of software development developers were not enough to build large and complex systems and resulted in delayed deliveries and failed projects [68].

Work on software engineering methods has been progressing rapidly during the past three decades. During the 1970s and early 1980s software engineering research was intensified mainly on function-oriented methods. Those methods were mainly attempting to identify the system building block which was mainly functions. Among the earliest software engineering methods was the work of Dijkstra "Structured Programming" [33] where he defined the term *structured programming* and emphasized that well-structuredness of the code is as important as producing the correct answer and prevent errors. Parnas described the fundamentals of modular programming and introduced the concept of information hiding as the principle through which a system can be divided into modules [64]. In 1971 Niklaus Wirth described one of the earliest formal software development processes in his work "Program Development by Stepwise Refinement" [90].

Jackson described a method for program design based on data structures and program flows [96]. In 1978 Demarco provided a detailed methodology for structured programming (Structured design and structured analysis) [30]. While several other researches appeared after then, one of the most remarkable is the spiral model which forms the basis for evolutionary software development by performing risk analysis at each stage of the development and making use of software prototyping [14].

Object oriented methods started to appear late in the 1980s especially after the wide adoption of C++ (very widely used powerful object oriented programming language). Several OO methodologies have been developed and presented in published papers and/or books. The most widely used OO oriented methodologies include: Shlaer and Mellor [78, 79], Coad and Yourdon [28, 29], Wirfs-Brock et al [89], Grady Booch [16], IBM [47], Rumbaugh et al [70], and Jacobson OOSE methodology [48]. Lastly, and at around 1995 Jacobson unified his work with Booch and Rumbaugh and developed the unified modeling language (UML) which in 1997 became the standard object oriented methodology used everywhere.

2.2 Foundations of Software Measurement and Metrics

While some prefer to distinguish between the terms measure and metric, the terms are mostly used as synonyms in the literature. Following from that, the terms *software metrics* and *software measurement* are used interchangeably most of the time. In the early days of software development people used to argue whether it is necessary to measure software products or not. Nowadays the question has changed from whether to measure or not to *"how to measure?"* So, software measurement has become a fundamental activity of any software process.

Work on software metrics followed two different tracks. In one track metrics are estimated simply by directly counting some features or by performing simple arithmetic. In the other track information theory principles and mainly the concept of entropy are used for measuring software system complexity. In the following sections the mostly widely cited works in both tracks are briefly described.

2.2.1 Direct Counting Approach of Software Metrics

In the direct counting (simple) approach of software measurement, metrics that quantify some aspects of the software product like (size, complexity, connectivity, functionality, etc..) are estimated by counting some attributes or performing simple arithmetic. Following this principle, several metrics for traditional software systems, object oriented software systems, and component oriented systems have been proposed, evaluated, validated and practically applied and proved to be successful.

2.2.1.1 Metrics for Traditional Software

Early measures focused mainly on size of the product. Number of lines of code (LOC) may be considered as the earliest measure of software size or the earliest measure ever used for software systems. LOC, although being used very frequently and very easy to count, still has several drawbacks. Among them are: no single definition to what a line of code is; whether to count the number of executable statements or the number of physical lines. In some programming languages it is possible to have several executable statements in one single line; comments can also be included. Counting physical lines can easily lead to confusing results.

Halstead's work [41] is considered as one of the earliest researches that aimed at quantifying software system complexity. It formed a strong basis from which most of the research in software measurement was derived. Need for enhancements to what Halstead introduced are due to the advancements in software development approaches and paradigms i.e. the object-oriented and most recently the component oriented paradigms. Halstead identified aspects of software (software programs were mainly algorithmic based) that can be measured as:

- Number of distinct operators
- Number of Distinct operands

- Total occurrence count of operators
- Total occurrence count of operands
- Frequencies of occurrences of operators and operands

After obtaining estimates for the mentioned aspects, Halstead introduced formulas using these estimates to evaluate Program Length, Program Vocabulary, and Program Size. Using these, Halstead presented a method for estimating programming effort. Halstead's work also has its reflections to modular decomposition process.

McCabe presented another striking effort in software complexity evaluation by introducing the *Cyclometic complexity* measure [57]. Cyclometic complexity has been and still is a very important means for evaluating complexities of software artifacts. McCabe suggests a graphical representation of the program and then estimates program complexity as the number of linearly independent cycles in the graph. *Cyclometic complexity* is calculated as:

$$V(G) = e - n + p$$

Where e is the number of edges, n is the number of nodes and p is the number of disjoint graphs. The main benefit of cyclometic complexity number is to determine the number of distinct paths in an algorithm graph representation which is used to determine the number of test cases to be used.

Researchers working in the field of software measurement have focused on measuring the degree of interactions between different system components and derive relationships between the values of interconnectedness and other product and process aspects like maintenance effort, testing effort, development cost, defect density, and other important product and process features of interest. One of the earliest works in this field was that of Henry and Kafura [46]. They developed measures to assess the degree of interactions between software system modules. One of the important features of these measures is that they can be obtained early at the design stage when it is possible to determine problematic modules before implementation and redesign them. The two basic measures introduced are *fan-in*

15

and *fan-out*. *Fan-in* of a procedure (a procedure is the fundamental module in Henry and Kafura approach due to the fact that structured programming languages in which procedure is the fundamental decomposition entity) *is the number of local flows into a procedure "A" plus the number of data structures from which procedure "A" retrieves information.* Fan-out of a procedure *is the number of local flows from the procedure plus the number of data structures which the procedure updates.*

From fan-in and fan-out estimates, a measure of procedure complexity can be obtained as:

Length * (*fan-in* * *fan-out*)^2 where length is the number of lines of code of a procedure.

Albrecht and Gaffney introduced the function points (FP) [4] measure of software functionality which is independent of the programming language used (FP count is interpreted as a measure of size by some researchers in the field). FP's can be estimated by counting the number external inputs and outputs, number of user interactions, number of internal files, and number of external interfaces. After counting these attributes, a weighing process is carried out for each item. The weighting factor values vary from 3 to 15 depending on the degree of the complexity of the weighted item. Items are considered to having simple, average, or complex weights. Then, items are multiplied by their weighing factor.   A single complexity value can be obtained from a specific combination of these counts. The initial estimate of function points produced the so-called unadjusted function points (UFP). UFP can further be modified by considering other attributes of the system. Adjusting function points takes into consideration attributes like performance, distribution, reuse, and some other factors as well.

2.2.1.2 Metrics for Object Oriented Software

The decade of 1980's witnessed the real birth and wide adoption of the object-oriented software development paradigm. Due to the new concepts and units of abstractions, the object-oriented paradigm required a different approach towards

16

metrics as well as it required a different approach of problem decomposition and integration. One of the earliest and widely accepted object-oriented software complexity measures was the metrics set introduced by Chidamber and Kemerer in their work described in [20 and 21]. The metrics set later started to be known as the CK metrics set named after the developers initials. The CK metrics set defines six different metrics that give numerical estimations of different features of the *class* and *class interactions*. These metrics and their definitions as given in the original papers [20, 21] are:

Weighted Methods Per Class (WMC): The sum of the complexities of all methods of a class.

Depth of Inheritance Tree (DIT): The maximum length from the node where the class is in the inheritance hierarchy, to the root.

Number Of Children (NOC): Number of immediate subclasses subordinate to a class in the class hierarchy.

Coupling Between Object classes (CBO): the count of the number of classes to which a class is coupled.

Response For a Class (RFC): the set of methods in the class plus the set of methods called from the methods of that class.

Lack of Cohesion in Methods (LCOM): the count of the " method pairs" whose similarity is 0 minus those whose similarity is not 0.

CK metrics set is not free of criticism. The most widely argued metric from the set is the LCOM metric where no interpretation explanations are given to the possible negative values that can be obtained. On the other hand, it is important to admit that the CK metrics remain the most widely used and referenced object-oriented design metrics. Recently, several automated collection tools of CK metrics have been implemented and commercially used.

Several researches tackled CK metrics to detect their benefits from managerial and technical perspectives. Subramanyam and Krishnan [82] considered a subset of CK metrics (WMC, DIT, CBO). The outcomes they obtained revealed that a high correlation was found between these metrics values and defect rates found during acceptance testing. Chidamber et al. [24] found relationships between the CK

metrics values and *Productivity, Design Effort and Rework Effort*. High levels of coupling and low levels of cohesion were associated with low productivity, greater rework, and greater design effort. The CK metrics have been empirically evaluated to detect whether they have any power in discovering error proneness classes. The results obtained were of interest to those who believe in metrics as quality indicators. High error rates were associated with high WMC, DIT, CBO, and RFC values. High values of NOC led to low probability of fault detection. Although cohesion is a deemed design feature, LCOM appeared to be insignificant. This can be attributed to the definition of the LCOM metric. Basili et al. [10] suggest that CK metrics, in general, can be used as good indicators of fault proneness.

Encapsulation and polymorphism are two among the very important object-oriented principles. Encapsulation and Polymorphism measures have not been considered in CK metrics set. Pons et al. [66] tackled polymorphism in object-oriented systems. They introduced three definitions for three different levels of polymorphism as follows:

Polymorphic methods: if they have same name and same signature.

Polymorphic Classes: if they define the same polymorphic methods.

Polymorphic hierarchies: if all of its classes share a core interface where a core interface is a set of polymorphic methods.

The interesting outcome from this work is that the higher degrees of polymorphism were associated with higher degrees of readability, extensibility, and maintainability.

Another widely discussed object-oriented metrics set is the MOOD set [2]. The MOOD set introduced six metrics to measure aspects of inheritance, encapsulation, and coupling. The MOOD set has been tackled by several researchers later, namely the work of Harrison et al. [42] where they showed that the MOOD set can be used as an aid in the management process of software systems and can give an overall assessment of the system. They also suggest that the MOOD set can work efficiently at the system level and can be applied complementary to CK metrics

18

which are more efficient at class level. The MOOD set has been theoretically evaluated and empirically validated to be of valuable managerial use.

Chen and Lu [19] introduced object-oriented metrics to measure complexities of operations, arguments, classes, and class interactions (couplings), and class hierarchies. Chen and Lu stated that it is possible to obtain very different regression models based on data from different data sets.

2.2.1.3 Metrics for CO Software

We have seen earlier that the CO software development requires a new approach towards development. Due to that, it also requires a new approach towards measurement. This is a natural consequence since new concepts are introduced and system building blocks have changed. The principal unit of abstraction is the *component* rather than the *class* in the OO approach and *function* in the traditional approach. Components provide services through their interfaces. Several components may communicate to provide some service(s). Metrics for CO systems should mainly focus on the communications between components [76]. Several challenges face researches in the field of CO metrics. One of the most important challenges is the unavailability of source code to examine and use in metrics validation. Lack of experimental data makes the process of developing and validating metrics for CO oriented systems a difficult task to achieve [76].

A review of the metrics literature reveals that very little serious CO metrics existed before. That is of course due to the fact that CO software development is relatively new. Also, in all other approaches (Traditional and OO), first development methods and methodologies are defined then metrics are presented accordingly.

2.2.2   Information Theory Based Software Metrics

Entropy is the fundamental concept of information theory that attracted researches in the field of software measurement. In communication systems *Entropy*

corresponds to the relative degree of randomness. The higher the entropy value, the higher the possibility of errors in the system. Shannon and Weaver found that entropy of a system is usually related to and evaluated based on the information content of that system [77].

The information theory-based approach or, as it is mostly called, the entropy-based approach of software metrics tried to benefit from the definition of entropy (the degree of uncertainty) to quantify some aspects of software products. This approach did not receive interest as much as the simple (direct counting) approach. Also, most of the proposed entropy-based metrics sets have not been empirically validated. In industry practices entropy-based metrics do not have a significant contribution as well. The most widely known entropy based metrics for traditional software, object oriented software and component oriented software are briefly outlined in the following paragraphs.

Entropy and amount of information in a communication system can be defined as follows: Let X be a discrete random variable taking a finite number of possible values x_1, x_2, .. , x_n with probabilities $p_1, p_2, ..., p_n$ respectively such that $p_i \geq 0$, $i = 1,2, ..., n$, and $\sum_n p_i = 1$. We attempt to arrive at a number that will measure the amount of uncertainty and it is obtained as:

$$H_n(p_1, p_2, ..., p_n) = \sum_{i=1}^{n} p_i h(p_i)$$ Where $h(p_i)$ is the entropy of x_i with probability p_i

Thus $H_n(p_1, p_2, ..., p_n)$ is the average uncertainty removed by revealing the value of X. This definition of entropy has been applied in software measurement mainly to obtain a numerical estimation of the average information content of a software module. Also, entropy-based metrics have been used to measure the flow of information between system modules/components and overall system complexity.

One of the earliest attempts to obtain measures of some software aspects using entropy is presented by Hellerman [45]. Hellerman described an entropy-based estimation of the computational work of a *boolean transformation*. Hellerman's

measures may be used to compare the advantages of several alternatives of a process implementation.

Allen et al. [5] developed measures for inter-module coupling, intra-module coupling, and the degree of cohesiveness of a module. All of those measures are based on the information content of the module. The metrics were evaluated using coupling and cohesion metrics properties described in by Briand et al. [17]. The metrics have been empirically validated using industrial projects data. The results of the validation revealed that these entropy-based metrics are finer grained relative to similar normal counting based metrics.

Harrison [43] presented information theory based estimation of program complexity where the text of a program is considered as a message that is mainly obtained by observing occurrences of special operators. Harrison stated that "*complexity of a program is inversely proportional to its information content*". The results obtained by Harrison have demonstrated some practical power and have been tested on commercial applications. The results of applying Harrison metrics revealed that information content of a program is related to error frequency.

Ned Chapin developed an entropy-based metric that measures the complexities of interactions in COTS based systems and focuses mainly on messages flowing in and out of the system [18]. A similar work is presented in [51] which describe entropy based measures of size, length, complexity, coupling, and cohesion.

Abd-El-Hafiz presented an approach for deriving entropy-based software complexity measures [1]. Her approach focuses on function calls (in procedural languages) or method invocations (in object-oriented languages). Abd-El-Hafiz suggests that a system s can be represented as a set of elements E and Relationships R such that for any E_m in E, and R_m in R, then $<E_m, R_m>$ is a module of s. Empirical validation of the metrics and their effects on understandability, maintainability, and reliability was left as an open research problem.

One important contradictory point related to entropy based metrics is the interpretations of the relationship between the terms complexity and entropy.

21

While some findings relate increased complexity to increased entropy, other works found that complexity is inversely proportional to entropy.

2.3 Metrics Evaluation and Validation Approaches

Metrics that are developed are of little value unless they are validated and examined against measurement theory rules and principles. Also metrics should be validated with real projects to check whether they meet the initial assumption of their development. Different approaches to evaluate and validate metrics have been described in the literature. We briefly describe the most widely cited works in this respect.

Blundell et al. [12] argue that software metrics so far has failed to precisely evaluate software quality due to: 1) Measured attributes are not clearly identified, 2) metrics are created before examining their relevance, and 3) metrics are not objectively validated.

Alsharif et al. [6] stated the main objective as: Inter-module complexity resulting from interactions between system models should not be larger than that of the original problem complexity before decomposition. Basili et al emphasize [10] that it is important to note that not every theoretically correct metric will have practical relevance to the problem in hand.

Briand et al. [17] suggest that the first step in developing software metrics programs is identifying classes of software characterization measures. The authors defended that most of the inconsistencies and incomplete works in software measurement field are due to the different understandings and interpretations of the terms that are frequently handled like; size, complexity, cohesion, coupling, etc. They proposed a specific set of properties against which the related concepts can be evaluated. Each concept is evaluated against its property set. Size is evaluated against size properties; complexity is evaluated against complexity properties, and so on. The authors considered as examples, several previously developed and widely known measures like Halstead's metrics [41] where they suggested that

22

length and size [41] of a program are measures that fall in two different categries. While size is additive, length of a program is not. The *cyclometic complexity* [57] was evaluated against complexity measure properties and failed to satisfy all properties of complexity measures as they are defined in [17]. CK [21, 22] metrics also were evaluated and found not to be complexity metrics. CBO metric of CK satisfies the properties of a coupling measure and RFC metric satisfies the properties of size and coupling measure properties.

Poels and Dedene [65] wrote some comments on [17]. The first of their critisizms is that Briand et al. did not state that their properties of measure are enough to validate. Second, some more properties are needed to be identified for different attributes. Third, the definitions of the additivity and connectivity properties are inconsistent and have some contradictions.

Mendonça and Basili [58] show that a good measurement framework is one that measures all the software aspects needed to achieve the user goals consistently, and measures only what is needed but not more. They identified the key components in any measurement framework as: metrics and attributes, data, users of data, and usage of data. They also suggest the use of GQM paradigm [11] to achieve these purposes which can be summarized as follows:
- Define goals
- Refine goals a set of questions that can be measured
- Find metrics implied by questions.

Kitchenham et al. [53] presented an embracing work towards developing validation approaches of software metrics. The authors claim that research in software engineering lacks formality and compared to other engineering disciplines software engineering is still immature. The measurement framework proposed focuses on identifying the aspects of a software system to measure and the properties of these aspects, defining these aspects while developing measures, and lastly identifying the validation scheme to be applied. The validation framework steps can be summarized as follows:
- Identify entities, attributes and their relationships.

23

- Identify units, scale types and their relationships. Distinguish between compound and scalar units.
- Identify values (numerical or not). Permissible and not permissible values
- Identify measurement instrument and calibrate it.
- Identify measurement protocols. Where a protocol must enable us measure a specific attribute on a specific entity consistently and repeatedly.
- Distinguish between direct and indirect measures.

Validation of a metrics program means proving that all items listed above are valid. Morasca et al [61] have strictly criticized the paper as misinterpreting Weyuker's properties [88].

Kitchenham, Pfleeger, and Fenton [55] partially accepted that they did a mistake in evaluating Weyuker's properties but insisted that Weyuker properties can not be satisfied simultaneously by any useful measure and Weyuker properties 5 and 6 are not relevant to a single view of complexity. In another work Kitchenham et al [56] proposed a set of valuable guidelines that researcher working in the field of empirical software engineering can follow to empower their research and validate their results.

Weyuker [88] presented nine properties a complexity metric must posses in order to be considered as a good complexity metric. Weyuker properties have been used by several researchers [21, 71] and others as the main validation criteria of their metrics. Weyuker's properties have also been criticized by Kitchenham et al [55] as not being relevant to a single view of complexity. Zuse [93] claims that two of Weyuker's properties are inconsistent.

Tian and Zelkowitz suggest that a measure must compare between a component and a composite program [84]. The interesting outcome here is the authors' claim that complexity of a software system can be less than the complexity of any of its components [84]. This outcome is interesting since it almost violates the majority of software complexity views observed in the literature.

Zelkowitz and Wallace [91 and 92] stressed that data collection is the key activity in software experimentation. They also suggested experimentation methods can be grouped into three classes as: *Observational* where data is collected as the project develops *Historical* which depends on data from projects that have been completed, *and Controlled* provide for multiple instances of an observation to statistically validate the results. The authors surveyed all papers published in *IEEE Transactions on Software Engineering* for the years 1985, 1990, and 1995 and found that all the papers that exhibited experimentation followed one of the presented methods.

Schneidewind [72] suggested that a metric is valid if its values can be shown/have been shown to be statistically associated with some corresponding quality factor. Schneidewind described an approach for relating metrics validation to quality functions. Quality can be controlled by metrics if the metrics have discriminative power and is capable of tracking changes. Also, quality can be controlled by metrics if metrics have the predictability property. Repeatability property is necessary for any metric to be used in any quality function. A metric can be valid if we can establish a statistical relationship between that metric and some quality factor and make sure that the metric provides a correct estimate of the intended attribute [73].

Fenton introduced the necessary basis for measurement in software engineering, guidelines and rules to follow, and tips to avoid [36]. There are two types of measurement: *Direct* and *indirect*. While direct measurement of an attribute does not depend on the measurement of any other attribute, indirect measurement involves the measurement of one or more other attributes. Measurement can be used for both assessing the software quality and predict its future behavior. The first thing we need to do in a measurement program is setting our objectives; why we measure? To assess or predict! What attributes should we measure? To answer the later question we need to first identify our entities and their attributes. Then, we need to determine how to signal an attribute as measured. We also have to keep in our mind that there is no single number to characterize every aspect of quality.

Zuse has presented the foundation of object oriented measures properties [94] and evaluated CK [21, 22] against these properties.

Kitchenham [52] performed an experiment to examine the validity of structural metrics *fan-in* and *fan-out* [46] from a practical perspective and detect whether they can predict change-prone and error-prone modules at early stages of the development. The results of the experiment revealed that these metrics are not good quality predictors but are good to use for project control activities.

Clark presented eight issues and identified them as the secrets in software measurement [23]. The most important of these are: we have to make well-use of data coming from measurement activities; we need to know that applying metrics require cultural change to the organization since people may resist metric application, and variability in data provides a powerful decision tool.

IEEE standard [98] for software quality metrics methodology outlines the steps of a software metric program as:
- Establishment of software quality requirements
- Identification of software metrics to be used
- Implementation of metrics
- Results analysis
- Validation of metrics: Do the empirical results coincide with the initial assumptions? It is not necessary to obtain universally validated metrics.

2.4 Different Views of Software Complexity

Software complexity has been interpreted in completely different manners by different authors. While some related complexity to size others related complexity to understandability and readability. According to the view of complexity described by Briand et al [17] C*yclomatic complexity* [57] is not a complexity measure, the *fan-in* and *fan-out* measure of [46] is a complexity measure and all of CK [21 and 22] metrics are not complexity measures.

Tian and Zelkowitz [84] considered software complexity as the aspect of software that is used to predict external properties of the program (reliability, understandability, maintainability) using internal measures like *cyclomatic complexity* [57] or Halstead's measures [41]. They also suggest that complexity of software is measured to make choices between functionally equivalent solutions [84].

Almost everyone involved in the software process agrees that software complexity must be managed to ensure the development of efficient and cost effective software systems. The main problem in managing software complexity is the existence of too many different interpretations of the term complexity. Mainly, "Divide and conquer" is the strategy that is followed by software developers to manage complexity [30, 33, 50, 64, and 90]. Alsharif et al. [6] introduced a method for evaluating the complexity of a module, inter-module complexity, and the complexity of the whole system. Although there is no consensus on what software complexity means it is generally accepted that decomposition reduces complexity. New complexity will be a result of the inter-module connections.

It is globally accepted that decomposition, without going into the details of how to decompose, is the means for well-controlled complexity. We will present a summary of the different views of software complexity as they appear in the literature.

Zuse considers complexity of software as some measure of the mental effort required to understand that software [93]. According to Zuse, the complexity of a system design can be estimated as a function of the relationships among all of the external interfaces of the product. Complexity of architecture is a function of the relationships among subsystems and complexity of a module is a function of the relationships/connections among program instructions [93].

Visaggio introduced a layered approach to defining software complexity. Visaggio described three levels of software complexity [86] and defines *internal complexity* as the degree of difficulty of understanding the system through its code, *intrinsic*

complexity as the degree of interconnectedness, and variety of implemented aspects, and *external complexity* as the relative difficulty of understanding a program with the availability of its documentation.

The principal tool for managing complexity is hierarchical decomposition and then complexity will be a function of the number of modules in each level of the hierarchy, number of levels, number of interfaces, and number of interconnections [50]. Keating also provided some guidelines regarding the number of modules in one level and stated that this number should be 7 ± 2 since human beings can concentrate on 7 ± 2 chunks of information at the same time [60]. We can relate this rule to software complexity and develop estimations of software complexity resulting from interactions of system modules by benefiting from this rule.

2.5 Summary

The survey of the literature presented above can be summarized in two classes of outcomes. The first class presents the current state-of-art which can be summarized as follows:

1- Component Oriented Software Development is believed to reduce development costs and lead to the construction of more efficient and reliable software products.

2- Software Measurement is a necessary practice in order to efficiently control, manage, and contrast software products, projects, and processes.

3- The principal factor to the success of any measurement program is the availability of a good set of metrics.

4- Metrics are of little value unless they are validated against accepted and proved to be correct set of attributes and properties.

5- Collecting metrics from software designs or source codes can be a costly process.

6- Metrics programs can gain more importance if metrics results are related to critical factors of software quality like: maintainability, reliability,

performance, and process features such as design effort, development effort, integration effort, etc.

7- Early estimations of metrics can lead to early detection of defect-prone components which will in turn lead to reduction in maintenance costs.

8- Metrics collection must be a cost-effective process. If collecting metrics from software designs or source codes will cost too much then no one will be encouraged to use them.

The second set of outcomes describe the steps that need to be performed. These points mainly focus on issues related to CO paradigm and can be summarized as follows:

1- Attempts to provide measurement frameworks for component oriented software systems do not have real existence.

2- Serious component anatomy to extract the quantitative aspects and quality determinating attributes in a software component are not available.

3- A specialized method describing the validation criteria for component oriented software metrics are not available yet.

4- Works trying to investigate the relationships between component oriented complexity metrics and reliability, maintainability, and development effort did not reach the community's satisfaction level so far.

5- Dedicated CO software development CASE tools are still in trial phase or are research projects in academic institutions.

6- Automatic metrics collection tools for component oriented software engineering have not been encountered.

CHAPTER 3

QUANTIFYING THE COMPLEXITY OF COMPONENT ORIENTED SYSTEMS

The importance of measuring software and, particularly, software complexity has been emphasized in details in chapter 1. We have seen clearly in Chapter 2 that there is a lack of research on measurement methods for CO systems. Description of measurement frameworks is one of the most important aspects to have a mature software development process.

In a previous research we extended CK metrics [21, 22] to component-oriented models and presented that in [71]. A set of properties for verifying and validating component oriented metrics have been described in [104]. In another work the relatedness of a subset of component oriented complexity described in [71] metrics with design and correction efforts was explored and a direct relationship was found [97, 103]. The same research also revealed that more research is still necessary in the field. In this research we carry out a detailed analysis of component oriented measurable features and metrics quantifying them. The research is divided into three phases:

1) Identification of the most significant features of components to be quantified. These features are identified based on their intuitive power in predicting some process related aspect(s).

2) Metrics quantifying these aspects are defined. Initial viewpoints about the potential impact of the metrics values on the process and/or the product are presented. These viewpoints are presented based mainly on intuition,

experience, and results appeared in other researches that considered other metrics sets mainly for OO systems.

3) Extended evaluation and validation schemes are performed. To evaluate metrics we used two sets of properties of complexity metrics. To validate metrics, we used empirical data collected from graduate students projects.

3.1 A Glance on the Terminology

The terms used in this paper have been widely used by researchers in computer science. Yet, the terms have been used to mean different things by different authors. For example, the term software component, which is a fundamental term in our research, is very widely used in software development cycles; the term has several views and these different views are sometimes used interchangeably and in a confusing manner. The term, viewed from an Object-oriented point of view is used interchangeably with the term object (an instance of a class in object oriented programming). The term component is used interchangeably with the term module in modular programming environments (Modula-3). Little background in computer science and particularly in programming paradigms lets someone know that the terms module and object are too different constructs. In our research we are going to use the term component as to what it means in the component oriented view which is a third view different from both module and object in modular programming and object oriented programming respectively. Besides having some new features, the component oriented view of a "component" captures some of its features from the object oriented view and some features from the modular view. Not only the term component, but other terms of interest like component orientation, component oriented systems, and component oriented modeling languages, are not defined in a standardized manner in the industry and academic practices. So, before any other step in our research, we are going to provide definitions and introduce the component oriented view of these terms with major reference to the work [23] which represent the pioneering research in the field.

- *Component Orientation*: A new software development paradigm. It focuses on development by integrating already available components rather than writing

from scratch [34]. The first step in system development is specifying the structural decomposition of the system where components, components' hierarchies, and intercmoponent relationships are defined. In Component orientation, composition is the principal means for building large systems. A component is mainly viewed as a black-box which can be accessed only through its interfaces. In the rest of the paper we try to be loyal to the process model described in [34] for a component oriented system. Also the notation used is from the language COSEML which is also described there. Some basic attributes are the as follows:

- *Component:* A Unit of independent deployment. A component builds upon encapsulation, and polymorphism where "Complex" components can be obtained through composition [83]. A component can implement several interfaces, each abstracting a specific service. Components functionality is implemented in methods and is provided through the interfaces only which can be considered as the component's access points. Figure 3.1.a describes the notation used in COSEML [34] for a component with single interface and Figure 3.1.b is the notation used for a component with multiple interfaces.

<div style="display: flex; justify-content: space-around;">

Component

Interface1 ●
Interface2 ●
  ⋮

Figure 3.1.a: A Component with Multiple Interfaces

Component

| Properties |
| Methods In |
| Methods Out |
| Events In |
| Events Out |
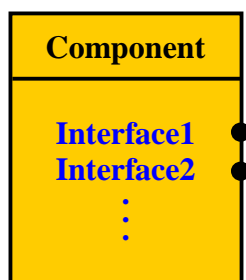
Figure 3.1.b: A Component with A Single Interface

</div>

- *Interface:* Interfaces are components access points. Components' services are presented through interfaces. An interface is generally an abstraction of a service. A component may implement single or multiple interfaces. Besides

32

properties and In/Out methods, an interface can include lists of In/Out Events. An output method is actually a request, and an input method is a service. The notation used for an interface in COSEML is shown in Figure 3.2.

Figure 3.2: An Interface

-   *COSEML:* A dedicated CO modeling language. Being a dedicated CO modeling tool was the main reason behind its demand. COSEML presents three types of entities: abstract components (Package, Data, Control, Function, and Connector), physical or implemented components (Component, and interface), and connections (Connector, Inheritance, Composition, Method Link, Event Link, and Represents). The COSEML notations used for abstract components are presented in Figure 3.3, physical components are a component with single interface (see Figure 3.1.a), a component with multiple interfaces (see Figure 3.1.b), or an interface (see Figure 3.2). Optional symbols can be used for connectors to add more clarity on the type of the connection. The different optional symbols that COSEML support are presented in Table 3.1.
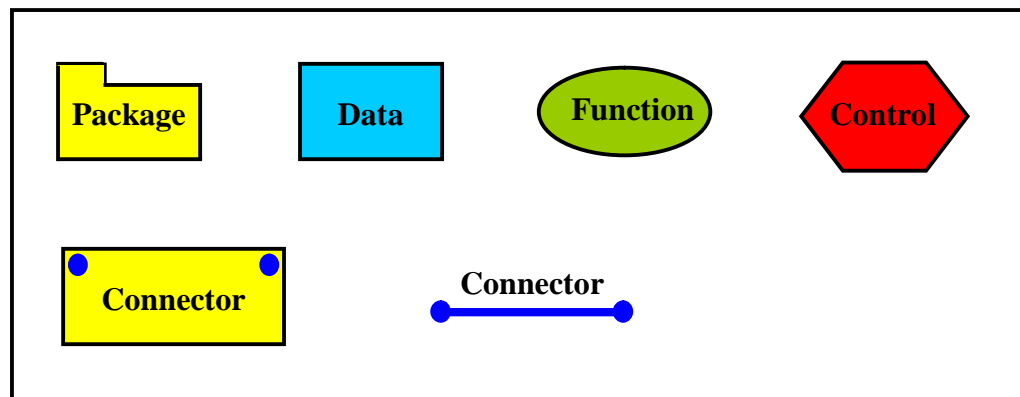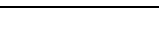
Figure 3.3: Notation Used for Abstractions in COSEML

33

Table 3.1: Optional Connector Symbols in COSEML

Link Symbol	Link Name
◆	Composition
▶	Inheritance
→	Method Link
→	Event Link
—	Represents

- *Component Oriented System:* A component oriented software system is a software system that is developed based on a component oriented process model (e.g. CO process model presented in [34]) where the development process comprises the steps:

  • Software system specification is performed; services and boundaries are identified.

  • Specifying the structural decomposition of the system which comprises building decomposition hierarchy.

  • The specifications of system components are prepared. This step may lead to creating components from the scratch, search for already available components, or adapt some ready-made components to match the specifications in the system.

  • The last step which comprises the integration of the components that are specified and implemented in steps 2 and 3.

Figure 3.4 depicts a simplified university information system design that is created using COSEML media.
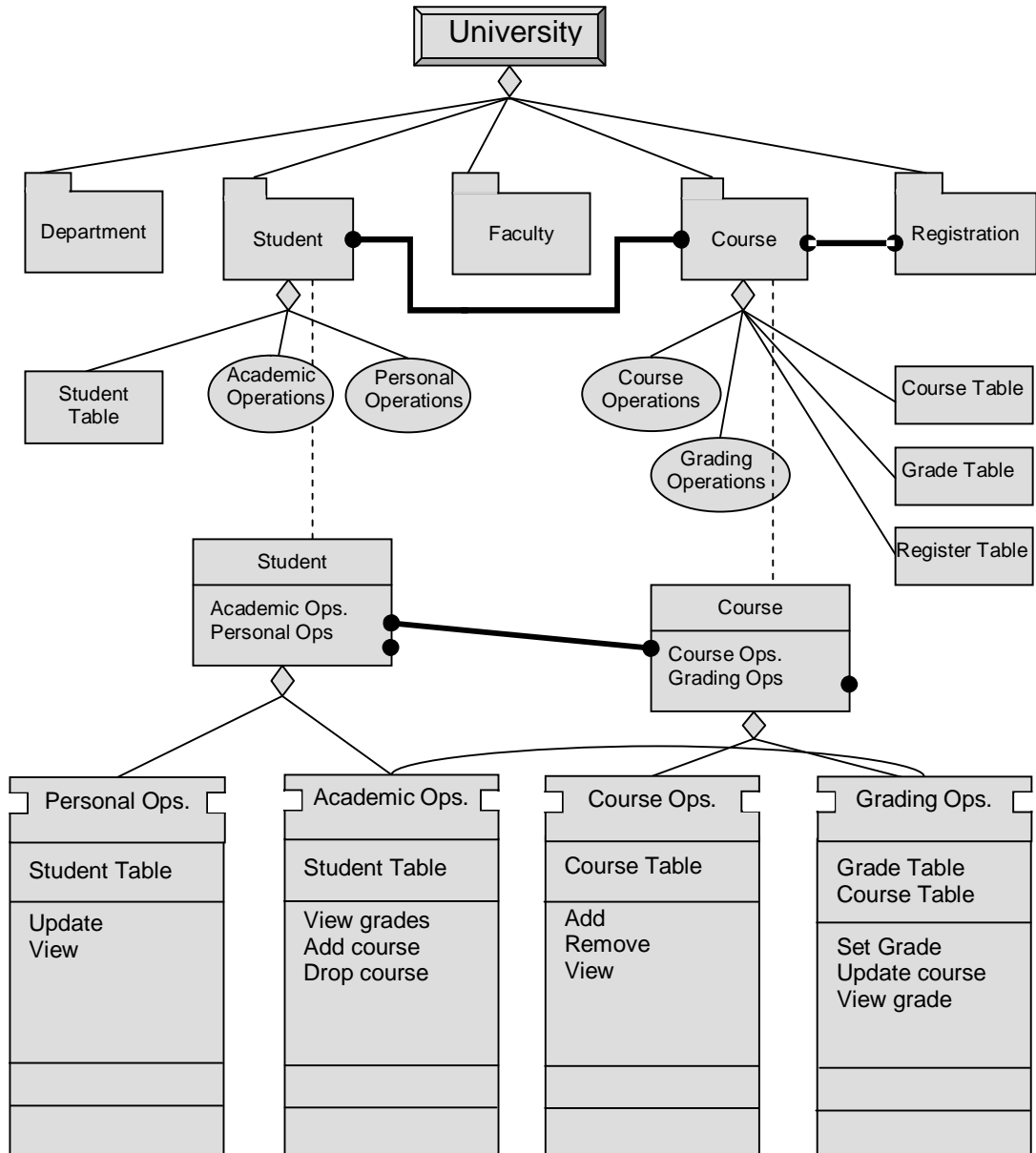
Figure 3.4: Simplified University System Prepared in COSEML

- *Complexity*: In section 2.4 different views of complexity from the literature have been discussed. Most of complexity views relate it to the lack of structure in software systems, difficulty to comprehend, to maintain, to test, etc. [8 and 40]. Others view complexity as the factor associated with higher probability of defects. The IEEE standard [98] defines component or system complexity degree to which the design or implementation is difficult to understand and verify. The IEEE view of complexity is contrasted with simplicity which is the

35

degree to which a component or a system design is straightforward and easy to understand. Our view of CO system complexity is not very different from these views. In our view of complexity, CO software complexity is the aspect that is related to the difficulty to understand, and then will increase design, correction, integration, and maintenance costs of the system. Also, our view of complexity suggests that complexity is a composite aspect that is evaluated from different independent attributes that can be quantified from the system design models. Thus complexity has a direct impact on overall quality of the system.

3.2 Defining the Steps of Our Approach

The steps in our measurement framework for CO software systems include the following activities:

1- Identification of measurable product aspects.
2- Deriving metrics that can appropriately characterize the different aspects to be measured.
3- Collecting Data that is needed to derive metrics and validate them.
4- Interpretation of the results.
5- Providing feedback according to the obtained results.

Our aim is to characterize software attributes which individually or collectively affect complexity. Pressman [68] outlines the most important metrics that can be collected during and after the design phase as:

1- metrics for characterizing architectural quality,
2- complexity of system building elements (components), and
3- characteristics of components and their interaction characteristics.

Earlier researches in the field suggest that the existence of a single metric that can characterize the overall system complexity seems to be impossible [36, 99, 100]. Smith also has a similar argument about computer performance and suggests that a single number to characterize computer performance can be misleading [80]. In our research, though we believe that CO system complexity is a multidimensional feature we are examining the possibility to come up with a single compound measure that can characterize component oriented software complexity. To make it

more clear, lets consider, as an example, the volume of a rectangular prism is dependent on height, width, and length values of that rectangular prism but it s still a single value that characterizes volume. A change to any of these values will result in a change to the volume of the rectangular prism. We still believe that a single value that characterizes CO oriented system complexity obtained from the combination of several related values is still a very useful metric.

3.3 CO Software Systems Quantifiable Aspects

In the component oriented paradigm main focus is on system structure [34]. Due to that, while requiring internal complexities of components, more attention will be paid to the system's overall structural complexity. The first question that needs an answer is: what attributes of a CO system characterize its structural complexity? In finding answer to this question we will first explore the attributes that are known/believed to be related to system's structural complexity. Our complexity analysis will focus on features that characterize system's structural complexity, components' internal complexity, and interfaces complexity.

1- System Structural Complexity: Software system structure is defined as the way through which system building elements are organized with respect to each other and with respect to their surroundings [39]. Software Architecture deals with methods that can be applied to the structure to achieve maximized reusability and reliability [24, 25, and 100]. Software structure is a design decision: Two or more different design alternatives may result in multiple structures. Measuring the degree of structuredness in software systems is an important issue since system organization has its impact on maintainability [86]. Also, it is intuitively clear that different structures of the same system will certainly lead to different values of structural complexity. Clements et al [24] emphasized the importance of evaluating software architecture at early stages of development. They noted that evaluation of software architecture, besides not being the only factor, plays an important role in evaluating the overall system quality. Depending on the definitions of system structure described in [24, 39] and building on our definition of a component oriented system (see section 3.1) one can notice that a CO system structure is a function of that

system's Components, Connectors, and the Composition Tree. Below we will define attribute metrics characterizing them. For each metric our initial viewpoints about the potential impact on structural complexity, are also included:

1.1 Depth of Composition Tree (DCT): Count of the number of distinct level of the composition tree. Our selection of this attribute is based on the following initial viewpoints:

   a. The deeper the composition tree the better the system decomposition is. Higher values of DCT are an indication that system components are more specific and may have higher potentials for inter-system reuse.

   b. The deeper the DCT the more components we have. Components, at levels closer to the root of the tree tend to be having many sub-components making them more difficult to compose and test.

1.2 Width of Composition Tree (WCT): There is usually a trade-off between width and depth of the composition tree at the level close to the root of the decomposition tree. While deeper trees may lead to more integration effort, favoring wider trees will result in less integration effort but may decrease chances of inter-system component reusability. In Figures 3.5.a and 3.5.b, two decomposition alternatives for the same system are described. For the model shown in Figure 3.5.a we need eight time units to integrate (assuming equal times for integrating different components) all the components, while in the model shown in Figure 3.5.b we need 14 time units. On the other hand, we have 8 reusable components in the alternative shown in Figure 3.5.a while we have 14 reusable components in the alternative shown in Figure 3.5.b. The trade-off between reusability and effort is clear.
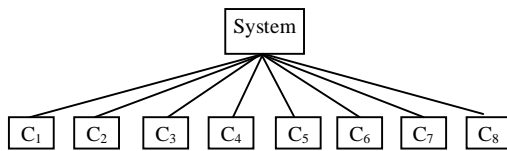
Figure 3.5.a: High WCT
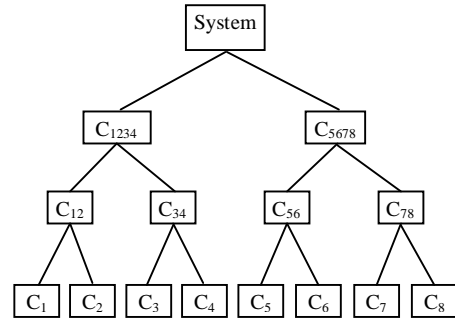
Figure 3.5.b: High DCT

1.3 Coupling Between Components: The degree of interdependence between software modules [98]. In CO systems coupling is directly affected by the degree of connectivity between system components. Two components C1 and C2 are coupled if there is a connector linking these components with each other. At the system level coupling between system components is estimated by counting the number of connections between system components. For the sample model shown in Figure 3.6 the coupling value is equal to 15. Arrow directions indicate service requests. Two different metrics are defined to characterize coupling between components. The first metric is the count Total Number of Connectors (abstract connectors plus messages). This metric is a characterization of the overall system complexity. The second metrics is Average Number of Connectors per Component. This metric characterizes the potential impact of inter-component dependencies on the overall structural complexity of the system.
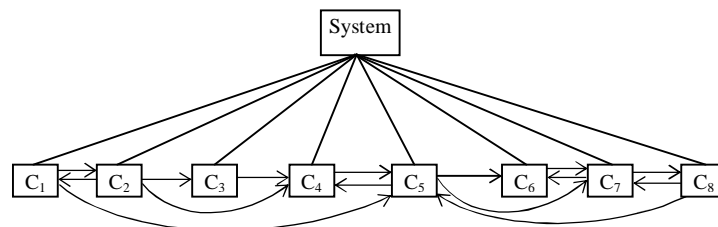
Figure 3.6: Coupled Components

1.4 Cohesion of System Components: The degree of cohesion of a system is usually measured from the degree of relatedness of that system's building elements (components). Higher cohesion is a deemed feature of system designs. It is important to note here that there is usually a trade-off between coupling and cohesion values in a system design. While zero coupling is impossible, very low coupling can be as bad as excessive coupling and very high cohesion associated with very low coupling may lead to undesirable results [99]. Measuring the degree of cohesion in the system requires knowing which component presents services that are related to the overall system functionality. Components are accessed through their interfaces. So, if a component implements interfaces which are never contacted by other components in the system may be an indication of low cohesion. The number of interfaces which have a fan-in value of zero indicates the lack of cohesion in the system. In a typical system all implemented interfaces should be used by other components, hence the best cohesion is in the case where number of interfaces with zero fan-in is zero interfaces. Taking the average of the number of used interfaces to the number of total interfaces may provide an insight about the degree of cohesion in a system. An average value of 0.8 means that 80% of interfaces provided are accessed by other components of the system. For sure, high rate of unused services of the system indicates that extra costs are paid for unneeded functionality. It is just like adding and extra cost on a mobile phone for the service of using it like a joystick. A service which is rarely needed by users.

1.5 Total Number of Interfaces (TNI): In CO system interfaces play the important role of being components' access points. Number of interfaces per component is an indication of the amount and diversity of functionality delivered by the components. Components of a system exchange services through their interfaces. Also interfaces are the main means of integrating components with each other [8].

Viewpoints:

a. Increased number of interfaces implies a wide set of services since an interface usually presents one category of services for a component. This may limit the possibilities of a component reuse among systems as also, diversified interfaces indicate specialized connections.

b. Increased number of interfaces implies increased fan-in value for a component which means that it is supposed to be highly dependable; it should be designed, implemented, and tested with a lot of care. A bug that may exist in such a component may be cascaded to several other dependent components.

c. On the other hand, increased number of interfaces of a components could mean cleared service descriptions and then would lead to less effort to integrate with other components.

1.6 Total Number of Methods (TNM): The count of the total number of methods in the system. Components implement their functionality in their methods. More methods in the system indicate increased functionality. Two systems that deliver the same functionality where one has less number of methods indicate that methods of the system represent a wider range of services.

1.7 Total Number of Implemented Components (TNIC): The count of the total number of implemented components only. Implemented components are where system functionality is implemented.

1.8 Total Number of Components (TNC): According to the CO process model described in [5] two groups of components can be seen in a CO model. These two groups constitute abstract components that exist only in the conceptual model and physical components that represent the implemented ones. Total number of components in the systems is a design decision. While some designers may favor relatively smaller components, another design decision may favor a fewer number of relatively larger components. So the total number of components in the

system is a design issue that influences the overall structure of the system.

Viewpoints:

    a. Increased number of components implies that components are more specific and every component delivers limited functionality.

    b. Since components are specific they have higher potential for reuse.

    c. More components indicate that more effort will be spent during the integration stage.

2- Component Internal Complexity: Conte et al [27] found that the internal structure of system building elements affects the overall complexity of the system. Building on Conte et al findings, and after investigating the internal of a component the following attributes can be identified as potential factors influencing a component's structural complexity.

    2.1 Methods Complexity: In the previous discussion we have shown that a component's functionality is implemented in its methods. Methods structural complexity is widely discussed in the literature. The two principal influencing factors are again coupling between methods of a component, cohesion of methods of a component. Figures 3.7.a, 3.7.b, and 3.7.c present three different pictures that can be conceived from methods inside a component. In Figure 3.7.a we have very high cohesion and zero coupling, in Figure 3.7.b moderate levels of cohesion and coupling between methods of a component and in Figure 3.7.c we have excessive coupling and low cohesion. While zero coupling means that every method implements all the functionality it requires inside itself and is not dependent on any other methods, excessive coupling indicates that methods are highly dependable on each other. For example, in order to comprehend method *m7* in Figure 3.7.c it is necessary to also understand methods *m2, m3, m4, m5,* and *m6* since all these methods are invoked from m7 to complete a requested service.

Besides coupling and cohesion factors, the internal design of a method has been proven to play an important role on the methods testing and maintenance. McCabe Cyclomatic complexity [57] is being efficiently used for more than two decades as a predictor of the testing effort of a method. Cyclomatic complexity of methods, Coupling between methods, and Cohesion of methods of a component are considered to characterize methods in a component. Henry and Kafura described two measures of coupling which are Fan-in and fan-out [Henry and Kafura]. These two measures have been widely discussed and empirically validated. We suggest the use of Fan-in and fan-out of a component's methods as one indicator of the internal complexity of the component.
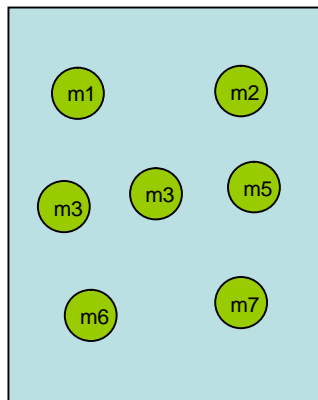
Figure 3.7.a: Zero Coupling   **Figure 3.7.b: Moderate Coupling**
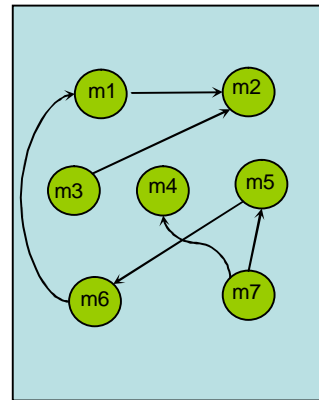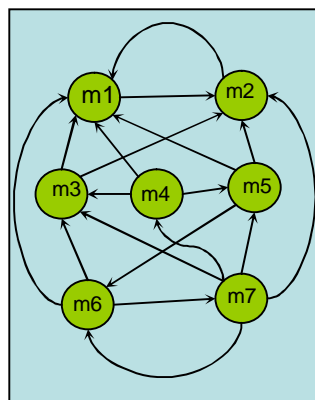

Figure 3.7.c: Excessive Coupling

2.2 Interfaces Complexity: In the previous discussion we have mentioned that interfaces are components' access points. Number of interfaces per component, number of methods per interface, interface fan-in counts are important aspects to consider when tackling influence of interfaces on component's complexity. The actual influence of interface structures on a component's complexity can only be determined during the experimental validation. The basic viewpoints about the potential influences of the number of interfaces on a component complexity have been discussed before. The following viewpoint is about the potential impact of number of methods in an interfaces on the complexity of a component.

a) The number of methods in an interface implies the breadth of the service it supplies. Too many methods in an interface may limit the possibilities of its use by other components. On the other hand, zero methods in interfaces implies no service is provided by that interface.

3- Interface Internal Complexity: Interfaces play principal role in CO software development. They are considered as the components' access points. Due to their important role interfaces should be designed with a lot of care. While experience and intuition help designers make decisions about alternative interface designs, decision based on quantifiable aspects proved to be more accurate and dependable. Interfaces' complexity which is an indication of its quality as well is a composite aspect that depends on the interface building elements and its interactions with other interfaces. Gill and Grover [101, 102] say that CO software complexity can be measured based on interface characterization. That is due to the fact that better characterization of component interfaces helps to easily understand and resolve components problems [83]. We here present the elements and their quantifiable aspects in interfaces.

3.1 Number of Methods of an Interface: Usually an interface represents a service supplied by a component. Large number of methods in an interface implies that the interface provides a wide service which may limit the possibility of utilizing the interface in different systems. Also

large number of methods in an interface indicates that its fan-in count will be high which indicates that the interface is highly dependable and extra effort for testing and implementing may be a consequence.

3.2 Events Out: Events out are the events the interfaces notifies others about. Events our count is an indication of dependence on other components.

3.3 Events In: Events in are the events that the interface is notified about from other component interfaces. Count of events in indicates the degree to which the interface is critical in the system and the intensity in which it will respond to other components interface events.

System complexity spreads over system building elements. On one side, we have complexity which is inherited from internal complexities of a system's elements and one the other side we have complexity resulting from the interactions between these elements. In Table 3.2 we present a summary of the quantified attributes and their metrics with the metrics definitions.

Table 3.2: Attributes, Metrics, and Metric Definitions

Attribute	Metric	Definition
System Structure	TNC	Total number of Components in the system
	TNI	Total Number of interfaces in the system
	TNCO	Total number of Connectors and messages in the system
	DCT	Depth of the composition tree. Count of levels in the composition tree
	WCT	Width of the composition tree. Maximum width of the composition tree.
	TNIC	Total number of implemented components only.
	TNM	Total number of methods in the system
	CSC	Cohesion of system components
Component's Internal Complexity	NOCC	Number of connectors per component
	NOMC	Number of methods per component
	NOIC	Number of interfaces Per component

Table 3.2 (Continued)

		NOMI	Number of methods per interface
Interfaces Internal Complexity		NOEO	Number of events out in an interface
		NOEI	Number of events in of an interface
Methods Internal Complexity		CC	Cyclomatic complexity from McCabe [57]
		Fan-in	Fan-in metric from Henry and Kafura [46]
		Fan-out	Fan-out metric from Henry and Kafura [46]

3.4 A Complexity Model for CO Software Systems

Based on the detailed metrics analysis presented in section 3.3, it became clear that a complete complexity model can be built. Also, the metrics analyses have shown that CO systems complexity is a multidimensional feature that spreads over components, connectors, interfaces, methods, and other less significant elements. We will define Complexity of CO systems in three levels. The first (lowest) level complexity is a result of the complexities of the component's methods. The second level complexity is a result of the internal complexity of components. The third (highest) level complexity is a result of components organization in the system.

3.4.1 Level 1: Method Complexity

Method Complexity (MCOM): The lowest level of complexity in a CO system is the complexity of component methods. Methods are the functionality producing units of a component. The complexity of a method can be characterized by two metrics:

   1.1 Cyclometic Complexity (CC). Cyclometic complexity [57] has been used effectively as a count of the number of test cases required to test an algorithm and then a measure of the testing effort required.

1.2 Number of Calls to Other Methods (NCOM): This metric is estimated as the count of methods called from this method. It is derived froim the fan-out metric [46]. This metric is an indication of how much the method is dependent on other methods. Dependency means that in order for the method to provide its functionality some other methods are required. Understanding, updating, and maintaining a method that is dependent on other methods will necessarily require an understanding of all these methods.

The Method COMplexity (MCOM) of a method m will be estimated as a function of its CC number and its NCOM value and can be expressed as:

$$MCOM (m) = f (CC, NCOM)$$

Since a method's complexity is affected mainly by the factors described above, the function f can be the sum of the two values. Then, MCOM (m_i) for any method m_i can be estimated as:

$$MCOM (m_i) = CC(m_i) + NCOM(m_i) \qquad \text{for any method } m_i$$

The total method complexity of a component C_j (TMCOM) is the sum of all complexities of individual methods and is estimated as:

$$TMCOM(C_j) = \sum_i MCOM (m_i) \qquad \text{for all methods } i \text{ in the component } C_j$$

3.4.2 Level 2: Component Complexity

Component Complexity (CCOM): complexity of a software component can be characterized as 1) complexity coming from the component's methods, 2) complexity coming from the component's interfaces, and 3) complexity resulting from its dependency on other components. These three aspects will be estimated using the following metrics:

The complexity of a software component C can be viewed as a function f that is affected by all these three factors and can be expressed as:

$$CCOM\ (C) = f(TMCOM(C), NOI(C), NCO\ (C))$$

The total components' complexity (TCCOM) based on all components C_j in a component-oriented software system S is estimated as:

$$TCCOM(S) = \sum_{j} CCOM\ (C_j) \quad \text{for all components } C_j \text{ in S}$$

3.4.3 Level 3: System Structural Complexity

This complexity mainly results from the organization and interactions between system components. We will call this level complexity as Emergent System Complexity (ESCOM). Emergent system complexity is a function of the structural attributes of the system.

$$ESCOM(S) = f\ (TNC, TNI, TNM, TNCO, DCT, WCT)$$

When considering or trying to evaluate Overall System Complexity (OSCOM) it is necessary to consider both components of complexities and emergent system complexity. That is because a CO system is a set of components and connectors organized in some structure. The overall system complexity of a component-oriented software system S can be evaluated as a function f of these types of complexities and can be expressed as:

$$OSCOM(S) = f(\ TCCOM(S), ESCOM(S))$$

Evaluation and validation of these metrics will be provided in Chapter 4.

CHAPTER 4

METRICS EVALUATION AND VALIDATION

Several researches in the field presented properties that are used to characterize good metrics from a mathematical and measurement theoretical perspectives [17, 36, 52, 72, 84, 88, 91, and 92]. The common features in all of these works can be summarized as follows:

1- A metric must possess some desirable mathematical properties. Provide a scale and range of values. Provide thresholds of good and bad behavior. Metrics value should be observer independent.

2- A metric must be empirically valid: Can be used to make managerial and/or engineering decisions. Also the metric must precisely characterize the attribute of interest.

4.1 Properties of CO Complexity Metrics

The literature of metrics evaluation approaches does not present a globally accepted set of properties of complexity metrics. Also, none of the described properties have specifically tackled the particular and new aspects of component-oriented software systems. Due to these two reasons, we introduce a set of properties that a component-oriented system complexity metric must satisfy. The properties defined here came as a result of investigating the properties described in Weyuker [88], Briand et al. [17], Kitchenham et al. [55], Tian, and Zelkowitz, [84], Schneidewind, [72], and Zuse, [94] and the viewpoints of others criticizing them. The properties described here do not have a generic nature in the sense that we do

not claim that they can apply to all types of complexity metrics. Proposed properties are listed below:

Property 1: A complexity metric value can not be a negative number. For some complexity metrics it is necessary to be even stricter, since a value of zero will not always be accepted.

Interpretation guidelines: The meaning of a complexity metric value for a software artifact (a software artifact can be a method, component, or the whole system) that provides some functionality to be equal to zero is that the artifact is the least-complex possible design that can provide that functionality. A lower complexity value, for two functionally equal designs, is preferred over a higher value since lower complexity is believed to be associated with less development, testing, and maintenance efforts.

Property 2: A software complexity metric must provide a scale of values. Comparison between different alternatives must be possible. For any two software artifacts it must be possible to compare and then make managerial decisions according to the metrics values. For any two functionally-equal components C_1 and C_2, if Complexity(C_1) > Complexity(C_2) then C_2 is preferred over C_1 assuming that we keep all other parameters constant. This is due to the fact that C_2 will require less development, less testing, less integration, and less maintenance efforts. Also, metrics must provide enough information to help managers make business decisions and compare different alternatives.

Property 3: The complexity of a single software unit S composed of two software components can not be less than the sum of the complexities of the individual components. So, for any CO system S and any two components C1 and C2

$$Complexity(S) >= Complexity(C_1) + Complexity(C_2)$$

According to the metrics described in section 4, the complexity of a component-oriented software system is a function of the complexities of individual components that make it up, and an added complexity will appear as a result of new

50

interactions that may exist between the components. In the best case, when a system is composed of two components and no new added interactions between the components are available, the system's complexity will be equal to the sum of the individual component complexities.

Property 4: If a component C is decomposed into two or more components C1, C2, .., Cn then the sum of complexities of the resulting components is no more than the overall complexity of the original component.

$$Complexity(C_1) + Complexity(C_2) + \ldots + Complexity(C_n) <= Complexity(C)$$

The reason for this is that, according to our perception of the three-level component-oriented software complexity, there is usually an added complexity whenever two components are composed. The new complexity usually results from the interactions between these components. So, when the component is decomposed these links will disappear and only the component's intrinsic complexity will remain.

Property 5: The complexity value of one component does not have a direct relation to its functionality, i.e. for any two components C_1 and C_2, if Complexity(C_1) > Complexity(C_2) then it is not necessary that C_1 provides more functionality than C_2. The same functionality can be obtained by different designs and then implementation. The complexity measures described in this article are those that enable software developers and/or managers to take decisions and contrast/compare different alternative solutions to the same problem. Of course, any added functionality may introduce an added complexity. So, a complexity metric does not consider evaluating functionality of the system or provide any information about the system size.

Property 6: The complexity metric value is directly influenced by system structure. Two different structures for the same functionality can result in two different complexity values. A complexity measure of the system can have different values for different alternative architectures of the same functionality.

51

4.2 Metrics Evaluation

The proposed metrics have been evaluated against Tian and Zelkowitz axioms of a good complexity metrics [84]. Tian and Zelkowitz described and approach for both evaluating metrics and another approach to make choice between alternative metrics that qualify. The set of axioms defines five properties that a complexity metric must possess in order to qualify for adoption. These properties can be briefly described as follows:

1- Property 1: A complexity metric must have the capability to compare between functionally equivalent alternative systems.
2- Property 2: A complexity metric must have the capability to compare between components and composites.
3- Property 3: A complexity metrics must possess a discriminative power and can produce different values to different programs.
4- Property 4: A measure must not have a region where all values cluster around.
5- Property 5: A measure is a complexity measure if it satisfies properties 1-4.

Tian and Zelkowitz also introduced a metrics classification approach which defines a boundary condition that can be used to reject inappropriate metrics. They also suggest that a metric's discriminative power can be evaluated according to that metric's predictive power, simplicity, and the value of information it embodies.

We evaluated the proposed complexity measures against these properties. All of the proposed metrics are qualified to be considered as complexity metrics. In making selection between different alternatives we followed a mixed approach where simplicity is important but also focused more on the predictive power of the metric.

4.3 Metrics Validation: The Experiment

It is widely accepted that software metrics are useless unless they can be of some practical use. Metrics can be of practical use for users, developers, managers, and team leaders.

For developers, managers, and team leaders metrics are useful in making predictions about some process features (e.g. cost/effort estimation, resources, etc..). Also metrics can be used to make predictions about the potential behavior of the system (Performance, reliability, efficiency, maintainability, etc..). Metrics help developers and managers detect the more complex components early at the design stage and take decisions to redesign these components. On the other hand, customers/users can use metrics values to make comparison between several alternatives, and identify the ones with higher quality.

Our validation approach comprises checking the potential of using metrics values in predicting one or more of the followings: design effort, correction effort, integration effort, and productivity. We do not claim that if a metric does not have direct influence on one or more of these process factors should be considered invalid. That is because this metric still can have some influence on some other product nonfunctional attributes like reliability, performance, or any other factor whose examination requires experimenting implemented systems.

In the study, we have considered both cases where 1: a complete component orientation with assumed available components and also 2: the case where some component development is necessary. To remove any ambiguity that may arise on the reader side, before proceeding any further, we will provide the definitions of the terms: Design Effort, correction effort, integration effort, and productivity:

Design Effort: Design is the process of defining the system abstractions, components, interfaces, data structures, and the working relationships among components [98]. The design process results in a document that contains system models in some design description language (e.g. COSEML). The system design should be described detailed enough to be implemented. Design effort is the time spent to transform system specifications into design models including editing of the models.

Correction Effort: The time that is spent to make any changes affecting methods, interfaces, properties, or relationships of the component after being initially

53

designed. Total correction effort is the total correction effort spent on all components.

Integration Effort: the time spent to define components relationships with other components, including the designing of connectors and their specifications.

Productivity: Developer productivity (FP/Person-Hour) is estimated as the total function point count divided by the total time spent on design, correction, integration, modeling, and editing the design models.

4.3.1 Data Sources

Data has been collected from 40 student projects developed in three different semesters (Fall 2002, Fall 2003, and Fall 2005). All projects have been designed for component oriented software development using the dedicated CO software modeling language COSEML. The majority of the projects have been designed as a term project in a graduate level course 'System development using abstract design' at the Middle East Technical University (METU) in Ankara. Eight out of the forty projects were designed by senior students in the same department. One project was prepared as the main part of a master's thesis in the department. That project was the largest in terms of number of components and function points (FP)'s counts. Projects were designed in teams of one, two, or three students and vary in their sizes based on the number of team members enrolled. To have a feeling of project sizes, we collected Function Points [4] and the total number of boxes (abstractions and implementation level components) values for every project. As of the Function points count, the largest project has 510 FPs and the largest project in terms of the count of boxes has 287 boxes and 16 physical components with 33 interfaces. The project with the least FP value has 24 FPs and the same project has a total of 12 boxes with only 3 components and 3 interfaces.

4.3.2 Developers' Backgrounds

The developers of the projects were all students at the Department of Computer Engineering in METU. Some of the developers were research assistants while the majority were working as software engineers in the industry. All of the developers had previous programming and system analysis and design experiences in OO software engineering using C++ and/or Java. On the other hand, none of the developers had any previous experience in CO software modeling and design using COSEML modeling language before enrolling to the class.

4.3.3 Data Collection

Contribution to project metrics was completely a voluntary job. Directly after students submitted their project proposals we performed a one-hour lecture in which we described the followings:

1) Benefits of metrics on the overall software development process were described.
2) The metrics to be collected were defined. Example metrics estimations were demonstrated.
3) Students were informed that their projects data will be used in a serious research so those who do not want to contribute are free in that.
4) Also the terms design effort, correction effort, and integration effort were defined to students.
5) Metrics collection forms fields were described field by field.
6) Online material was posted. Contact information e.g. email address, phone number, and street addresses were available to developers so that they were free to contact anytime.
7) Students have been assured that metrics will never be used to evaluate their performance or be used while grading their projects.

Data has been collected by the developers themselves. The metrics collection forms that have been used by the classes of fall, 2002, and fall, 2003 are available in Appendix A. The metrics set has been further refined. Some metrics were eliminated, new metrics were added. So, for the class of fall 2005 we distributed

two separate forms; one includes metrics for the project as a whole and the other contains metrics to estimate for every component separately. The two metrics forms are available in Appendix B. A document containing a detailed discussion of the metrics, their definitions, and their estimations were available online for the developers' free access. Every possible effort has been expended in instructing the developers to ensure a clean data collection with least number of errors.

4.3.4   Correctness Test

Data collection is important; to be useful it must provide the correct data. We applied the following procedure to eliminate inconsistencies and casual defects that existed in the data:

1)   Six projects have been eliminated from the study because they included data that violates intuition and well known facts of software engineering. Examples of these inconsistencies include- but not limited to: in one project developers reported the total design effort as 1800 person hours which is more than the total period (days x 24 hours) allocated for the project design, unfortunately we could not return to the developers to inquire about the correctness of this number. In another project developers reported that number of components is more than the number of interfaces, this violates the fundamentals of CO since every component must have at least one interface.

2)   Three projects were removed from the experiment due to inconsistencies that seemed to be a result of misunderstanding of the terms e.g. one of these projects reported the total number of interfaces as 9 and the average number of methods per interfaces 3 while the total number of methods was reported as 11 methods.

3)   Four projects were removed because the developers reported that they could not estimate the effort they spent on preparing the design, correction, integration.

4)   Two projects have been removed because developers demonstrated very unexpected system decomposition. In one project the total number of

components, interfaces, and abstractions was 9 while the estimated function points value was 535.

We have carried walkthroughs to make sure that maximum effort was used to eliminate error in data. We found some errors in function points estimations. Error-prone locations have been handled and new values have replaced the old ones.

4.3.5 Regression Analyses

The aim of the regression analysis is to investigate whether metrics can help in making predictions about the followings:

1) Complexity related to Size: Function Points (FP) [4] is a widely accepted size related measure. We are investigating the relationships between CO metrics and FP counts in using single regression and multiple regression models.

2) Complexity resulting from Connectivity and its influence on Effort: Henry and Kafura [46] evaluated system complexity resulting from connectivity using mainly two metrics; Fan-in and Fan-out. In a CO software system the degree of connectivity is related to:

    a. Number of interfaces.
    b. Number of connectors.

   The potential influences of these factors on Design effort, correction effort, integration effort, and developer productivity is explored.

3) Complexity resulting from Structure and its influence on Effort: A CO system is structured in a hierarchy in several levels. High levels usually include system abstractions while lower levels represent implementation level components. The potential influence of system structure on design, correction and/or integration efforts is explored.

Regressors that are intuitively believed to have an impact on the different regressands are first identified. A summary of the regressands and the intuitively influential regressor sets are presented in Table 4.1.

Table 4.1: Regressands and Their Intuitive Regressors Summary

No.	Regressand	Description	Intuitively Influential Regressors
1	FP	Total count of unadjusted function points (Abretch and Gaffney, 1979, 1983)	# of Components, # of Methods # of Imp.. Components # of Methods # of Connectors
2	FP/Interface	Total count of unadjusted function points (Abretch and Gaffney, 1979, 1983) divided by the total number of interfaces	# of Methods/Interface # of Connections/Interface # of Events/Interface
3	Design Effort	Total Effort in person hours spent on building system models using COSEML	# of Components # of Methods # of Imp. Components # of Methods # of Connectors # of Connectors/Comp. # of Methods/Comp. # of interfaces/ Comp. # Depth of Composition Tree
4	Design Effort/ Component	Total Effort in person hours spent on building system models using COSEML divided by the total number of components	# of Components # of Methods # of Imp. Components # of Methods # of Connectors # of Connectors/Comp. # of Methods/Comp. # of interfaces/ Comp. # Depth of Composition Tree
5	Correction Effort	Total Person hours spent on making changes affecting methods, interfaces, properties, or relationships of the components after being initially designed.	# of Components # of Methods # of Imp. Components # of Methods # of Connectors # of Connectors/Comp. # of Methods/Comp. # of interfaces/ Comp. # Depth of Composition Tree
6	Correction Effort/ Component	Person hours spent on making changes affecting methods, interfaces, properties, or relationships of the components after being initially designed divided by the number of components.	# of Components # of Methods # of Imp. Components # of Methods # of Connectors # of Connectors/Comp. # of Methods/Comp. # of interfaces/ Comp. # Depth of Composition Tree
7	Integration Effort	Total person hours spent on defining components' relationships with other components.	# of Components # of Methods # of Imp. Components # of Methods # of Connectors # of Connectors/Comp. # of Methods/Comp. # of interfaces/ Comp. # Depth of Composition Tree

Table 4.1 (Continued)

8	Integration Effort/ Component	Total person hours spent on defining components' relationships with other components divided by the total number of components.	# of Components # of Methods # of Imp. Components # of Methods # of Connectors # of Connectors/Comp. # of Methods/Comp. # of interfaces/ Comp. # Depth of Composition Tree
9	Productivity	Estimated as the ratio of FP/Person hour.	# of Components # of Methods # of Imp. Components # of Methods # of Connectors # of Connectors/Comp. # of Methods/Comp. # of interfaces/ Comp. # Depth of Composition Tree
10	Total Development Effort	Total person hours spent on system development.	# of Components # of Methods # of Imp. Components # of Methods # of Connectors # of Connectors/Comp. # of Methods/Comp. # of interfaces/ Comp. # Depth of Composition Tree

Ten different regression models are obtained. For each model the regression equation, model plot, and a Table of regressand's actual values and predicted values are presented. For every regression equation the coefficients' p-values are presented below them. The p-values are presented to provide a feeling of the statistical significance of the corresponding variable. Also the practical importance of every model is discussed and supported by the average error estimates of the predicted values. To remove any confusion that may arise, we present in Table 4.2 a list of the terms and abbreviations with their corresponding descriptions as they appeared in the regression models.

Table 4.2: List of Terms and Abbreviations Used in the Regression Models

Term or Abbreviation	Description
Component or Comp.	Number of Components (abstractions, implementation level components, and interfaces)
PComponent	Number of Implementation (Physical) level components (not including interfaces)
Interface or Int.	Number of Interfaces
Methods or Meth.	Number of Method
Connector or Conn.	Number of links between two components or two interfaces
FP	Function Points count

4.3.5.1 Total FP Count Regression Model

In order to find a generalized model for predicting FP count in the system, potential influencing regressors have been identified first (See Table 4.1). A forward addition approach has been used to test the regressands one by one. The resulting generalized model is given as:

$$FP = 0.8 * (Components) + 4.3 * (PComponents) - 1.8 * (Interfaces) + 0.5 * (Connectors) + 12.7$$
$$(p = 0.01) \qquad (p = 0.03) \qquad (p = 0.06) \qquad (p = 0.1) \qquad (p = 0.02)$$

The model demonstrated a good prediction level with an average error rate of %21 in the worst case and %8 when the outliers were excluded. The complete list of actual and predicted FP values with the corresponding error rates are presented in Table 4.3. The most surprising and, perhaps, "unexpected" finding is that the number of FPs decreases as the total number of interfaces increases. This finding violates intuition since in a CO system a component's functionality is presented through its interfaces. So, more interfaces should lead to more functionality and more FP count in the system. The reason(s) for this contradictory result may be related to one or more of the followings:

1) Due to the inclusion of implementation components "*PComponents*" whose count is highly correlated with the count of *Interfaces* with a correlation

coefficient value 0f 0.97. The coefficient value of *PComponents* is quite high relative to the coefficient value of *Interfaces*. So, the negative sign for the *Interfaces* coefficient may be to balance the high positive value of the coefficient of *PComponents* in the model.

2) The fact that system developer had their first CO software design with the projects used in this research may be a reason for coming up with low quality designs.

The curve fitting plot of the model is given in Figure 4.1.

Table 4.3: FP Estimation Results

Actual	Estimated	Residual	%Error
142	128.06	13.94	9.82
146	130.54	15.46	10.59
108	96.77	11.23	10.40
85	87.45	-2.45	-2.88
66	62.16	3.84	5.82
113	82.83	30.17	26.70
65	54.29	10.71	16.48
44	56.22	-12.22	-27.77
28	35.97	-7.97	-28.47
66	62.34	3.66	5.55
76	102.55	-26.55	-34.94
58	55.54	2.46	4.24
95	79.24	15.76	16.59
66	94.15	-28.15	-42.65
110	99.96	10.04	9.13
72	72.51	-0.51	-0.71
102	107.47	-5.47	-5.36
70	66.86	3.14	4.48
44	51.87	-7.87	-17.89
38	54.79	-16.79	-44.20
69	71.03	-2.03	-2.94
93	95.15	-2.15	-2.31
60	61.19	-1.19	-1.98
48	51.32	-3.32	-6.91
510	513.76	-3.76	-0.74
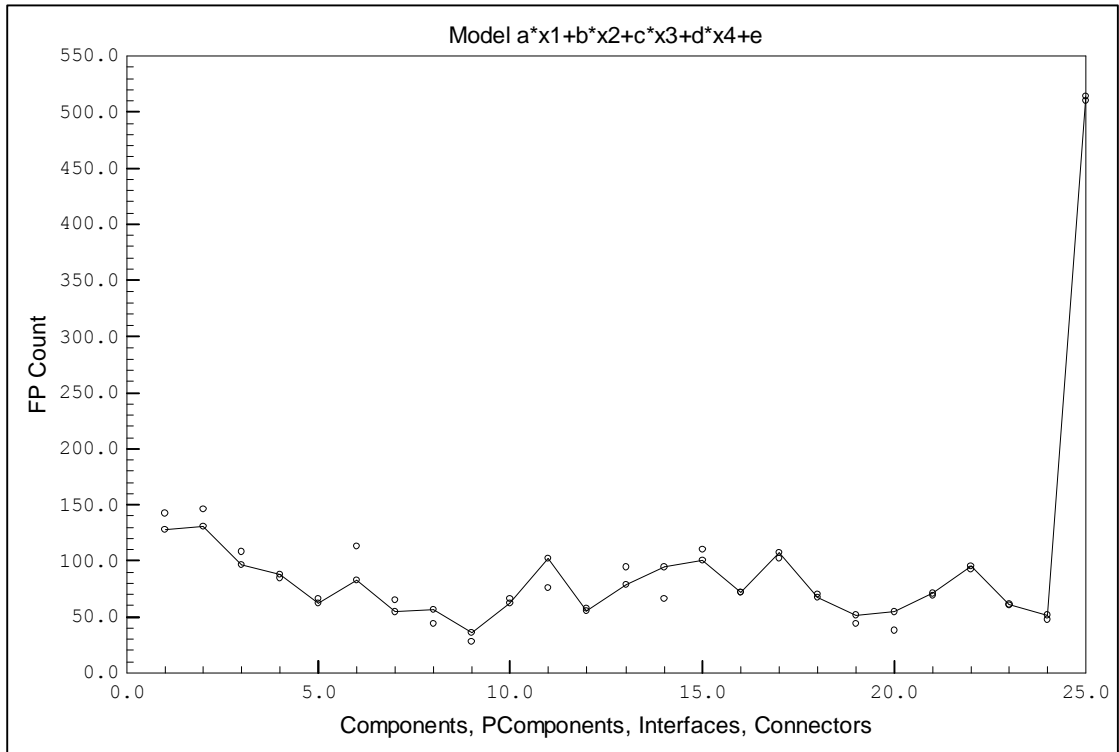
Figure 4.1: Total FP Regression Model Plot

4.3.5.2 FP Per Interface Regression Model

Returning to the definition of a CO software system, we can see that components implement interfaces to provide services to their clients. So finding the factors that influence the count of FP in an interface becomes a necessity. Also, here a forward addition approach has been followed to find the influencing factors with reference to the intuitive relationships presented in Table 4.1. The result of the regression analysis is the following model:

FP/Interface = – 8.5/ (Methods/Interface) + 0.94 * (Connectors/Interface) + 5.2
                 (p = 0.007)   *(p = 0.04)*   *(p = 0.004)*

While the R^2 value is relatively low "0.45", the model demonstrated an acceptable level of average error rates with an initial value of 22% and 17% after removing the outliers. Also the model is statistically significant in the confidence interval of 95%. The highest p-value is 0.04 and prob(F) is 0.0.

Looking deeply in the model we can see that the FPs in an interface increases with the increase in the numbers of methods and/or connectors in that interface. These

results are practically important for us since they both meet intuitive thinking and are practically applicable. A detailed list of the actual and predicted FP/Interface values with the corresponding residual and estimated error rates is presented in Table 4.4. The curve fitting plot of the model is shown in Figure 4.2.

Table 4.4: FP Per Interface Estimates

Actual Value	Estimated Value	Residual	%Error
4.30	3.79	0.51	11.84
11.23	7.51	3.72	33.09
6.35	4.73	1.62	25.49
7.08	4.40	2.68	37.89
7.33	6.52	0.81	11.10
9.42	7.72	1.70	18.06
7.22	6.78	0.44	6.07
5.50	6.26	-0.76	-13.79
3.50	4.39	-0.89	-25.52
7.33	8.48	-1.15	-15.70
5.43	6.13	-0.70	-12.84
8.29	6.72	1.56	18.86
5.00	4.50	0.50	9.91
3.14	3.92	-0.78	-24.78
5.00	3.63	1.37	27.45
4.50	6.22	-1.72	-38.29
5.37	5.28	0.09	1.66
4.38	5.68	-1.31	-29.91
2.44	5.03	-2.59	-105.96
4.75	6.20	-1.45	-30.43
4.93	5.64	-0.71	-14.36
4.89	4.87	0.02	0.44
4.29	5.18	-0.89	-20.87
5.33	6.12	-0.79	-14.83
6.22	7.51	-1.29	-20.67
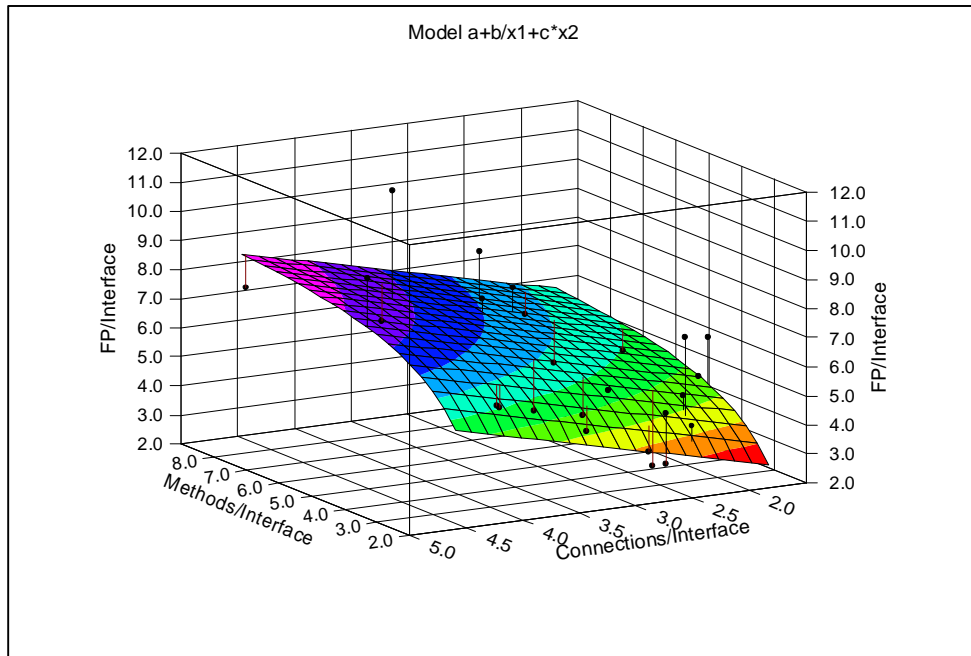
Figure 4.2: FP per Interface Regression Model Plot

4.3.5.3 Total Design Effort Regression Model

The necessity of obtaining a general model for evaluating Design effort using some product measures is widely recognized. One of the main advantages of having such a model is that it can help in building a rather more generalized model for predicting total development effort benefiting from the relatedness of design effort to the total development effort. The obtained regression model is described as follows:

Design Effort = -0.65*(Comp.) + 0.83*(meth.) + 0.73*(Conn.) – 33.9* (Meth./Comp)
(p = 0.0004)   (p = 0.0)   (p = 0.0006)   (p = 0.0004)

The model is statistically significant at the confidence interval of 99% with highest p-value of 0.0006 and p(F) of 0.0. The model has a high R^2 value of 0.98. The model is not without unexpected features. Two unexpected features available in the model are:

1) Design effort decrease when the total number of components increases.

2) Design effort decrease when the average number of methods per component increases.

64

The reason(s) to these unexpected features in the model can be due to one or more of the following:

1) High degree of correlation is observed between the variables included in the model. The correlation matrix between Components, Methods, and Connections which are included in the model is presented in Table 4.5.

Table 4.5: Correlation Matrix between Model Variables

	Components	*Methods*	*Connections*	*Methods/Comp.*
Components	1			
Methods	0.95	1		
Connections	0.95	0.96	1	
Methods/Comp.	-0.03	0.27	0.11	1

The high degree of correlation between model variables (multicollinearity) can lead to a situation like what encountered in our model.

Also a possible explanation is that: providing a solution with bigger components providing rich services, design becomes easier; connection design is easier: most of the job is being handled inside the components.

The predicted results are practically interesting. The average error rates are 27% for the initial estimate and 15% when excluding the outliers. The list of actual and predicted values of total design effort with the corresponding residual and average error rate are presented in Table 4.6 and the model plot is shown in Figure 4.3.

Table 4.6: Total Design Effort Estimates

Actual Value	Predicted Value	Residual (Actual - Predicted)	%Error
38	57.68	-19.68	-51.78
32	43.01	-11.01	-34.42
24	24.03	-0.03	-0.11
16	3.48	12.52	78.26
24	25.56	-1.56	-6.48
42	46.26	-4.26	-10.15
20	26.72	-6.72	-33.58
26	22.64	3.36	12.91
19	6.03	12.97	68.24
24	45.58	-21.58	-89.91

Table 4.6 (Continued)

32	31.05	0.95	2.97
24	18.38	5.62	23.42
34	35.56	-1.56	-4.60
32	37.05	-5.05	-15.79
44	35.48	8.52	19.36
40	49.17	-9.17	-22.94
46	36.81	9.19	19.97
38	38.98	-0.98	-2.58
76	40.95	35.05	46.12
24	17.60	6.40	26.66
22	39.50	-17.50	-79.56
45	34.86	10.14	22.52
32	30.84	1.16	3.64
20	20.06	-0.06	-0.32
525	522.08	2.92	0.56
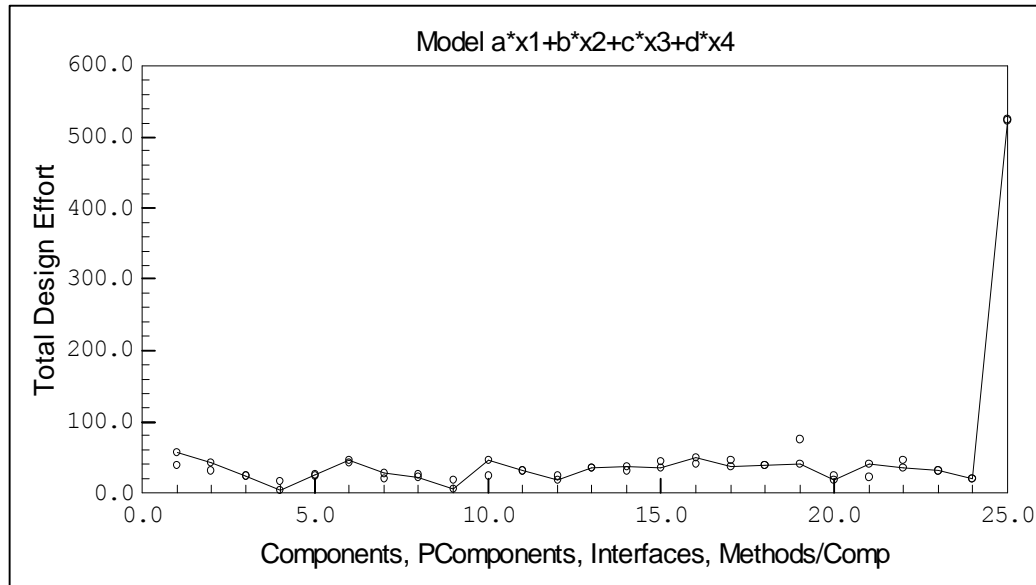
Figure 4.3: Total Design Effort Regression Model Plot

4.3.5.4 Design Effort Per Component Regression Model

In the previous section we presented a prediction model for total design effort and we have seen the influencing factors. A prediction model for design effort per component is as import as that for total design effort. The necessity of a prediction model for design effort per component emanates from the need to find the factors that can explain increasing effort per added component. A forward addition

approach is followed to identify the statistically significant influencing factors. The set of potentially influential variables has been identified and variables are added when p-value is less than or equal to 0.05. The resulting regression model takes an exponential curve with four variables as follows:

Des. Effort/Comp.= e^ (-0.03*TNC)+0.13*(TNIC)+0.46*(TNI/TNC)+0.32*(TNM/TNC)- 1.48)
$\quad\quad\quad\quad\quad$ (p = 0.0) $\quad\quad$ (p = 0.0) $\quad\quad$ (p = 0.0) $\quad\quad$ (p = 0.0) $\quad\quad$ (p =0.0)

The model demonstrates a high statistical significance with all p-values being equal to 0.0. Also, the value of R^2 is quite good for adoption of the model.

From a practitioner's point of view the model is not encouraging due to its exponential nature. This makes it difficult to make predictions about potential change when a variable value is changed. The interesting thing about the model is that it meets intuitive thinking. Additional effort to system design increases when a component is added, number of interfaces a component implements increases, when there are more connections, and when the number of methods increases.

The predictive power of the model is quite good to recommend for application. The average error rates are %17 and 13% for initial estimate and estimate after excluding outliers respectively. A complete list of actual and predicted values with their corresponding residual and error estimates is presented in Table 4.7. The model plot diagram is shown in Figure 4.4.

Table 4.7: Design Effort per Component Estimates

Actual Value	Estimated Value	Residual	%Error
0.46	0.72	-0.26	-56.17
0.40	0.29	0.10	26.10
0.37	0.48	-0.11	-30.63
0.27	0.31	-0.04	-16.71
0.83	0.92	-0.10	-11.66
1.56	1.45	0.11	6.81
0.71	0.91	-0.20	-28.02
0.90	0.87	0.03	2.86
1.58	1.20	0.38	24.01
0.80	1.03	-0.23	-28.40
0.57	0.47	0.11	18.58
0.80	0.65	0.15	19.10
0.76	0.92	-0.17	-21.97

Table 4.7 (Continued)

0.64	0.89	-0.25	-39.67
0.77	0.73	0.04	5.41
1.14	1.18	-0.03	-2.92
0.68	0.46	0.21	31.31
0.81	0.80	0.01	1.65
2.53	1.73	0.80	31.56
0.92	0.86	0.06	6.52
0.44	0.65	-0.21	-47.89
0.69	0.56	0.14	19.66
1.00	0.92	0.08	7.89
1.00	0.93	0.07	7.23
1.83	1.72	0.11	6.11
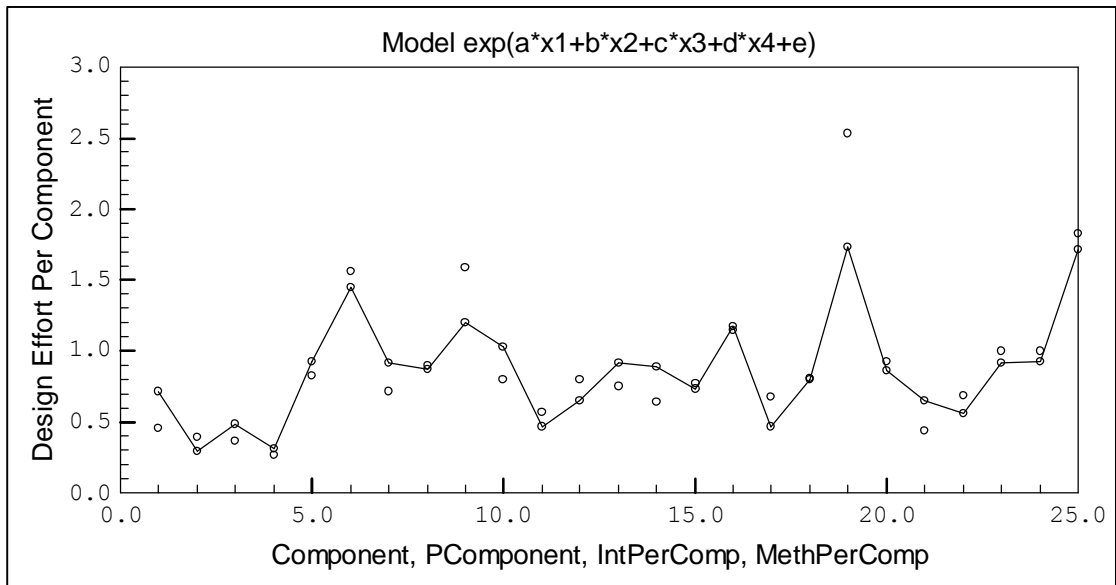
Figure 4.4: Design Effort per Component Regression Model Plot

4.3.5.5 Correction Effort Regression Model

In general, maintainability in software systems is one of the most important quality indicators. Maintainability can be quantified as the average time required to fix an error [81]. It is possible to build prediction models for maintainability only if data from implemented and operational systems' are available. The data used in this research has been collected from projects which are only designed but never implemented and put into operation. For this reason actual maintainability prediction models are not possible, at least, for the time being. We looked for another factor that can be used to give an indication about maintainability. We

68

used the term *correction-effort* in our research to provide some sense about maintainability. In Table 4.1 we identified the different variables (quantified features) of a system model that can have influence (positive or negative) on correction effort. In the regression analysis the variables have been added in a forward addition manner where variables are added if they satisfy the 0.05 confidence interval. The regression analysis produced the following model:

Correction-Effort = 0.08 * (# of Components) - 1.5 / (# of Methods/Component) + 2.13
  (p = 0.0)   (p = 0.02)   (p = 0.0)

The model demonstrates a high statistical significance with maximum p-value 0f 0.02 and R^2 0f 0.95. Besides being statistically significant, the model also is practically significant due to at least the following reasons:

1) The coefficient variables do not violate intuition since it is intuitive that correction effort should increase as the number of components and the average number of methods per component increase.

2) The average error rates are encouraging to recommend the model for practical use with a value of 19% for the initial error rate which drops to 13% when removing the outliers. The complete lists of actual and predicted value with the corresponding residual and error rates are presented in Table 4.9. The plot of the regression model is shown in Figure 4.5.

Other variables are believed to have influence on correction effort. The reason(s) behind this violation can be one or more of the followings:

1) High multicollinearity between the different variables. Table 4.8 presents a complete correlation matrix of all the variables that have been fed to the model. Multicollinearity is a common problem that encounters researchers carrying out such researches.

Table 4.8: Correlation Matrix between Regression Variables

	Comp.	Meth.	Pcomp	Int.	Conn/Int	Conn.	In./Comp	Meth/Comp	Conn./Comp	Meth./Int.	DCT
Comp.	1.00										
Meth.	**0.95**	1.00									
Pcomp	**0.98**	**0.97**	1.00								

Table 4.8 (Continued)

Int.	**0.96**	**0.92**	**0.97**	1.00						
Conn/Int	0.09	0.17	0.14	0.05	1.00					
Conn.	**0.95**	**0.96**	**0.98**	**0.97**	0.27	1.00				
Int./Comp	-0.21	-0.20	-0.25	-0.04	-0.20	-0.12	1.00			
Meth/Comp	-0.03	0.27	0.07	0.00	0.29	0.11	-0.05	1.00		
Meth/Comp	-0.15	-0.07	-0.14	-0.03	0.53	0.06	0.71	0.16	1.00	
Conn./Comp.	0.18	0.36	0.18	0.04	0.23	0.14	-0.43	0.74	-0.18	1.00
Meth./Int.	0.55	0.47	0.51	0.53	-0.14	0.46	-0.16	-0.09	-0.26	0.08	1.00

2) Lack of experience in CO software system design may be a reason to having such results.

Table 4.9: Total Correction Effort Estimates

Actual Value	Estimated value	Residual	%Error
6	7.24	-1.24	-20.68
6	7.29	-1.29	-21.46
5	5.94	-0.94	-18.76
5	4.36	0.64	12.87
5	3.75	1.25	24.93
6	3.62	2.38	39.70
3	3.67	-0.67	-22.42
3	3.81	-0.81	-27.05
2	2.21	-0.21	-10.37
4	3.99	0.01	0.29
5	4.73	0.27	5.44
3	3.69	-0.69	-22.97
5	4.58	0.42	8.31
4	4.60	-0.60	-14.98
5	4.86	0.14	2.74
4	4.17	-0.17	-4.27
8	5.99	2.01	25.17
4	4.74	-0.74	-18.43
5	3.85	1.15	23.07
2	3.47	-1.47	-73.33
6	5.33	0.67	11.23
6	6.17	-0.17	-2.77
4	3.50	0.50	12.45
2	2.69	-0.69	-34.30
25	24.77	0.23	0.90
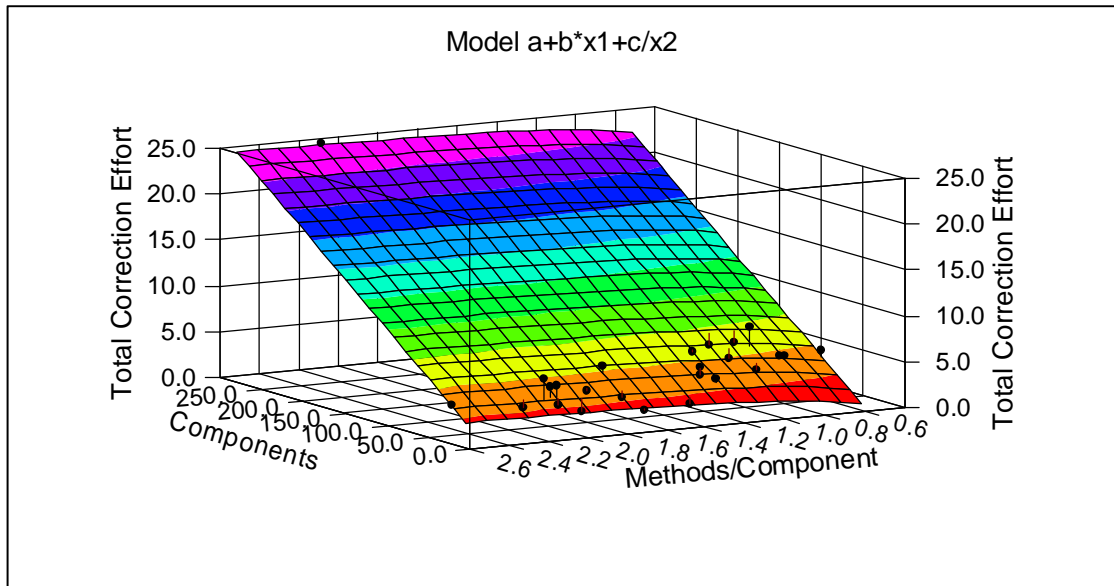
Figure 4.5: Total Correction Effort Regression Model Plot

4.3.5.6 Correction Effort Per Component Regression Model

Knowing the features of a component that may increase the correction effort is a very important issue. Once these features are identified it is possible to favor alternative designs where the correction effort elevating features are avoided. A forward addition approach has been followed where variables have been added if they satisfy a 0.05 or less confidence interval. It is our preference to obtain a model that uses more variables since that will better describe the relationships between the dependent variable and independent variables. The obtained model takes an exponential form which can be a practical disadvantage for the model. The model uses only three variables and it is defined as follows:

$$\text{Correction/Component} = \exp(-0.02*\text{Components} + 0.07*\text{PComp.} + 0.2*(\text{Methods/Comp}) - 2.37)$$

$$(p = 0.003) \qquad (p = 0.008) \qquad (p = 0.04) \qquad (p = 0.0)$$

In the model, all p-values are less than 0.05 which means the model is statistically significant. The R^2 value of 0.54 is obtained which is not too high but the average error rate results encourage the adoption of the model. Average rates started with an initial value of 18% which dropped to 12% when the outlier values are excluded from the average error estimation.

71

The model violates intuitive thinking and initial expectations in at least one or more of the followings:

1) Correction effort per component decrease as total number of components increases.

2) Some variables that are intuitively believed to increase correction effort per component have not been included in the model. Among these variables are the total number of connections, and number of connections per component. It is natural to spend more time on making corrections when the component exhibit high connectivity.

The complete list of values of actual and predicted correction effort per component values with their corresponding residual values and error rates are presented in Table 4.10. The plot of the model is shown in Figure 4.6.

Table 4.10: Correction Effort per Component Estimates

Actual Value	Estimated Value	Residual	%Error
0.07	0.08	-0.01	-17.02
0.07	0.07	0.01	7.34
0.08	0.08	-0.01	-7.89
0.08	0.07	0.02	19.65
0.17	0.14	0.03	17.37
0.22	0.18	0.04	17.52
0.11	0.13	-0.02	-18.12
0.10	0.14	-0.03	-33.48
0.17	0.14	0.02	14.45
0.13	0.14	0.00	-2.72
0.09	0.09	0.00	4.30
0.10	0.11	-0.01	-14.07
0.11	0.11	0.00	-2.01
0.08	0.11	-0.03	-40.01
0.09	0.10	-0.01	-10.17
0.11	0.13	-0.02	-16.33
0.12	0.08	0.04	34.53
0.09	0.08	0.01	8.30
0.17	0.13	0.03	20.88
0.08	0.14	-0.06	-77.68
0.12	0.09	0.03	21.51
0.09	0.09	0.01	6.52
0.13	0.11	0.02	14.67
0.10	0.13	-0.03	-26.33
0.09	0.09	0.00	-3.16
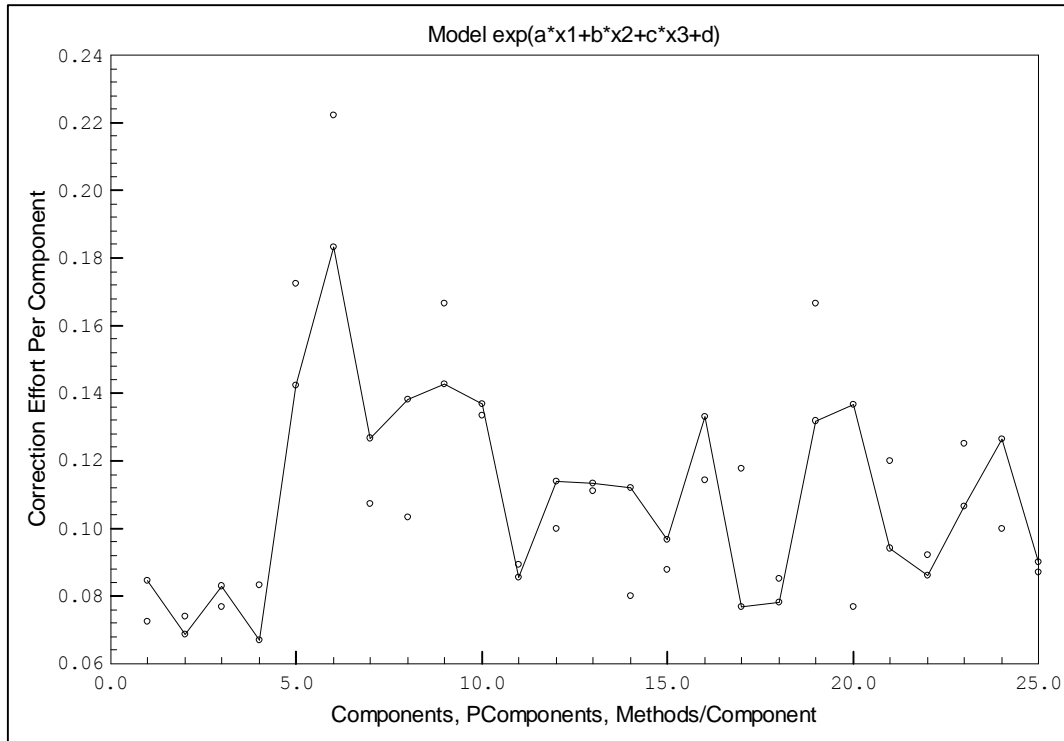
Figure 4.6: Correction Effort per Component Regression Model Plot

4.3.5.7 Integration Effort Regression Model

The CO software development paradigm focuses on building large systems by integrating pre-built components [34, 83]. Following from this, it can be clear that integration effort is a very critical factor when making a choice between alternative components. Identifying the features of a component or a CO system and their weighing factors in increasing or decreasing integration effort is a practical need. Obtaining accurate integration effort records is a necessary prerequisite to building integration effort prediction models. Also, having accurate integration effort records is only possible if implemented and composable components are available. Such components were not available during the time when this study took place. We used a different estimate of integration effort which is the effort spent on identifying component relationships, connections, and interfaces with other components. Although this is not the exact record of integration effort, it is highly related to the actual integration effort. The model is obtained as a result of applying variables in a forward addition approach. Only one variable in the model is with a

73

p-value that is less than or equal to 0.05. The model uses only the total number of connectors measure. We believe that some other variables must be related to integration effort e.g. number of interfaces but even with a p-value of less than or equal to 0.1 this measure could not be added. The reason for that is mostly related to strong multicollinearity between the different variables. The obtained model has the following form:

$$\text{Integration Effort} = 0.1 * (\text{Connectors}) + 2.6$$
$$(p = 0.0) \qquad\qquad (p = 0.0)$$

Despite the fact that the model contains only one variable, it still bears both statistical and practical significance. Both the p-value of the variable coefficient and the constant are equal to 0.0. R^2 has a value of 0.91 which is also quite high to encourage the adoption of the model. Average error rates are 17% and 11% for the initial estimate and the estimate without outliers respectively. A complete listing of the actual and predicted integration effort estimates with their corresponding residual and error rate values are presented in Table 4.11 and the model plot is shown in Figure 4.7.

Table 4.11: Total Integration Effort Estimates

Actual Value	Estimated Value	Residual	%Error
7	11.09	-4.09	-58.43
5	7.89	-2.89	-57.87
4	5.87	-1.87	-46.73
4	5.55	-1.55	-38.74
6	5.23	0.77	12.84
8	8.43	-0.43	-5.33
5	5.55	-0.55	-10.99
5	4.48	0.52	10.32
4	4.91	-0.91	-22.76
7	6.93	0.07	0.93
8	8.43	-0.43	-5.33
4	4.70	-0.70	-17.43
11	7.25	3.75	34.05
8	8.53	-0.53	-6.66
10	8.96	1.04	10.41
8	8.21	-0.21	-2.66
10	8.43	1.57	15.74
8	7.68	0.32	4.00
10	7.25	2.75	27.46
4	4.70	-0.70	-17.43

Table 4.11 (Continued)

8	5.44	2.56	31.96
8	6.30	1.70	21.31
7	7.36	-0.36	-5.15
6	6.30	-0.30	-4.92
35	34.53	0.47	1.34
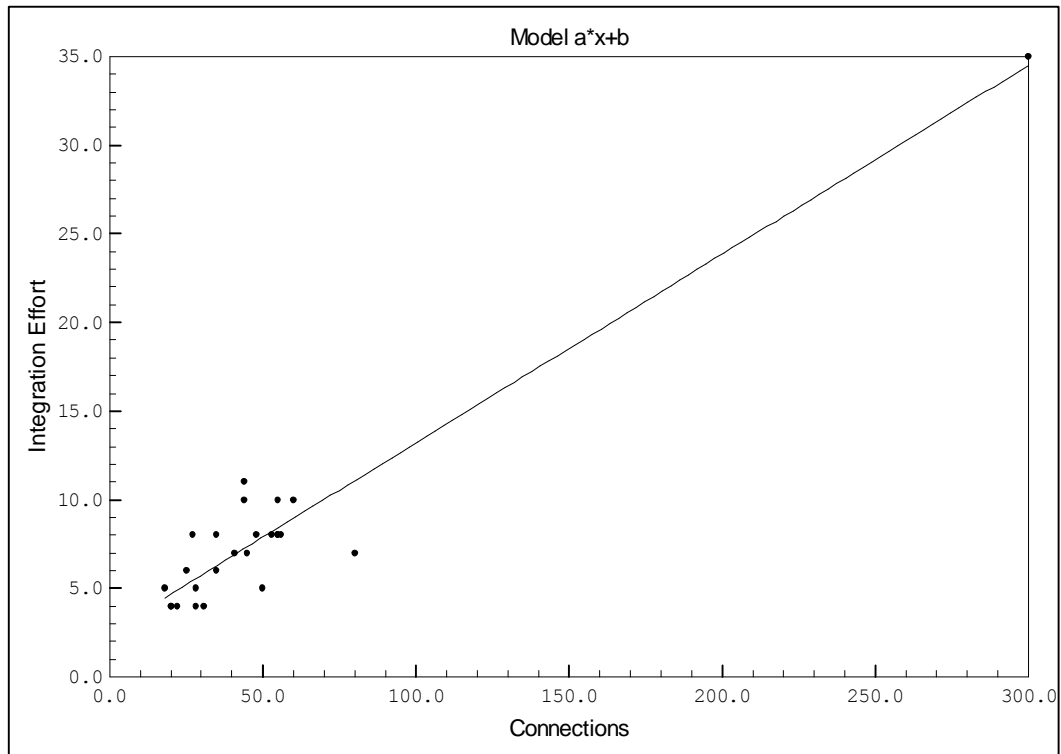
Figure 4.7: Total Integration Effort Regression Model Plot

4.3.5.8 Integration Effort per Component Regression Model

As well as total integration effort, integration effort per component is a measure of great practical importance. Besides the deemed functionality and performance requirements, one of the main factors for making a decision whether to buy a pre-built component or not is whether it is easy to be integrated with other components or not. The degree of easiness to integrate can be best quantified by estimating the time spent on integrating the component. In this part we tried to identify the factors that increase or decrease integration effort of a component and their weights in a backward elimination approach. We started with all variables in the model and

eliminated all variables that have p-values greater than 0.05. The final model with all variables having a p-value less than or equal to 0.05 is defined as follows:

Integration/Component = -0.003 * (TNI) + 0.08 * (TNM/TNC)+ 0.02 * (TNCO/TNC) + 0.11
$$(p = 0.01) \qquad (p = 0.0) \qquad (p = 0.001) \qquad (p = 0.01)$$

It is clear that the model is statistically significant since all p-values are less than 0.02. The model meets intuitive thinking in that integration effort of a component increases when both the number of methods and/or the number of connections in that component increase. One problem of the model is that integration effort per component decreases as the total number of interfaces increases. It is also possible to interpret the result as: A well planned component framework should dedicate more interfaces for composability. According to this interpretation, increasing number of interfaces may reduce integration effort as they provide help in integration. Also, the coefficient value is too small (0.003) making its effect to be insignificant in the model.

The average error estimates are 16% and 8% for the initial estimate and the second estimate (after removing outliers) respectively. A complete list of the actual and predicted results with their corresponding residual and error rate values are presented in Table 4.12. The model plot is shown in Figure 4.8.

Table 4.12: Integration Effort per Component Estimates

Actual Value	Estimated Value	Residual	%Error
0.14	0.24	-0.09	-66.46
0.20	0.27	-0.07	-34.82
0.14	0.23	-0.08	-59.01
0.20	0.22	-0.02	-9.04
0.38	0.34	0.03	8.98
0.36	0.39	-0.02	-6.28
0.36	0.39	-0.03	-8.93
0.36	0.34	0.01	3.81
0.33	0.36	-0.03	-7.91
0.50	0.49	0.01	1.65
0.33	0.28	0.06	16.98
0.33	0.33	0.00	1.25
0.38	0.28	0.10	26.66
0.24	0.26	-0.01	-5.77
0.29	0.25	0.04	14.88
0.33	0.38	-0.05	-15.41
0.33	0.26	0.07	20.52
0.38	0.41	-0.02	-6.44

Table 4.12 (Continued)

0.42	0.41	0.00	0.71
0.29	0.33	-0.04	-15.16
0.38	0.31	0.07	17.81
0.27	0.25	0.02	7.70
0.35	0.36	-0.01	-2.60
0.43	0.37	0.06	13.23
0.24	0.23	0.01	5.30
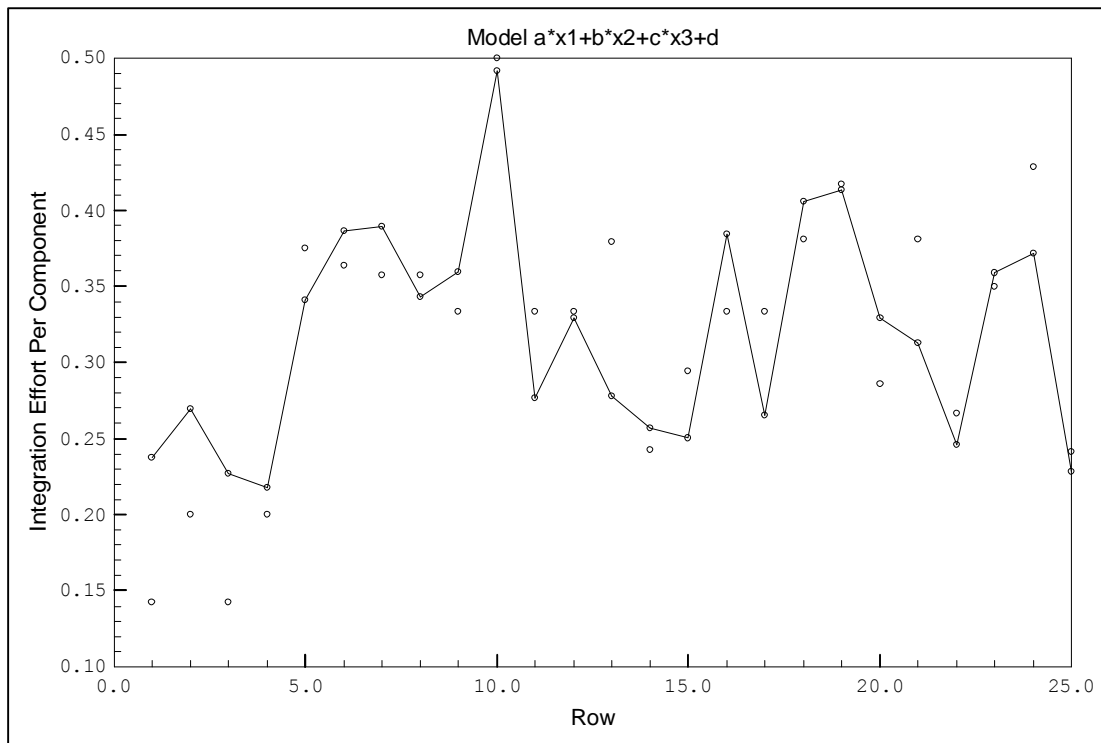
Figure 4.8: Integration Effort per Component Regression Model Plot

4.3.5.9 Productivity (FP/Person-Hour) Regression Model

Developer productivity is one of the most important features that influence total development effort. Finding out the system features that influence developer productivity has been a critical issue since the early days of software engineering. In our research we identified the intuitive system features that may potentially affect productivity and followed a backward elimination approach in which variables with p-values that are greater than 0.05 are eliminated. The final model is defined as follows:

$$\text{Productivity} = 5.41 \ - \ 0.82 * (\text{TNM/TNC}) - 0.95 * (\text{TNI/TNC})$$
$$(p = 0.0) \quad (p = 0.04) \quad\quad (p = 0.01)$$

The model shows that increased number of methods in a component is favored over increased number of interfaces. It suggests that interfaces with more methods are better than having too many interfaces with small number of methods.

Despite the fact that the model variables are all statistically significant, the model did not produce good practical results. Average error rates are 34% and 23% for the initial and second (after excluding outliers) estimates respectively. A complete list of the results is presented in Table 4.13 and the model plot is shown in Figure 4.9.

Table 4.13: Productivity Estimates

Actual Value	Estimated Value	Residual	%Error
3.74	2.84	0.90	24.08
4.56	3.52	1.04	22.82
4.50	3.17	1.33	29.54
5.31	3.63	1.68	31.58
2.75	2.42	0.33	12.00
2.69	2.42	0.27	10.11
3.25	2.00	1.25	38.61
1.69	2.22	-0.53	-31.21
1.47	2.18	-0.71	-48.29
2.75	1.49	1.26	45.73
2.38	3.52	-1.14	-48.11
2.42	2.64	-0.22	-9.19
2.79	2.66	0.13	4.59
2.06	3.07	-1.01	-49.21
2.50	3.15	-0.65	-25.90
1.80	2.00	-0.20	-11.08
2.22	3.13	-0.91	-41.13
1.84	1.60	0.24	12.98
0.58	0.98	-0.40	-68.54
1.58	2.49	-0.91	-57.35
3.14	2.13	1.01	32.19
2.07	2.85	-0.78	-37.50
1.88	2.33	-0.45	-23.90
2.40	2.60	-0.20	-8.21
0.97	2.30	-1.33	-137.55
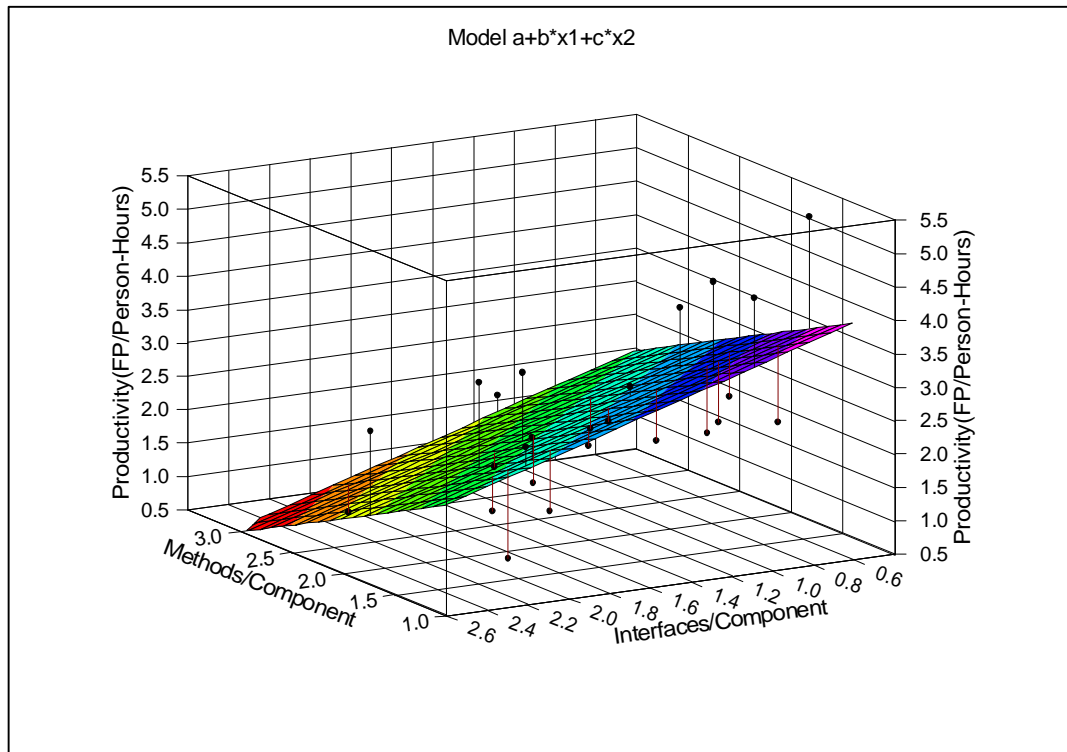
Figure 4.9: Productivity Regression Model Plot

4.3.5.10   Total Development Effort Regression Model

The importance of predicting development effort at early stages of the development process is very clear [81]. Several cost estimation methods have been proposed in the literature. These methods vary in their nature. Some methods are parametric like Putnams model (SLIM) first described in [67] , and PRICE-S which has been partially described in [63] and used by the DoD and NASA in their project estimations. Another widely cited cost estimation method is the COCOMO model which was first described in [14] and then revised to address the new changes in the software development life cycle and released as COCOMO II [15].

Expert judgment is one other approach that has been applied in software cost estimation techniques. In this approach software cost estimation is done based on the previous experience and practices in software development. Expertise makes predictions based on outcomes of his/her past projects. One known expert judgment technique is Delphi as described in [44].

79

Learning oriented techniques for cost estimation have been proposed. These models are mainly dependent on neural networks models that are based on previous experiences.

Regression-Based techniques have been successfully used in software cost estimation. The common features in these methods depend on Least Squares regression where a set of independent variables (regressors) are identified and a prediction model is obtained based on previous projects data. Boehm used regression models to calibrate COMOMO II.

In this study we built a regression based model for effort prediction based on effort estimation data obtained by a comparative estimations tool prepared by the International Software Benchmarking Standards Group (ISBSG). ISBSG is a non-profit organization whose main aim is providing help to improve the management of IT resources. It maintains two repositories for: 1) Software Development and Enhancement and 2) Software Maintenance and Support. The repositories maintain data of more than 3000 projects sponsored by software development organizations mostly from USA, Japan, Australia, and several other countries. The effort estimation tool performs predictions based mainly on FP counts. Other parameters are also necessary to make accurate predictions. These parameters are:

- *functional size range*
- *development type*
- *development platform*
- *business area type*
- *application type*
- *maximum team size*
- *language type*
- *primary programming language*
- *user base – business units*
- *user base – locations*
- *user base – concurrent users*
- *used CASE tool*
- *used methodology*
- *how methodology was acquired*

While using the tool, for all projects the following parameters were set to "match none":

- *language type*
- *primary programming language*
- *user base – business units*
- *user base – locations*
- *user base – concurrent users*
- *used CASE tool*

The *development type* parameter was set to "new development". The *development platform* parameter was set to "PC". The maximum team parameter size was set to "2" since all system models were prepared by teams of two. The *used methodology* parameter was set to "Yes" since all developers followed the component oriented software development approach. And the *how methodology was acquired* parameter was set to "built in-house".

The regression model is obtained with a significance level of 85% and is defined as:

Development Effort = 2.4 * (# Comp.) + 26.9 *(# PComp) – 6.2 * (# Int) + 9.8 *(Conn./Comp)
$(p = 0.05)$ $(p = 0.001)$ $(p = 0.15)$ $(p = 0.04)$

Besides being statistically significant, the model also bears great practical importance due to the following reasons:

1. The model has prediction power with an average error rate of 9% only.
2. Predicting development effort using complexity metrics that can be collected at the design phase of the system development process will enable managers and developers to make better decisions related to the product and process.

The predicted values compared to actual values with their corresponding estimated residual and error percentage are presented in Table 4.14. The estimated error rates are 17% and 9% for both the initial estimate and the second estimate after removing the outliers. The model plot is shown in Figure 4.10.

Table 4.14: Total Development Effort Estimates

Actual Values	Predicted Values	Residual	Error%
480	481.43	-1.43	-0.30
559	485.36	73.64	13.17
365	380.23	-15.23	-4.17
287	324.86	-37.86	-13.19
253	239.89	13.11	5.18
391	315.89	75.11	19.21
225	203.46	21.54	9.57
152	213.58	-61.58	-40.52
133	141.81	-8.81	-6.62
223	234.01	-11.01	-4.94
204	375.89	-171.89	-84.26
220	205.22	14.78	6.72
364	306.38	57.62	15.83
235	362.87	-127.87	-54.41
421	377.40	43.60	10.36
349	268.12	80.88	23.17
495	396.68	98.32	19.86
268	246.56	21.44	8.00
167	196.25	-29.25	-17.52
145	209.36	-64.36	-44.39
246	263.97	-17.97	-7.30
356	371.32	-15.32	-4.30
230	227.85	2.15	0.93
184	197.23	-13.23	-7.19
1953	1948.67	4.33	0.22
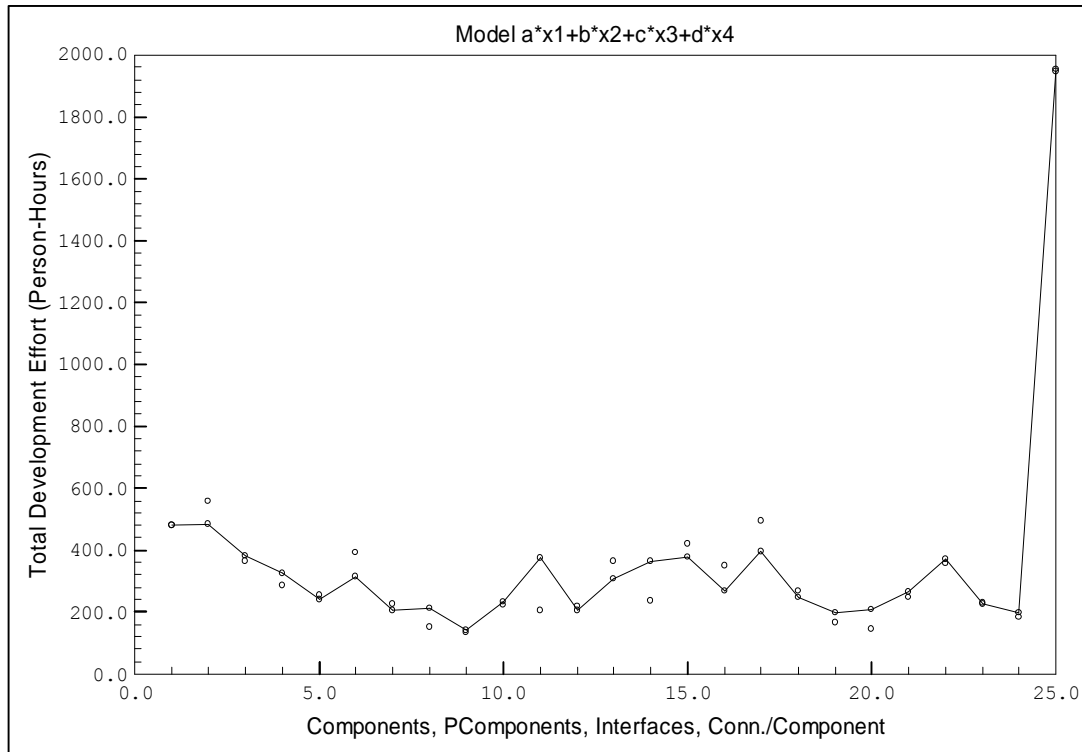
Figure 4.10: Total Development Effort Regression Model Plot

4.3.7 Summary of the Results

Ten different models have been developed. While building all of these models, both statistical significance and practical importance have been taken into consideration. For statistical significance purposes, apart from the first model, in all of the models variables were only added if their corresponding p-values were less than or equal to 0.05. In the first model a variable (# of connectors) is added while its corresponding p-value is equal to 0.1 and in the total development effort prediction model a variable with p-value is added to the model with p-value of 0.15. The reason behind this exception is due to the believed inherent importance of the variable in the estimation of FPs count. We have considered practical perspectives, such as average error rates and simplicity of the model. In two models exponential functions have been used since they had the lowest average error rates compared to the alternatives. A summary of all the developed regression models with their corresponding R2 values and average error rates are presented in Table 4.15.

Table 4.15: Summary of the Regression Models

Estimated Factor	Used Elements	Regression Model	Correlation	%Error 1	%Error 2
FP COUNT	ALL Componets, PComponents, Interfaces, Connectors	0.8 * (# Components) + 4.3 * (# PComponents) - *(p = 0.01)* *(p = 0.03)* 1.8 * (# of Interfaces) + 0.5 * (#Connectors) + 12.7 *(p = 0.06)* *(p = 0.1)* *(p = 0.02)*	0.97	21	8
FP/Interface	Methods/ Int Connections/ Interface	5.2 – 8.5/(Methods/Int)+ 0.94 * (Connectors/Int) *(p = 0.004)* *(p = 0.007)* *(p = 0.04)*	0.43	22	17
Total Design Effort	All Components, Methods, Connections, Methods/ Component	-0.65 * (# Comp.) + 0.83 * (#methods) *(p = 0.0004)* *(p = 0.0)* + 0.73 * (# of Connectors) – 33.9 * (Meth./Comp) *(p = 0.0006)* *(p = 0.0004)*	0.98	27	15
Average Design Effort/Component	All Components, PComponents, Interfaces/ Component, Methods/ Component	exp(-0.03*Components) + 0.13*(PComponents) + *(p = 0.0)* *(p = 0.0)* 0.46 * (Int. / Comp) + 0.32 *(Meth / Comp) - 1.48) *(p = 0.0)* *(p = 0.0)* *(p =0.0)*	0.79	20	13
Total Correction Effort	All Components, Methods/ Component	0.08 * (# Comp.) - 1.5 / (Meth./Component) + 2.13 *(p = 0.0)* *(p = 0.02)* *(p = 0.0)*	0.95	19	13
Correction Effort/Component	All Components, PComponents, Methods/ Component	exp(-0.02 * (# Comp) + 0.07* (PComp) *(p = 0.003)* *(p = 0.008)* + 0.2 * (Methods/Comp) – 2.37) *(p = 0.04)* *(p = 0.0)*	0.54	18	12

Table 4.15 (Continued)

Integration Effort	Connectors	0.1 * (Connectors) + 2.6 *(p = 0.0)* *(p = 0.0)*	0.91	18	11
Integration Effort/PComponent	Interfaces, Methods/ Component, Connections/ Components	-0.003 * (# Interfaces) + 0.08 * (Meth./Comp) *(p = 0.01)* *(p = 0.0)* + 0.02 * (Conn./Comp.) + 0.11 *(p = 0.001)* *(p = 0.01)*	0.67	16	8
Productivity (FP/Hour)	Methods/ component, interfaces/ component	- 0.82 * (Meth./Comp.) – 0.95 * (Int./Comp) + 5.41 *(p = 0.04)* *(p = 0.01)* *(p = 0.0)*	0.35	34	23
Total Development Effort	Components, Pcomponents, Interfasces, Connections/ Component	2.4 * (# Comp.) + 26.9 *(# Pcomp) – 6.2 * (# Int) + (p = 0.05) (p = 0.001) (p = 0.15) 9.8 *(Conn./Comp) (p = 0.04)	0.96	17	9

The results obtained from the experiment can be classified into three broad classes as follows:

1. Meeting initial (intuitive) view points: regressors described regressands in a manner meeting intuitive thinking (e.g. FP should increase with increased number of methods).

2. Violating initial view points: the relationships between regressands and regressors do not meet intuitive thinking (e.g. an increase in the total number of interfaces results in a decrease in the total FP count).

3. No results (unexpected behavior): No relationship –with statistical significance- could be detected between regressands and regressors which are intuitively believed to be related.

The results obtained from the regression analysis revealed that complexity metrics collected from the design models can be great managerial and practical use. While some of the results slightly violate initial expectations, most of the obtained results sound reasonable. Some metrics which are intuitively believed to be related to some process features (regressands in our study) could not be added to the regression models due to their corresponding p-values which were above the maximum acceptable value of 0.1. In Table 4.16 we present a summary of the metrics collected form the projects used in our study with their practical influences. In the column *Influence* the symbol "↑" is used to mean that when the metric value increases the estimated feature value increases as well. The symbol "↓" means that the estimated feature value decreases when the metric value increases. In the column *Comments* we comment on whether the metrics use meets initial expectation that are built based on intuition. If the metrics use violates the initial expectations then a short reasoning is provided when relevant.

Table 4.16: Summary of Metrics Practical Applications

No	Metric Name	Used Model	Influence	Comments
1	Total Number of Components	Total FP Count	↑	Agrees with Intuition
		Total Design Effort	↓	Violates intuition due to multicollinearity
		Design Effort/ Comp.	↓	Violates intuition due to multicollinearity
		Total Correction Effort	↑	Agrees with Intuition
		Correct. Effort/ Comp.	↓	Violates intuition due to multicollinearity

Table 4.16 (Continued)

2	Total Number of Implementation Components	Total FP Count	↑	Agrees with Intuition
		Design Effort/Comp	↑	Agrees with Intuition
		Total Correct. Effort	↑	Agrees with Intuition
3	Total Number of Interfaces	Total FP Count	↓	Violates intuition due to multicollinearity
		Total Integration Effort	↑	Agrees with Intuition
4	Total Number of Methods	Total Design Effort	↑	Agrees with Intuition
5	Total Number of Connections	Total FP Count	↑	Agrees with Intuition
		Total Design Effort	↑	Agrees with Intuition
		Total Integration Effort	↑	Agrees with Intuition
6	Total Number of Events In	Not used in any model	-	Events In are not utilized in the models
7	Total Number of Events out	Not used in any model	-	Events Out are not utilized in the models
8	Maximum Depth of the Composition Tree (DCT)	Not used in any model	-	Almost all projects have similar DCT value. DCT lost its discriminative power
9	Maximum Width of the Composition Tree (WCT)	Not used in any model	-	Almost all projects have close WCT value. WCT lost its discriminative power

Table 4.16 (Continued)

10	Average number of Interfaces Per Component	Design Effort/ Comp.	↑	Agrees with intuition
		Productivity (FP/Person-Hour)	↓	Agrees with intuition
11	Average Number of Methods Per Component	Total Design Effort	↓	Violates intuition due to correlation with # of interfaces
		Design Effort/Comp	↑	Agrees with intuition
		Integ. Effort/ Comp.	↑	Agrees with intuition
		Productivity	↓	Agrees with Intuition
12	Average Number Connections Per Component	Total Integration Effort	↑	Agrees with intuition
13	Average Number of methods Per Interface	Average FP count per Interface	↑	Agrees with intuition
14	Average number of Connectors per Interface	Average FP count per Interface	↑	Agrees with intuition

In all of our models we used average values rather than exact values per component. We tried the possibility of analyzing individual components and examine the relatedness of complexity metrics with productivity, design-effort, correction-effort, and integration-effort based on records of individual components. This part could not be completed successfully due to the fact that developers could not provide exact records for the design, integration, and correction efforts they spent on individual components.

CHAPTER 5

AUTOMATING METRICS COLLECTION PROGRAMS

5.1 The Need for Metrics Automation

Computer Aided Software Engineering (CASE) is a generic term that is widely used for the different tools used in software development. CASE tools play an important role in modern software engineering practices and have an important influence in the production of cost-effective and efficient software systems. Today, CASE tools are used in the different stages of the software process. Project planning and scheduling tools help in scheduling and organizing activities in the software process (e.g. MS Project® from Microsoft, and ManagePro™ from Performance Solutions Technology). Effort estimation tools are used at early stages and particularly during and just after the requirements definition of the software process to make predictions about total development effort ad then total cost (e.g. Effort Estimation Toolkit from ISBSG). System modeling tools are used during the requirements specifications and system and software design stages of the software process. Several modeling tools have been developed and widely used by software developers. Among the most widely used modeling tools is IBM Rational Rose that is used for creating UML models for OO systems. Besides being a powerful modeling tool, Rational Rose also supports both reverse engineering and automatic code generation. Another UML modeling tool is Visio® from Microsoft. Several other types of CASE tools can be used at different stages of the development process like report generating tools. Integrated development environments (IDEs) help in automating many programming

processes. Debugging, system integration, and testing tools are also available in many flavors.

Use of product complexity metrics in software engineering as a primary means for making process and product related decisions has been utilized mostly in the last few years. Automatic metrics collection tools have developed for OO software development and particularly for CK (Chidamber and Kemerer, 1994) metrics. Some of these tools have been embedded into modeling tools and enabled automatic metrics collections directly from the system models.

On the other hand, lack of dedicated CASE tools for COSE is obvious. Apart from some extensions to OO modeling tools, no commercial dedicated CASE tools for COSE ever existed. COSECASE is a dedicated COSE tool which is developed in consecutive versions each of which was a part of a master thesis work in the department computer engineering in the Middle East Technical University-Turkey. Each new version introduced represents an enhancement to its predecessor version. The last version of COSECASE is functional. It enables model creation, and performs rule violations checks in the following situations: 1) when creating a relationship between components, 2) removing a component from the model, 3) adding a new component to the model. The final version of the tool was lacking some usability related enhancements. As part of the research described in this thesis, some usability related enhancements have been added to the tool. Our contributions to the tool can be summarized in three main groups:

1) Usability Enhancements: The enhancements to the tool usability include the followings:
   a. Enabling automatic resizing of components in the model
   b. More nice-looking connectors.
   c. Deleting using keyboard delete key in addition to mouse right button.
   d. Dialogue boxes of components properties have been modified.

2) Automatic metrics collection: Complexity metrics defined in Chapter 3 are automatically collected during model creation.

3) Producing estimation results: Prediction models that are developed based on complexity metrics in Chapter 4 are used to make process related estimations. The tool automatically produces estimations based on these models and the product metrics and can target the results to both screen and/or a text file. Also for system models prepared using other tools it is possible to produce estimations after manually estimating the complexity metrics and entering their values in the corresponding fields if the form that is appearing in Figure 5.1.

Figure 5.1: Metrics Collection and Estimation Tool

5.2 Enabling Automated Metrics Collection in COSECASE

COSECASE provides a good environment for creating COSE system models only based on the COSEML (Dogru and Tanik, 2003) notation. Screen shot of the tool is shown in Figure 5.2.
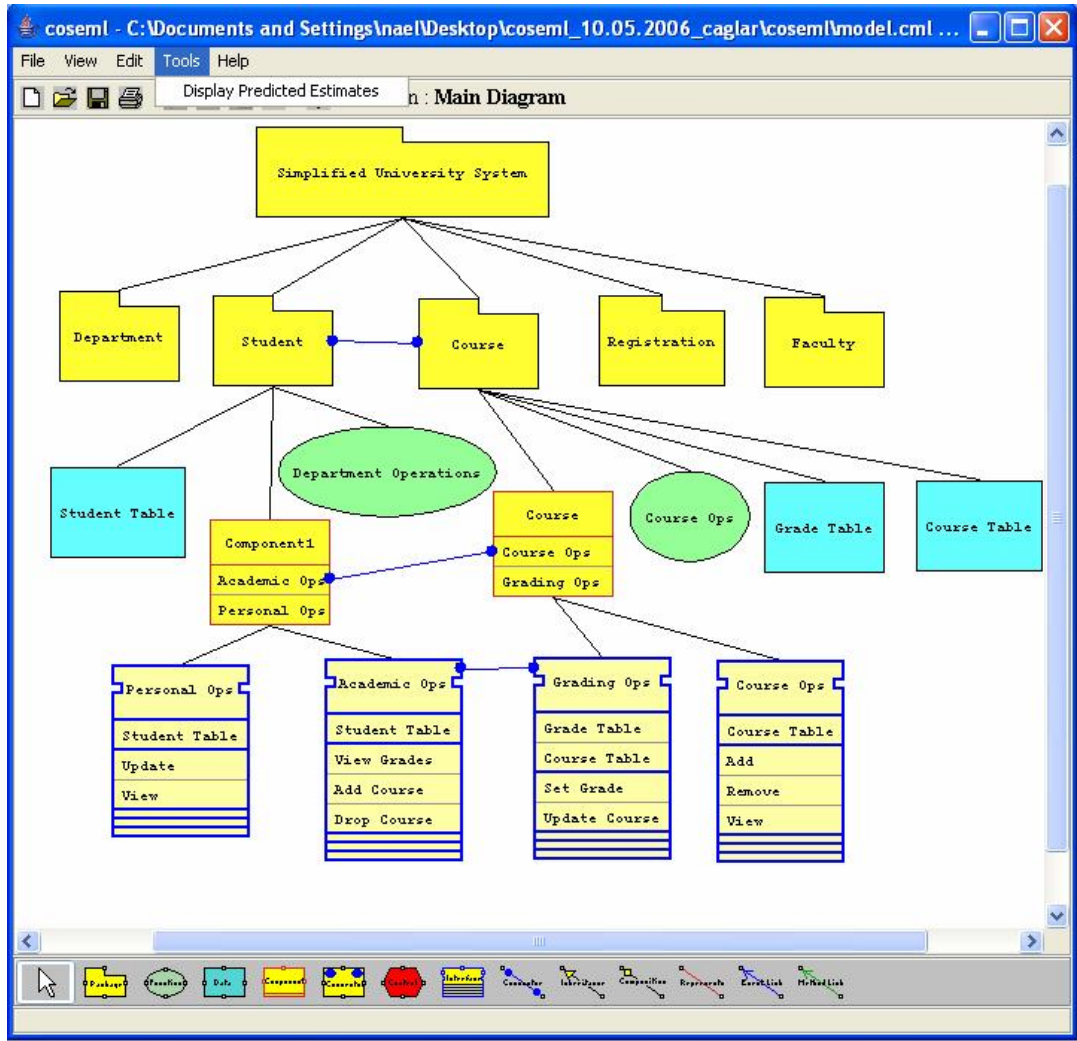
Figure 5.2: Screen Shoot from COSECASE with Estimations Options

After creating or loading an already existing COSEML model, a user will be able to obtain predicted estimations by just selecting "Display Predicted Estimates" menu item from the "Tools" menu. The results of estimations are displayed in a form as shown in Figure 5.3. The user will have the chance to save results by just clicking on the "Save Results" button.

Figure 5.3: Screen Shoot of Estimation Results Form

CHAPTER 6

CONCLUSION AND FUTURE WORK

A measurement framework for Component Oriented Software Engineering has been developed and investigated. This was to support the newly developing radical Software Engineering approaches that are expected to offer a long waited answer. Besides the lacking industrial experimentation, our results obtained through statistical analyses over academic case studies that extended three years, yield valuable conclusions. Some relations among process and product properties and proposed metrics have been founded. Besides, the converging analysis results in many aspects are an indication of the validity of the foundation.

6.1 Summary

Quantifiable aspects of CO systems are identified. Then, metrics to measure these aspects are defined. Relationships of metrics with the aspects they are intended to quantify have been defined. For every defined metric, its potential impacts on the product and the process have been presented. These potential impacts represent the initial viewpoints based on intuition, previous experiences of software metrics, and related work on software metrics performed by other researches in the field. A set of properties that a CO complexity metrics must possess are defined and justified. The proposed metrics have been evaluated against the properties defined in this thesis and then evaluated against another set of properties defined in [84]; all defined metrics qualified and satisfied all properties in both sets.

A Complexity model has been defined for CO software system in three levels. At the lowest level, complexity aspect related to methods is included. At the intermediate level, component-related complexity aspects are included. The highest level of complexity in CO systems is the Overall System Complexity

(OSC). OSC is the complexity that can be estimated from quantified aspects of components plus an added complexity resulting from bringing the components together into a single system.

To explore the validity of metrics from a practical perspective an experiment has been performed. Complexity metrics are important because they are known to be important players when making process and/or product related decisions. Metrics-based regression models are developed. These regression models are all prediction models that enable making predictions about: Size (as a function of FP count), development effort (Person-hours), integrability (as a function integration effort), and maintainability (as a function of correction effort). While the experiment revealed that metrics can be highly dependable in making process and product related predictions, it suggests that further research covering more project data should continue.

6.2 Discussion of the Results

The significantly notable (expected or unexpected) results can be summarized in the following points:

1. Total number of components of a system is an important factor in predicting Total FP count, Total development effort, and Total correction effort, of that system. This result meets initial viewpoints and expectations. Fixing all other parameters, "Total design effort" decreases when total number of components increases. This outcome seems to violate initial expectations and even intuition. On the other hand, assuming the same overall functionality, a system with more components may take less effort to design, when compared to another with less components. In some cases, the integration among a few components may be difficult, probably due to increased fan of services and their connections. A bigger set of smaller connections would be less complex than a small set of bigger (complexity) connections. This follows the famous complexity relation which states that the overall complexity is conserved but it is possible to be moved to where it is easier to manage.

2. Total number of interfaces in a system is an important factor in predicting total integration effort. An increase in total number of interfaces increases total integration effort. This result is within expectation and meets intuitive thinking; hence more interfaces in a system means more relationships exist between components and then more effort will be required to integrate them. On the other hand, the detected relationship between total FP cont and total number of interfaces is unexpected and violates intuitive thinking. An increase in the total number of interfaces results in a decrease to the total FP count in a system. This outcome needs to be further examined and validated. This result also may imply tendency to reduce interfaces, as total complexity increases: The experimentation considered declared (created) components rather than being industry-wide available. Students may have chosen simpler connections for bigger projects. A natural consequence would be expecting well established domains where complex systems will be built by highly cohesive and relatively larger-grained components that require less connectivity.

3. No relationship between the depth of the composition tree and any of the checked regressands could be detected. The reason behind that can be due to the fact that all projects included in the experiment have almost similar values for depth of the composition tree metric. An opportunist speculation would advise developers to freely choose their preferred decomposition. Their ideal decomposition would not affect the complexity.

4. Total number of connectors' relationship with total FP count, total design effort, total integration effort, and total development effort meets intuitive thinking and within expectations. An increase in the total number of connectors in a system results in an increase in all values of these variables.

5. Numbers of Events In/Out in a system are not used in any regression model. The reason can be due to the fact that uses of events in and events out were not utilized well in all designs used in the study.

While developing all of the regression models attention was paid to both statistical significance and practical importance of the model. Statistical significance has been monitored through R^2 values and coefficients corresponding p-values.

Practical importance of each model has been assessed through average error rates in the predicted values when compared to actual values. Even in the worst case, average error estimates remained below those obtained in similar studies. Sommerville [81] reported that during the design phase cost estimation techniques can have an error rate of 50%. In all of the regression models we developed average error rates were less than 25% when removing the outliers. The most important two challenges we encountered during the research are the unavailability of industrial projects and the lack of standard definitions to the terms of interest.

6.3 Comparison with Related Works

Component Orientation is a new trend towards software development. The process model described by Dogru and Tanik [34] and the COSEML language represent one of the earliest and serious works in the field. The work presented in this thesis focused mainly on identifying a metrics set characterizing complexity in CO systems that are developed using the COSE approach. The metrics are validated using experimental data and the results of the validation showed that metrics can be of great value in predicting several critical and important process and product features. The literature of software measurement presents several trials of evaluating software complexity and other works for relating systems complexity with one or more process or product aspects. The following reasons are enough to enhance the originality of this work:

1) A survey of the literature revealed that there is no complete pure CO measurement framework that aims at characterizing software complexity in CO systems and find its relationship with process and product features.

2) The works presented in the literature were either considering black-box reusability or white-box reusability of software components while in our methodology we defined metrics for both cases. Complexity is characterized with a set of metrics in the case of black-box reuse and another set of metrics is used for white-box reuse.

3) The approach presented here can not be generalized to all component methodologies. It is specific to COSEML only. Among several others, we

believe that COSE paradigm will be the future trend of CO software development.

In the following discussion we will describe the most widely known CO related metrics and their similarities and differences with our model.

1) Chidamber, Darcy, and Kemerer [22] investigated the relationships between programmer productivity, design effort, and re-work effort and the OO complexity metrics widely known as CK [20, 21]. The results obtained in their research support the idea of that complexity metrics can be used as predictors of some critical and process aspects. The main outcome of this work is that lower productivity, greater re-work effort, and greater design effort are highly associated with high coupling and low cohesion values. These outcomes meet intuition and strengthen the arguments of that excessive coupling is not a good design feature while cohesion is a deemed feature. The main problem with the outcomes is that they are highly dependent on the LCOM metric which is widely known to be ill-defined. The main similarity of this work with our work is that both works try to build prediction models of process features using complexity metrics collected during the design phase of system development. The main difference between this work and our work is that Chidamber et al. [22] findings are OO specific while our findings are CO specific. In our methodology it is revealed that high connectivity (a measure of coupling) is also associated with more rework and development efforts.

2) Cho, Kim, and Kim [112] proposed metrics for measuring complexity of software components. They assumed white-box reuse of components and measure complexity mainly based on the Cyclomatic complexity [57]. The method does not assume pure component orientation; it rather views a software component more similar to a class as defined in OO software development. Four types of component complexity are defined but the influence of these complexities on product and process features is not discussed making interpretation difficult to achieve. The similarity between this method and our method is that it considers high connectivity between

components as a dangerous feature which will influence maintainability, reliability, and other product aspects.

3) Goulao and Abreu [106] proposed metrics that cover composition of components. They defined metrics for measuring the ratio of used services to total services provided by the component and for measuring the interaction density between components. The method is similar to our work in that it suggests that high interaction density between components increases overall system complexity. This result is validated for our model while it is left without validation in the other model. The method is specific only to CORBA components and cannot be generalized to other component models. This fact makes the method different from ours since our method is specific to COSEML approach that is more generic.

4) Mahmood and Lai [107] presented a model for measuring complexity in UML components. The model assumes only black-box reuse of components and thus focusing on characterizing system complexity only using component interface specifications and interactions with other components. The main difference between this method and ours is that is UML specific while our method is COSEML Specific. Another difference is valuable to discuss that is our method deals with both black-box and white-box reuse while this method deals only with black-box reuse. The metrics are not validated using project data. It also lacks interpretation guidelines.

5) Banker, Datar, Kemerer, and Zweig [108] investigated the influence of several complexity metrics on maintenance costs and found that complexity measures significantly affect maintenance costs. The study considered several complexity measures defined for traditional software development. The main similarity between this method and ours is that in both methods is that high complexity is highly related with rework effort.

6) Lindval, Tvedt, and Cost [109] presented an approach for detecting the relationship between system architecture and its maintainability. The method characterizes system architecture at two phases of the development. Early architecture is the system architecture at the design phase and late architecture is the system architecture after development. They defined

new metrics of system architecture based on the CBO metric from CK set [20, 21]. The metrics they defined differentiate between inter-module coupling and intra-module coupling. The method is described for classes for OO development and module (close to component definition). The main outcome shows that loosely coupled designs are easier to maintain. This result meets our results for COSEML where we assume high connectivity increases rework effort.

7) Darcy, Kemerer, Slaughter, Tomayko [100] examined different measures of system structural complexity based on coupling and cohesion. Both this method and our method are similar in that they focus on structure. This method completely ignores algorithmic complexity while our method considers algorithmic complexity for the case of which-box reuse of components. Also the view of a component in this model is close to the class concept in object orientation while our model relates to pure component orientation. Results obtained in both our model and this model are similar in that they both suggest high coupling negatively influences maintenance effort. Darcy et al. [100] found that coupling and cohesion should be considered jointly and suggest that individual measures of coupling and cohesion can be useless.

8) Keating [50] introduced a model of complexity based on system structure and hierarchy. The system complexity is evaluated based on the degree of connectivity. Keating proposed some guidelines that will decrease complexity by imposing limitations on the number of modules at any level to 7 ± 2. Keating findings were not supported by experimental investigations making them less dependable.

9) Qiam, Liu, and Tsui [110] proposed a metric for evaluating decoupling for service components. The model assumes pure black-box reuse of service components. They evaluated decoupling using metrics of state dependency that tells the degree to which the service is stateless and dependency on other services. Although the authors claim that these metrics can be used to measure understandability, maintainability, reliability, testability, and reusability of service components, this claim remained without empirical validation. The metrics are well defined but no interpretation guidelines

100

were provided. The importance of these metrics is mainly due to the fact that they are among the few metrics that assume pure component orientation.

10) Braha and Maimon [111] introduced two measures for structural complexity and functional complexity of modules. The structural complexity is used to estimate total assembly effort (assembly effort can be related to integration effort in our model). The method is not proposed for component orientation and deals with white-box reuse only.

11) Dumke and Winkler [114] described a framework of measurement in component based software development. The described framework is dependent on OO software development where software components are used during the system integration phase. The framework is mainly validated for Java-based software development. Dumke and Winkler suggest the measurement process should start with the selection of metrics, then, the identification of thresholds for metrics values should follow. Then, the selected metrics should be adapted and refined to fit to the given paradigm. The last step in measurement is identified as the automation of the measurement process and experimental validation of the metrics. The measurement framework proposed is not for component oriented systems. It is for component based products and processes that rely on the OO paradigm.

A summary of these methods and their comparison to our method is given in Table 6.1.

Table 6.1: Summary of Related Works

No.	Brief description	Support for CO	Similarities	Differences	Weaknesses
1	Estimates Productivity, design effort and rework effort using OO complexity metrics	No	Use metrics to estimate productivity, design and rework efforts	Not CO.	Depend on LCOM metric which is not well-defined
2	Measures four types of complexity in components	Partially	Characterization of complexity	Not empirically validated Not pure CO	Assumes only white-box reuse
3	Measure of composition of CORBA components	Partially	Both methods suggest that high connectivity increases complexity	CORBA specific while ours is COSEML specific	No empirical validation

Table 6.1 (Continued)

4	Measures complexity of UML components based on interface specifications	Partially	Both characterize complexity using interfaces and inter component links	Does not consider white box reuse. No empirical validation	Not empirically validated
5	Investigates the influence of several metrics on maintenance costs.	No	Maintenance effort is negatively influenced by complexity	This method is for traditional SW development	-
6	A model for detecting the relationship between software architecture and maintainability	Partially	Loosely coupled designs enhance maintainability	Not CO	No clear definition of the system building unit under consideration.
7	Detect the influence of coupling and cohesion on complexity.	No	High connectivity increases complexity	Not CO; Coupling and cohesion must be jointly measured	-
8	A model for system complexity based on system structure and hierarchy.	Partially	Structural complexity is influenced by connections and number of modules in each level of the hierarchy.	Not CO	Not empirically validated
9	Defines a measure of decoupling for service components	Yes	Characterizes the degree of independence	Relationship with process features not discussed	No clear interpretation guidelines.
10	Describes models of structural and functional complexities of modules.	Partially	Relates complexity to integration effort	Assumes white-box reuse only	-
11	Introduces a measurement framework for component based software engineering	Partially	Identifies the steps in measurement in component development	Not pure component oriented. Based on the OO paradigm	Highly dependable on the OO paradigm

As a summary it can be said that there is no comprehensive measurement approach for a comprehensive COSE methodology, to support the build by integration paradigm. Among these approaches only the work of Qiam, Liu, and Tsui [110] can be seen as relevant for a meaningful comparison. The rest of the other works lack the Component Orientation philosophy that was described in the Introduction section of this thesis. Due to that they do not appear to be relevant for a meaningful comparison.

6.4 Future Extensions and Open Research Areas

One of the most important challenges we encountered during this research is the lack of industrial projects to use for metrics validation. The availability of such data is believed to enable further experimental validation. Also, the presence of data collected from operational projects will necessarily help in detecting important relationships between complexity metrics and product quality factors such as: reliability, performance, efficiency, and maintainability. These product features are critical and important to both the developer and the customer. Although student's projects are widely used in Academia as the principal means of practical validation, it is believed that considering industry practices in addition to students' projects will strengthen the validity of the results. Trying to consider industry practices as a validation means will be encountered with serious difficulties due to the following reasons:

1. Lack of standards for the definitions of the term component, component interface, component communication principles. The terms are handled differently by different practitioners.

2. Although component based software development is widely used, still real component orientated software development is not encountered.

3. Even with the availability of industry practices, obtaining the data will be another problem by its own.

According to the previous discussion future extensions to the research presented in this thesis should include some implemented projects data, and industry practices. The relationships between a component's complexity and its composability, reliability, performance, efficiency are still open research areas in this field.

REFERENCES

[1] S. K. Abd-El-Hafiz, "Entropies as Measures of Software Information," *Proc. of ICSM*, IEEE Computer Society, 2001, pp. 110-117.

[2] F. B. Abreu, "The MOOD Metrics Set," *Proc. ECOOP'95 Workshop on Metrics*, 1995.

[3] A. J. Albrecht, "Measuring Application Development Productivity," *Proceedings of the IBM Application Development Symposium*, Monterey, California, October 1979, pp. 83-92.

[4] A.J. Albrecht and J. E. Gaffney, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering*, vol. SE-9, 1983, pp. 639-648

[5] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen, "Measuring Coupling and Cohesion of Software Modules: An Information-Theory Approach," *in Proc. 7th IEEE Symp. Software Metrics, Metrics 2001*, pp. 124-134.

[6] M. Alsharif, P. Bond, and T. Al-Otaiby, "Assessing the complexity of software architecture," *In Proc. Of the 42nd Annual ACM Southeast Conference*, Huntsville, Alabama, 2004, pp. 98-193.

[7] D. Anselmo and H. Ledgard, "Measuring Productivity in the Software Industry," *Communications of the ACM*, vol. 46, No. 11, 2003, pp. 121-125.

[8] V. R. Basili, "Quantitative Software Complexity Models: A panel summary," *Tutorial on Models and Methods for Software Management and Engineering*, IEEE Computer Society Press, 1980.

[9] V. R. Basili and B. Boehm, "COTS-Based Systems Top 10 List," *IEEE Computer*, vol. 34, No. 5, 2001, pp. 91-93.

[10] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22. No. 10, 1996, pp. 751-761.

[11] V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, 1984, pp. 728-738.

[12] J. K. Blundell, M. L. Hines, and J. Stach, "The Measurement of Software Design Quality," *Annals of Software Engineering*, vol. 4, iss. 1, 1997, pp. 235-255.

[13] B. Boehm, "A spiral model of software development and enhancement," *IEEE Computer, Vol* 21, 1988, pp. 61-72.

[14] B. Boehm, "*Software Engineering Economics*," Prentice Hall. 1981, ISBN: 0138221227

[15] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost Models for Future Software Life-cycle Processes: COCOMO 2.0," *Annals of Software Engineering Special Volume on Software Process and Product Measurement*, Vol 1, 1995, pp. 45-60.

[16] G. Booch, "*Object-Oriented Design With Applications*," 2nd ed., Addison-Wesley Professional, 1993, ISBN: 0805353402.

[17] L. C. Briand, S. Morasca, V. R. Basili, "Property-Based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, vol. 22, No. 1, 1996, pp. 68-86.

[18] N. Chapin, "Entropy-Metric for Systems With COTS Software," *Proc. of the 8th IEEE Symposium of Software Metrics (METRICS'02)*, IEEE Computer Society, 2002, pp. 173-181.

[19] J-Y Chen, J-F Lu, "A new Metric for Object-oriented Design," *Butterworth-Heinemann Ltd, Information and Software Technology*, vol. 35, No. 4, 1993 pp. 232-240.

[20] S. R. Chidamber and C. F. Kemerer, "Towards a Metrics Suite for Object-Oriented Design," *Proc. Conf. Object-oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, vol. 26, No. 11, 1991, pp. 197-211.

[21] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, No. 6, 1994, pp. 476-493.

[22] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Transactions on Software Engineering*, vol. 24, No. 8, 1998, pp. 629-639.

[23] B. Clark, "Eight Secrets of Software Measurement," *IEEE Software*, vol. 19, No. 5, 2002, pp. 12-14.

[24] P. C. Clements, "From Subroutines to Subsystems: Component-Based Software Development," American Programmer, vol. 8, no. 11, 1995.

[25] P. C. Clements, L. Bass, R. Kazman, and G. Abowd, "Predicting Software Quality by Architecture-Level Evaluation," *Proceedings of the Fifth International Conference of Software Quality*, Austinx, Tx, October, 1995.

[26] Component Source homepage, URL: **http://www.componentsource.com**, February 2006.

[27] S. D. Conte, H. E. Dunsmorem, and V. Y. Shen, "Software Engineering Metrics and Models," Benjamin-Cummings Pub. Co., Inc, 1986.

[28] P. Coad and E. Yourdon, "*Object-Oriented Analysis*," 2nd Ed., Prentice Hall, 1990, ISBN: 0136299814.

[29] P. Coad and E. Yourdon, "*Object-Oriented Design*," 1st ed., Prentice Hall, 1991, ISBN: 0136300707

[30] T. DeMarco and P. J. Plauger, "*Structured Analysis and System Specification*," Yourdon Press, Prentice Hall, 1979, ISBN: 0138543801.

[31] T. DeMarco, "*Controlling Software Projects: Management, Measurement, and Estimation*," 1st ed., Englewood Cliffs, NJ: Prentice Hall, 1986, ISBN: 0131717111

[32] Dev Direct Website, URL: **http://www.devdirect.com/Content/More.aspx**, February 2006

[33] E. W. Dijkstra, "Structured Programming," Originally in a report on a conference sponsored by the NATO Science Committee, 1969.

[34] A. H. Dogru and M. Tanik, "A Process Model for Component Oriented Software Engineering," IEEE Software, March/April, 2003, pp. 34-41.

[35] K. El Emam, "A Primer on Object-Oriented Measurement," *IEEE Metrics 2001*, London, England, 2001, pp. 185-187.

[36] N. E. Fenton, "Software Measurement: A Necessary Scientific Basis," *IEEE Transactions on Software Engineering*, vol. 20. No. 3, 1994, pp. 199-206.

[37] N. E. Fenton and M. Neil, "Software Metrics: Successes, Failures, and New Directions," *The Journal of Systems and Software*, Elsevier Sciences, vol. 47, 1999, pp. 149-157.

[38] C. Chezzi, M. Jazayeri, and D. Mandrioli, , "Fundamentals of Software Engineering," 2nd ed., Prentice Hall, 2003, ISBN: 013099183-X.

[39] N. Gorla and R. Ramakrishnan, "Effect of Software Structure Attributes on Software Development Productivity," *Journal of Systems and Software*, vol. 36, 1997, pp. 191-199.

[40] R. B. Grady, "*Practical Software Metrics for Project Management and Process Improvement,*" Prentice-Hall, 1992, ISBN: 0137203845.

[41] M. Halstead, "*Elements of Software Science,*" Elsevier Computer Science Library, 1977, ISBN: 0444002057

[42] R. Harrison, S. J. Counsell, and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering*, vol. 24, No. 6, 1998, pp. 491-496.

[43] W. Harrison, "An Entropy-Based Measure of Software Complexity," *IEEE Transactions on Software Engineering*, vol. 18, No. 11, 1992, pp. 1025-1029.

[44] O. Helmer, "*Social Technology*," Basic Books, NY. , 1966, ASIN: B0007FBAWA

[45] L. Hellerman, "A Measure of Computational Work," IEEE Transactions on Computers, vol. c-21, No. 5, 1972, pp. 439-446.

[46] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, 1981, pp. 510-518.

[47] IBM International Technical Support Centers, Object-Oriented Design: A Preliminary Approach-Document GG24-3647-00. IBM International

[48] I. Jacobson, "Object Oriented Software Engineering: A Use Case Driven Approach," 1st ed., Addison Wesley, 1992, ISBN: 0201544350

[49] Sun Developer Network (The Source for Java Developers), URL: **http://java.sun.com**, February 2006.

[50] M. Keating, "Measuring Design Quality by Measuring Design Complexity," *1st International Symposium on Quality of Electronic Design (ISQED 2000)*, IEEE, San Jose, CA, USA, 2000, pp. 103-108.

[51] T. M. Khoshgaftaar and E. B. Allen, "Empirical Assessment of A software Metric: The Information Content of Operators," *Software Quality Journal*, Kluwer Academic Publishers, no. 9, 2001, pp. 99-112.

[52] B. Kitchenham, "An Evaluation of Software Structure Metrics," *Proc. 12th Int'l Computer Software and Applications Conf. (COMPSAC `88)*, IEEE, 1988, pp. 369-376.

[53] B. Kitchenham, S. L. Pfleeger, and N. Fenton, "Towards a Framework for Software Measurement Validation," *IEEE Transactions on Software Engineering*, vol. 21, No. 12, 1995, pp. 929-944.

[54] B. Kitchenham and S. L. Pfleeger, "Software Quality: The Elusive Target," IEEE Software, vol. 13, no. 1, 1996, pp: 12-21

[55] B. Kitchenham, S. L. Pfleeger, and N. Fenton, "Reply to: Comments on "Towards a Framework for Software Measurement Validation"," *IEEE Transactions on Software Engineering*, vol. 23, No. 3, 1997, pp. 189.

[56] B. Kitchenham, S. L Pfleeger, L. M. Pickard, P W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 28, No. 8, 2002, pp. 721-734.

[57] T. J. McCabe, "A complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, 1976, pp. 308-320.

[58] M. G. Mendonça and V. R. Basili, "Validating of an Approach for Improving Existing Measurement Frameworks," *IEEE Transactions on Software Engineering*, vol. 26, No. 6, 2000, pp. 484-499.

[59] COM, COM+m and DCOM Architectures from MSDN Library, URL: **http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/componentdevelopmentank.as**p, February 2006

[60] G. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits to Our Capacity for Processing Information," *The Psychological Review*, vol. *63*, 1956, pp. 81-97.

[61] S. Morasca, L. C. Briand, V. R. Basili, E. J. Weyuker, and M. V. Zelkowitz, "Comments on: Towards a Framework for Software Measurement Validation," *IEEE Transactions on Software Engineering*, vol. 23, No. 3, 1997, pp. 187-188.

[62] CORBA FAQ From Object Management Group (OMG) homepage, URL: **http://www.omg.org/gettingstarted/corbafaq.htm**, February 2006

[63] R. Park, "The Central Equations of the PRICE Software Cost Model," *4th COCOMO Users' Group Meeting*, 1988.

[64] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, 1972, pp. 1053 - 1058.

[65] G. Poels and G. Dedene, "Comments on Property-Based Software Engineering Measurement: Refining the Additivity Properties," *IEEE Transactions on Software Engineering*, vol. 23, No. 3, 1997, pp. 190-195.

[66] C. Pons, L. Olsina, and M. Prieto, "A formal Mechanism for Assessing Polymorphism in Object-Oriented Systems," *First Asia-Pacific Conference on Quality Software,* IEEE, 2000, pp. 53-62.

[67] L. Putnam and W. Myers, *"Measures for Excellence,"* Yourdon Press Computing Series, 1992, ISBN: 0963186809.

[68] R. S. Pressman, *"Software Engineering A Practitioner's Approach,"* 6th ed., McGraw-Hill, 2005, ISBN: 007-123840-9

[69] T. Ravichandran, and M. Rothenberger, "Software Reuse Strategies and Component Markets," *Communications of the ACM*, vol. 46, No. 8, 2003, pp. 109-114.

[70] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *"Object-Oriented Modeling and Design,"* Prentice Hall, 1991.

[71] N. Salman, "Extending Object-oriented Metrics to Components," *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, Pasadena, California, 2002.

[72] N. F. Schneidewind, "Methodology for Validating Software Metrics," *IEEE Transactions on Software Engineering*, vol. 18, No. 5, 1992, pp. 410-422.

[73] N. F. Schneidewind, "Software Metrics for Quality Control," *Proc. Of the 4th international Software Metrics Symposium,* IEEE CS Press, Los Alamitos, CA, 1997, pp. 127-136.

[74] R. W. Sebesta, *"Concepts of Programming Languages,"* 7th Ed., Addison Wesley, 2006, ISBN: 0321312511.

[75] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul, "Software Engineering Metrics for COTS-Based Systems," *IEEE Computer*, vol. 34, No. 6, 2001, pp. 44-50.

[76] R. Seker and M. Tanik, "An Information-Theoritical Framework for Modeling Component-Based Systems," *IEEE Transactions on Systems, Man, and Cybernetics-Part C: Applications and Reviews*, vol. 34, No. 4, 2004, pp. 475-484.

[77] C. E. Shannon and W. Weaver, *"The Mathematical Theory of Communication,"* University of Illinois Press. 1949, ISBN: 0252725484.

[78] S. Shlaer, and S. J. Mellor, *"Object-Oriented Systems Analysis: Modeling the World In Data,"* Yourdon Press: Prentice Hall, 1988, ISBN: 013629023X

[79] S. Shlaer and S. J. Mellor, *"Object-Oriented Systems Analysis: Modeling the World In States,"* Yourdon Press: Prentice Hall, 1991, ISBN: 0136299407.

[80] J. E. Smith, "Characterizing Computer Performance with A Single number," *Comm. of the ACM*, vol. 31, no. 10, 1988, pp. 1202-1206.

[81] I. Sommerville, "*Software Engineering*," Seventh Edition, Addison Wesley, 2004.

[82] R. Subramanyam, M. S. Krishnan, "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering*, vol. 29, No. 4, 2003, pp. 297-310.

[83] C. Szyperski, D. Gruntz, and S. Murer, S., "*Component Software - Beyond Object-Oriented Programming*," 2nd Ed. Addison-Wesley / ACM Press., 2002, ISBN: 0201745720.

[84] J. Tian and M. V. Zelkowitz, "Complexity Measure Evaluation and Selection," *IEEE Transactions on Software Engineering*, vol. 21, No. 8, 1995, pp. 641-650.

[85] P. Vitharana, F. M. Zahedi, and H. Jain, "Design Retrieval and Assembly in Component-Based Software Development," *Communications of the ACM*, vol. 46, No. 11, 2003, pp. 97-102.

[86] G. Visaggio, "Structural Information as a Quality Metric in Software Systems Organization," *Proceeding of ICSM*, 1997, pp. 92-99.

[87] D. A. Watt, W. Findlay, and J. Hughes, "*Programming Language Concepts and Paradigms*," Prentice Hall, 1990, ISBN: 0137288743

[88] E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering*, vol. 14, No. 9, 1988, pp. 1357-1365.

[89] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, "*Designing Object-Oriented Software*," Prentice Hall, 1990.

[90] Wirth, N., 1971, "Program Development by Stepwise refinement," *Communications of the ACM*, Vol. 14, No. 4, pp. 221- 227.

[91] M. V. Zelkowitz and D. Wallace, "Experimental Validation on Software Engineering," *Information and Software Technology*, Elsevier Sciences, vol. 39, 1997, pp. 735–743.

[92] M. V. Zelkowitz and D. Wallace, "Experimental Models for Validating Technology," *IEEE Computer*, Vol.31, No.5, 1998, pp. 23-31.

[93] H. Zuse, "Criteria for Program Comprehension Derived from Software Complexity Metrics," *Proc. of the Second Int. Workshop on Software Comprehension*, IEEE, Capri/Italy, 1993, pp. 8-16.

[94] H. Zuse, "Foundations of Object-Oriented Software Measures," *IEEE Proceedings of METRICS'96*, 1996, pp. 75-88.

110

[95] H. Zuse homepage, URL: **http://irb.cs.tu-berlin.de/~zuse/metrics/3-hist.html**, March 2006

[96] M. A. Jackson, "Principles of Program Design," Academic Press, New York, 1995.

[97] N. Salman, and A. H. Dogru, "Design Effort Estimation Using Complexity Metrics," Transactions of SDPS, 2004.

[98] _____, "IEEE Standard for a Software Quality Metrics Methodology," *IEEE Std 1061*, IEEE, 1998, pp. 1-20.

[99] J. M. Bieman,, L. M. Ott, "Measuring Functional Cohesion," *IEEE Transactions on Software Engineering*, Vol 20, no 8, 1994, pp. 644-657.

[100] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko, "The structural complexity of software: An experimental test," *IEEE Transactions on Software Engineering*, Vol 31, no. 11, 2005, pp. 982-995.

[101] N. S. Gill, and P. S. Grover, "Component-Based Measurement: Few Useful Guidelines," *ACM SIGSOFT Software Engineering Notes*, vol 28, no 6, 2003.

[102] N. S. Gill, and P. S Grover, "Component-Based Measurement: Few Important Considerations for Deriving Interface Complexity Metric for Component-Based Systems," *ACM SIGSOFT Software Engineering Notes*, vol 29, no 2, 2004.

[103] N. Salman and A. H. Dogru, "Design Effort Estimation Using Component Oriented Complexity Metrics," *Proc. of the Work in Progress Session 29th EUROMICRO Conf. EUROMICRO 2003 and the EUROMICRO Symp. On Digital System Design DSD 2003*, Belek, Turkey.

[104] N. Salman, "Metrics and Metrics Validation Approaches for Component Oriented Software Engineering," *Proc. of UYMS'05*, Ankara, Turkey, 2005. (In Turkish).

[105] N. Salman and A. H. Dogru, "Complexity and Development Effort Prediction Models Using Component Oriented Complexity Metrics," *Proc. of Int. Conf. on Software Product and Process Measurement (MESURA2006)*, Cadiz, Spain, Nov. 2006. (Accepted for presentation and publication).

[106] M. Goulao, F. B. e Abreu, "Composition Assessment Metrics for CBSE," *Proc. Of the 2005 31st EUROMICRO Conf. of Software Eng. And Adv. Applications. EUROMICRO-SEAA'05*. 2005

[107] S. Mahmood, R. Lai, "Measuring Complexity of a UML Component Specification," *Proc. Of the Fifth Int. Conf. on Quality Software (QSIC'05)*. 2005.

[108] R. D. Banker, S. M. Datar. C. F. Kemerer, and D. Zweig, " Software Complexity and Maintenance Costs," *Comm. Of the ACM*, Vol. 36. No. 11. 1993.

[109] M. Lindvall, R. T. Tvedt, and P. Costa, "An Empirically-Based process for Software Architecture Evaluation," *Empirical Software Engineering*, 8, 2003.

[110] K. Qian, J. Liu, and F. Tsui, " Decoupling Metrics for Services Composition," *Proc. Of the 5th IEEE/ACIS Int. Conf. on Computer and Inf. Sci. and 1st IEEE/ACIS Int. Workshop on Component-Based Software Engineering, Software Arch. And Reuse (ICIS-COMSAR'06)*. 2006.

[111] D. Braha and O. Maimon, "The Measurement of a Design Structural and Functional Complexity," *IEEE Trans. On Systems, Man and Cybernetisc-Part A: Systems and Humans*, vol. 28, no. 4, 1998.

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

[113] H. A. Simon, "Sciences of the Artificial," MIT Press, Cambridge, Massachusetts, 1969.

[114] R. R. Dumke and A. S. Winkler, "Managing the Component-Based Engineering with Metrics," Proc. of the 5th International Symposium on Assessment of Software Tools (SAST'97), 1997.

APPENDIX A

METRICS COLLECTION FORMS USED IN 2002 AND 2003

Number of people in the team:	
Total person-hours:	
Member 1 name:	Person hours:
Member 2 name:	Person-hours:
Total person-hours spent for modification:	
person-hours for an **average** maintenance (correction):	

Complexity of the Model:	
Number of boxes (total- abstractions, components, interfaces..):	
Number of Components:	
Number of Interfaces:	
Number of Connectors:	
Number of event links:	
Number of method links:	
Number of methods:	
Average number of methods per component:	
Average number of input events per component:	
Average number of interfaces per component:	
Average number of methods per interface:	
Maximum depth of the composition tree:	
Maximum width of the composition tree:	
Average NOC (Number of Children in the composition tree):	
Average DCT *(Depth of composition tree):	
Average CBC* (coupling = cardinality of methods called from outside):	
Average RFC (Response for a Component):	
Average Mean LCOM* (mean values averaged for Lack of Cohesion in Methods):	

Please give a grade (5: strongly agree; 1: strongly disagree)	
It was easy to model your problem using COSEML:	
Your model is an understandable representation of the problem:	

Sample Metrics of a project Components

Component Name	# of Methods	# of eventsIn	# of interfaces	NOC	DCT	CBC	RFC	LCOM
WebSite	0	0	0	4	1	0	0	0
Accounting	0	0	0	1	1	0	0	0
Cargo	1	0	0	1	1	1	1	1
Inventory	1	0	0	0	1	0	1	1
Delivery	0	0	2	0	2	1	2	1
Register	0	0	2	0	2	0	0	0
Login	0	0	2	0	2	0	0	0
Search	0	0	3	1	2	0	0	0
Buy	0	0	2	0	2	0	0	0
Pay	0	0	3	0	2	0	0	0
Catalog	0	0	3	0	3	0	0	0
ShopCard	3	2	0	0	3	0	3	-3
CreditCard	0	2	0	0	4	0	0	0
Order	0	2	0	0	4	0	0	0
Product	0	2	0	0	4	0	0	0
CustDB	3	2	0	0	4	3	3	-3
ProdDb	3	2	0	0	4	3	3	-3
InventoryDB	3	2	0	0	4	3	3	-3
RegisterUI	2	2	0	0	4	3	5	2
LogUI	2	2	0	0	4	2	2	0
SearchUI	2	2	0	0	4	4	7	3
BuyUI	2	2	0	0	4	5	6	2
PAyUI	2	2	0	0	4	3	4	1

APPENDIX B

METRICS COLLECTION FORMS USED IN 2005

Project title and brief description:	
Total person-hours	
Total person-hours spent for correcting design errors	
Total person-hours spent for Integrating components	
Complexity of the Model	
Total Function Points (FP)	
Total number of boxes (total- abstractions, components, interfaces..)	
Total number of Components	
Total number of Interfaces	
Total number of Connectors	
Total number of inter-component method links	
Total number of methods	
Depth of the structure tree	

Component name and a brief description:						
Total Person-hours (For designing this component and all of its related elements only)						
Total person-hours spent for correcting design errors (for this component)						
Total person-hours spent for Integrating the component to the system (for this component)						
Number of sub-components						
Number of methods						
Number of events						
Number of interfaces						
Number of methods per interface						
Number of methods called from outside						
Number of Inter-Component method calls						
Number of Intra-Component method calls						

VITA

Nael Salman was born in Nablus-Palestine on May 14, 1970. He received the B.Sc. degree in computer engineering from Bogazici University (Istanbul-Turkey) in July, 1995. He worked as an instructor in the department of Computer Software and Databases in Palestine Technical College (Toulkarem-Palestine) between 1996 and 1999. In October, 2000 Mr. Salman started working as an instructor in the Department of Computer Engineering at Cankaya University (Ankara-Turkey). Mr. Salman received his M.Sc. degree in computer engineering from Middle East Technical University (METU) in January, 2002. His research interests involve software engineering, software measurement and metrics, component oriented software development, object oriented software development, and database systems.