Design and Analysis of Metrics for Component-Based Software Systems

A Thesis

Submitted in fulfillment of the
requirements for the award of the degree of

Doctor of Philosophy

Submitted by

Arun Sharma
(Registration No. 9031451)

Under the supervision of

Dr. P. S. Grover
Professor of Computer Science,
Guru Tegh Bahadur Institute of Technology,
Guru Gobind Singh Indraprastha University,
Delhi – 110006.

Dr. Rajesh Kumar
Assistant Professor,
School of Mathematics and Computer Applications, Thapar University,
Patiala –147004.

School of Mathematics and Computer Applications
Thapar University,
Patiala–147 004 (Punjab), India.

March 2009

To

My Family

CERTIFICATE

I hereby certify that the work which is being presented in this thesis entitled **DESIGN AND ANALYSIS OF METRICS FOR COMPONENT BASED SOFTWARE SYSTEMS**, in fulfillment of the requirements for the award of degree of **DOCTOR OF PHILOSOPHY** submitted in School of Mathematics and Computer Applications (SMCA), Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Dr. P S Grover and Dr. Rajesh Kumar, and refers the work of other researchers, which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(Arun Sharma)
Registration No. 9031451

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge and belief.

(P S Grover)
Professor of Computer Science,
Guru Tegh Bahadur Institute of Technology,
Guru Gobind Singh Indraprastha University, Delhi–110006 (INDIA)

Supervisor

(Rajesh Kumar)
Assistant Professor,
School of Mathematics and Computer Applications,
Thapar University, Patiala–147 004 (INDIA)

Supervisor

ABSTRACT

Component-based Software Engineering (CBSE) is a paradigm that aims at constructing and designing systems by using a pre-defined set of software components. A component is a reusable, self-contained piece of software with well-specified interface that is independent of any application. Main advantages of this approach are low cost, high quality and less time to develop applications along with several others . There are many software component models available in the industry, for example: *Microsoft's* COM (Component Object Model), DCOM, .NET Framework, *Sun's* Java Beans, EJB (Enterprise Java Beans), J2EE Specification and *OMG's* (Object Management Group) CORBA (Common Object Request Broker Architecture).

However, besides several advantages, CBSE still faces many problems. The research aspect of component quality assessment, with emphasis on quantitative approaches, is fairly unexplored. The few existing evaluations are performed at a qualitative level. In traditional approach, most of the metrics models are based on source code analysis, which can not be used in component-based systems. Due to the black-box nature of the components, it requires considerably different approach for quantitative analysis of quality aspects.

OBJECTIVES OF THE PROPOSED WORK

In this work, the main emphasis is on quantitative evaluation of metrics for various functional and non-functional aspects of software components and their impact on the system, they are integrated in. Also, it proposes a quality model for estimation of quality for software components. Following are the objectives which are explored in this study:

Objective 1: *To identify the metrics for software components.*

Metrics have been thoroughly studied and explored for several quality features of component-based systems. These features include complexity, customizability,

reusability, maintainability and others. The new metrics have been proposed for interface complexity, reusability and maintainability aspects.

Objective 2: *To design a methodology to evaluate the metrics.*

We have proposed quantitative evaluation of interface complexity and reusability for black-box components and maintainability for component-based systems. Interface complexity metric is based on the interface methods and the properties of the component and is empirically evaluated on several Java Bean components. To estimate the reusability, Artificial Neural Network (ANN) technique has been adopted, while for maintainability of system, we used Fuzzy Logic based approach.

Objective 3: *To explore and compare these metrics along with some other existing metrics.*

Interface complexity metric is evaluated on several Java Bean components and validated against customizability, readability and performance metrics. A correlation analysis has also been conducted to establish the relationships between the suggested metric and these existing metrics. Reusability metric is validated by considering test data collected from real-life components. Maintainability prediction is validated by using Analytical Hierarchy Process (AHP) on two class room based projects.

Objective 4: *To propose a model to bring out the relationship among metrics and with a view to establish quantitative estimation of quality and validate the suggested model.*

A quality model for component-based systems is proposed which includes new quality characteristics like complexity, reusability, flexibility and trackability along with others, present in most of the quality models, including ISO 9126. Weight values are assigned to these characteristics and sub-characteristics by using Analytical Hierarchical Process (AHP) and by conducting a survey on experienced software professionals, working on component-based technologies. This model may be used to evaluate the quality of the component as a single variable. A case study from live project has also been considered for empirically evaluating the quality of software components by the proposed model.

THESIS OUTLINE

Thesis is divided into seven chapters as follows:

Chapter 1 covers the basic issues and understanding about the components and component-based systems (CBS). It consists of a detailed literature survey of various metrics for CBS. It covers complexity, reusability, customizability, maintainability, and other aspects of these systems. Some of the metrics proposed in the literature are derived from that of object-oriented systems with no or minor changes, others are exclusively specific for CBS. This chapter also covers various quality models, and discusses the major findings of these models in the context of CBS.

Chapter 2 proposes an interface complexity metric for software components. While proposing this metric, it considers that due to black-box in nature, source code of the components is not available to the application developers. Only interface methods and properties are available. Based on the complexities of these two, the interface complexity of the component is designed. The proposed metric is then evaluated theoretically against standard Weyuker's properties (Weyuker, 1988) and most of these properties are satisfied by the proposed metric. This metric is evaluated for several Java Bean components taken from different sources. The proposed metric is validated against performance, customizability and readability metrics (Washizaki *et al.,* 2003). Correlation study is performed between complexity metric and these metrics. The results show that there is strong positive correlation between complexity and execution time i.e. complex components take much time to execute. Also negative correlation coefficients between complexity and customizability and complexity and readability confirm that complex components are hard to understand and maintain. The proposed metric may be used by the developers to evaluate the complexities of identical target components and then select the least complex component among them for use. It will help in developing less complex systems which will be easy to understand and will have low maintenance cost.

Chapter 3 discusses the various methodologies (Vieira and Richardson , 2002; Li, 2003; Stafford *et al.,* 2003; Guo, 2002) to represent component dependency for a CBS which include adjacency matrix and graph based approaches also. The main problems of these approaches are that all of them store only the presence of dependency and do not store the interactions and their types. We propose a link-list based representation for

storing component dependencies. Proposed approach can store interactions and their type, which in turn can be used to evaluate several metrics like interaction density, dependency level, list of child components, most critical and isolated components etc. These metrics may be used to measure the interaction complexity of the system and are useful in understanding the system during the testing and maintenance. We conducted an experiment by taking a case study of component-based system and represented dependency by using the proposed methodology. We also evaluated several interaction metrics by this proposed representation.

Chapter 4 discusses several existing measures for reusability (Mili *et al.*, 1995; Dumke and Schmietendorf, 2000; Boxal and Araban, 2004; Rotaru *et al.*, 2005) and proposes an Artificial Neural Network based approach to predict it for software components. We identified the factors influencing reusability which include customizability, interface complexity, portability and understandability. These factors are categorized from very low to very high categories. Neural network is trained by considering a real-life component data, collected from various sources. The best training is obtained by taking different training functions, numbers of hidden layers and neurons. The network is then tested by taking another set of test data. Results indicate that proposed model is suitable for predicting reusability of software components.

Chapter 5 is about the maintenance of CBS. Maintenance of these systems may require several different activities than normal applications, such as, upgrading the functionality of black-box components (for which code may not be available), replacement of older version components with the new ones for better and improved functionality, tracing the problem of compatibility between the new components with system, and so on (Kajko *et al*., 2006). The focus of this chapter is on investigating several issues and concerns about maintainability of component-based systems. It also explores the acceptance of maintainability characteristic and its sub-characteristics as defined in ISO 9126 quality model for CBS. It proposes a new sub-characteristic 'trackability' to be added under maintainability. Trackability will keep track of various system properties during the maintenance activities. These activities may include tracking of system performance or resource utilization, before and after any maintenance activity. This may also include the possible security violations, like access authorization due to

some maintenance activities. Tracking, not only will validate any improvement effort, but also, it may provide insight into managing statistical control. This, in effect, will ease the overall maintenance process. The present work also proposes Fuzzy Logic-based approach to estimate the maintainability of component-based systems. We identified the factors influencing maintainability of the CBS and then categories them into low, medium and high. Rules were designed, based on the expert opinion. Finally, by defuzzification process, the maintainability is estimated. The proposed model is implemented on two classroom based projects to empirically evaluate and validate the maintainability.

Chapter 6 studies several existing quality models (McCall and Joseph, 1978; Boehm *et al.*, 1976; Grady, 1992; Dromey, 1995; ISO, 2001; Bertoa and Vallecillo, 2002; Rawashdeh and Matalkah, 2006) and then proposes a new quality model for CBS by adding several new sub-characteristics which are relevant to the CBS. These sub-characteristics include reusability, complexity, flexibility, trackability and scalability. It also proposes to remove analyzability and attractiveness sub-characteristics from standard ISO 9126 quality model, while keeping others in the proposed model as it is. This proposed quality model is then refined by conducting a survey on software professionals working on live projects on component technologies and collected their preferences on these quality characteristics and sub-characteristics on a scale of 1-4 (1-never used, 4 – always used). This data is analyzed by using Analytical Hierarchy Process (AHP) and weight values are calculated to quality characteristics and sub-characteristics, for the proposed model. The weight values obtained will help developers to select only those quality characteristics and sub-characteristics, which are important and relevant as per their quality requirement in that domain.

The proposed model is implemented on a live case study. Only those characteristics and sub-characteristics are selected, which are more important in that domain. Weight values are then normalized and metrics are identified for the selected characteristics and sub-characteristics. Some of the metrics need to be normalized to fit into same unit. The metric values are multiplied by their corresponding weight values to get the value of quality sub-characteristics, characteristics and then finally of the whole system. This experimentation leads to the useful conclusion related to the capability of

measuring quality of the component-based system as a single variable, which may be used to compare and select the better quality components for the end product.

Thesis summarizes in Chapter 7 with the major contributions of the present work. These contributions include inferences drawn as a result of various experiments conducted in this thesis. It also suggests some future work in this direction.

ACKNOWLEDGEMENT

The work leading to this doctoral thesis has been carried out by me as a research scholar at the School of Mathematics and Computer Applications (SMCA) of Thapar University, Patiala. These four years have been very stimulating and instructive for me.

At the outset, I am highly indebted to my revered supervisors, Dr. Rajesh Kumar and Professor P. S. Grover for their untiring support, encouragement and able guidance at each and every step throughout this research endeavor. I admire the knowledge of Professor Grover. It is really fortunate that I got such an opportunity to learn the ABC of research from him, which will definitely go a long way in my professional career. Further, I find no words to express my heartfelt thanks to Dr. Rajesh for supervising this work so diligently and delightfully. It has indeed been a privilege to carry out this research work under their guidance.

I am grateful to the Director, Thapar University, Patiala for providing me University's resources and facilities necessary to carry out this research work. I express my gratitude to the Doctoral Committee comprising Professor (Dr.) S. S. Bhatia, Professor (Dr.) R. K. Sharma and Professor (Dr.) Seema Bawa for monitoring the progress and providing valuable suggestions for improvement of my Ph. D. research work from time to time. I am also thankful to the entire faculty and staff at SMCA for healthy discussions, suggestions and necessary cooperation to complete this work.

I wish to thank two personalities who greatly influenced my life and career, Dr. Ashok K. Chauhan, Founder President, RBEF and Maj. Gen. K. Jai Singh (Retd.), Vice Chancellor, Amity University, Uttar Pradesh. I am thankful to my Director at AIIT for allowing me to use the Institute's resources, required for this research and investigations.

Special thanks are due to my friend Mr. Sandeep Chauhan, Project Lead, Steria (I) Pvt. Ltd. (Formerly Xansa India) for his valuable suggestions and providing the technical support to evaluate and validate some of the metrics. I am also thankful to my colleagues Mr. Avadhesh Kumar and Mr. O. P. Sangwan for their active suggestions and advise on several occasions.

My hearty thanks to all the anonymous referees of several international r
journals in the domain of software engineering, quality, neural and fuzzy comput
their constructive comments and new ideas for improvement of the research work.

I wish to thank many acclaimed researchers across the world for v
exchange of ideas and making available the reprints of their publications. These
Dr. Will Tracz, Dr. S. K. Chang, Prof. H. Washizaki, Dr. Ricard Land, Dr. O. P.
Dr. N. S. Gill and several others.

Finally I would like to dedicate this thesis to my family. I thank my paren
gave me the courage to get my education, supported me in all achievements thr
my life. I am grateful to my other family members for their continuous encoura
and motivation. I must acknowledge the constant cooperation, hard work an
support of my loving wife, Mrs. Geeta along with our children Sparsh and
throughout this project, without which it could have not been possible to achi
target so contentedly.

Above all, I pay my reverence to the almighty God.

(Arun S

Place: Patiala
Date: 09/03/2009

LIST OF PUBLICATIONS BY THE AUTHOR

Papers in International Journals (Published/Accepted/Communicated)

1. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Dependency Analysis for Component-Based Software Systems, accepted for publication in **ACM SIGSOFT Software Engineering Notes**, Vol. 34, Issue 4, July 2009, pp: 1-8.
2. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Reusability Assessment for Software Components, accepted for publication in **ACM SIGSOFT Software Engineering Notes**, Vol. 34, Issue 2, March 2009, pp: 1-6
3. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Evaluation of Complexity for Software Components, **International Journal of Software Engineering and Knowledge Engineering (IJSEKE),** Vol. 19, Issue 5, November 2008, pp: 919-931.
4. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Estimation of Quality for Software Components – an Empirical Approach, **ACM SIGSOFT Software Engineering Notes**, Vol. 33, Issue 6, November 2008, pp: 1-10.
5. P. S. Grover, Rajesh Kumar, **Arun Sharma**, Few Useful Considerations for Maintaining Software Components and Component-Based Systems, **ACM SIGSOFT Software Engineering Notes**, Vol. 32, Issue 5, September 2007, pp: 1-5.
6. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Complexity Measures for Software Components, **WSEAS Transactions on Computers**, Vol. 17, Issue 7, July 2007, pp: 1005-1012.
7. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Managing Component-Based Systems with Reusable Components, **International Journal of Computer Science and Security (IJCSS)**, Vol. 1, Issue 2, March 2007, pp: 60-65.

8. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Predicting Maintainability of Component-based Systems by using Fuzzy-Logic, **Communications in Computer and Information Science**, **Springer Berlin Heidelberg**, **USA,** Vol. 40, Issue 11, pp: 581-593.

Papers in Conference Proceedings

1. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Empirical Evaluation and Critical Review of Complexity Metrics for Software Components, published in the proceedings of the **6th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems**, Corfu Island, Greece, February 16-19, 2007, pp: 24-29.

2. **Arun Sharma**, Rajesh Kumar, P. S. Grover, A Critical Survey of Reusability Aspects for Component-Based Systems, presented and published in the proceedings of **International Conference on Computers, Information Sciences and Engineering (CISE'07)** at Bangkok, Thailand, January 29-31, 2007, pp: 411-415.

3. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Trackability Aspects for Component-Based Systems**,** in **INDIACom-2008: 2nd National Conference: Computing for Nation Development**, Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi, February 15-16, 2008, pp: 734-735.

4. **Arun Sharma**, Rajesh Kumar, P. S. Grover, Classification of Metrics for Component-Based Systems, in proceedings of **National Conference on Applied Computing and Mathematics (FACM'05)** at Thapar University, Patiala, March 04-05, 2005, pp: 267-174.

ABBREVIATIONS

ACC	Average Cyclomatic Complexity
AHP	Analytical Hierarchy Process
AID	Average Interaction Density
AII	Available Incoming Interactions
ANN	Artificial Neural Network
AOI	Available Outgoing Interactions
BPNN	Back Propagation Neural Network
CAID	Component Average Interaction Density
CBD	Component-Based Development
CBO	Coupling Between Objects
CBS	Component-Based System
CBSE	Component-Based Software Engineering
CICM	Component Interface Complexity Metric
ComPARE	Component-Based Program Analysis and Reliability Evaluation
CORBA	Component Object Request Broker Architecture
COTS	Commercial of the Shelf
CPD	Component Packing Density
CR	Component Reusability
CSI	Changed Source Instructions
DIT	Depth of Inheritance Tree

DL	Dependency Level
EC	Enterprise Component
EMI	Existence of Meta Information
ET	Execution Time
FL	Fuzzy Logic
FURPS	Functionality Usability Reusability Portability Stability
IID	Incoming Interaction Density
ISO	International Organization for Standardization
LCOM	Lack of Cohesion in Methods
MRE	Mean Relative Error
MTTR	Mean Time To Repair
NOC	Number of Children
NPV	Net Present Value
OID	Outgoing Interaction Density
OOP	Object Oriented Program
RCC	Rate of Component Customizability
RCO	Rate of Component Observability
REBOOT	Reuse Based on Object-Oriented Techniques
RFC	Response For a Class
RMSE	Root Mean Square Error
RSC	Readability of Source Code
RSI	Reused Source Instructions
SCC	Self Completeness of Component

SMPEEM	Software Maintenance Project Effort Estimation Model
SSI	Shipped Source Instructions
TMF	Triangular Membership Function
UII	Used Incoming Instructions
UML	Unified Modeling Language
UOS	Understanding of Source Code
WCC	Weighted Class Complexity

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

COMPONENT-BASED SOFTWARE SYSTEMS – AN INTRODUCTION AND REVIEW OF LITERATURE

1.1 INTRODUCTION

Improving business performance often needs the improvement in their software development performance and that is the reason, which enforces the dev elopers and researchers to think towards the adoption of latest technologies and development approaches. Earlier, systems were developed by using structured approach, which was very successful, but only for simple applications. Then ca me the object-oriented (OO) approach, which is based upon encapsulation, inheritance, and polymorphism. Encapsulation packages the data and operations that manipulate the data into a single named object. Inheritance enables the attributes and operations of a class to be inheri ted by all subclasses and the objects that are instantiated from them. Polymorphism enables a number of different operations to have the same name, reducing the number of lines of code required to implement a system. It is reported that during the first ha lf of the 1990s, object-oriented software engineering became the paradigm of choice for many software product builders and a growing number of information systems and engineering professionals.

But, besides several advantages of OO approach, the fundament al activity of programmers is still unchanged: writing code line by line. Furthermore, object orientation is difficult to learn and apply for complex applications. Also, OO languages support information hiding at class l evel, but not beyond that. Integrity and confidentiality are other critical issues in OO approach.

1.2 COMPONENT-BASED DEVELOPMENT

Many information-based legacy systems contain similar or even identical things, which are developed from scratch again and again. From the scratch, development is more expensive and can take a long time to complete. Critical applications with strict time limits may loose the market due to the delay in the development process. This has led to the evolution of a new approach, called component-based development (CBD), which uses the concept of reusability in the application development. The high productivity is achieved by using standard components. CBD can be best described by the following two guiding principles:

- Reuse but do not reinvent;
- Assemble pre-built components rather than coding line by line.

Here components are viewed as families of routines that are constructed on the basis of well-defined principles so that these families fit together as building blocks. In the last few years, CBD approach has become very popular. Component-based systems achieve flexibility by clearly separating the stable parts of the system (i.e. the components) from the specification of their composition.

1.2.1 Component

Component-based software development (CBSD) is an approach in which systems are built from well-defined, independently produced pieces, known as components. Some definitions emphasize that components are conceptually coherent packages of useful behavior, while some others state that components are physical, deployable units of software which are executed within a well defined environment. Researchers have proposed several definitions for component . Some of these are as:

- A component is a language neutral, independently implemented package of software services, delivered in an encapsulated and replaceable container, accessed via one or more published interface. While a component may have the ability to modify a database, it should not be expected to maintain state information. A component is not platform-constrained nor is it application-bound (Sparling, 2000).

- A software component is a unit of composition with contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subjected to composition by third parts (Szyperski, 1998).
- A software component is a unit of packaging, distribution or delivery that provides services within a data integrity or encapsulation boundary (Microsoft Corp).
- A software component is a coherent package of software implementation that can be independently developed and delivered. It has explicit and well-specified interfaces for the services it provides and for the services it expects from the others. Also, it can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves (D' Souza, 1999).

In summary, a component is a reusable, self-contained piece of software with well-specified interface that is independent of any application. The very important point which has to be kept in mind while developing a component is the reusability aspect, regardless of whether or not an organization can identify what the future requirements of the component will be. Components can be placed on any network node, depending on application needs and regardless on the type of particular network structure. An extra effort must be paid for the additional functionality of the component beyond the current application's need, to make the component more useful (Gill and Grover, 2003).

Components are black-box entities that encapsulate services behind well-defined interfaces. These interfaces tend to be very restricted in nature, reflecting a particular model of plug-compatibility, supported by a component-framework, rather than being very rich and reflecting real-world entities of the application domain. Components are not used in isolation; they may also have the rules governing their composition.

1.2.2 Component-based Software Engineering

In Object-Oriented Programming (OOP), code is reused in the form of objects, and several mechanisms such as inheritance and polymorphism let the developer s reuse these objects in several ways. The principle is the same with component-based software

engineering (CBSE) also, but here the focus is on reusing whole software component, not just the objects.

CBSE is a paradigm that aims at constructing and designing systems using a pre-defined set of software components explicitly created for reuse. It shifts the emphasis from programming to composing software systems. According to Clements (1996), CBSE embodies "the 'buy, don't build' philosophy". Developing software systems from existing components offer many advantages:

- Development cost is reduced because systems are developed from existing pre-built components
- Reliability is increased, since the components have previously been tested in various contexts
- Less time to market, since the components have already been created
- Low maintenance costs. Since the components are designed to be used in different contexts; the maintenance of each component will be beneficial to multiple systems.

There are many software component models available in the industry, some of these are *Microsoft's* COM (Component Object Model), DCOM, .NET Framework, *Sun's* Java Beans, EJB (Enterprise Java Beans), J2EE specification and *OMG's* (Object Management Group) CORBA (Common Object Request Broker Architecture) specification, and others.

For CBSD, many of the methods, tools and principles of software engineering can be used in the same or in a similar way as in other types of applications or systems. But, there is one distinction; CBS covers both component development and system development with components. There is also a slight difference in the requirements and development approach. Components are built to be used and reused in many applications, some not yet existing. A component must be well specified, easy to understand, sufficiently general, easy to adapt, easy to deliver and deploy and easy to replace. The component interface must be as simple as possible and strictly separated (both physically and logically) from its implementation. Marketing factors play an important role as development costs must be recovered from future earnings. Development with

components is focused on the identification of reusable entities and relations between them, starting from the system requirements.

Component-based development (CBD) is expected to have bright future and a tremendous scope for research. Present work in the thesis studies and explores existing metrics for several quality features for components and component-based systems and proposes new metrics for complexity, reusability, and maintainability aspects of quality. It uses Artificial Neural Network, Fuzzy Logic and Analytical Hierarchical Process to design, evaluate and validate these metrics. Further, it proposes a quality model for CBS and evaluates it on a case study from live project.

1.3 SOFTWARE METRICS

As the number of components available on the market increases, it is becoming more important to devise software metrics to quantify the various characteristics of components and their usage. Software metrics are intended to measure the software quality and performance characteristics quantitatively, encountered during the planning and execution of software development. These can serve as measures of software products for the purpose of comparison, cost estimation, fault prediction and forecasting. Metrics can also be used in guiding decisions throughout the life cycle, determining whether software quality improvement initiatives are financially worth while (Sedigh *et al.*, 2001).

A lot of research has been conducted on software metrics and their applications. Most of the metrics proposed in literature are based on the source code of the application. However, these metrics cannot be applied on components and component-based systems as the source code of the components is not available to application developers. Therefore, a different set of metrics is required to measure various aspects for component-based systems and their quality issues. The following section discusses several metrics to measure various aspects, including complexity, reusability, maintainability and others for legacy as well as for component-based systems.

1.3.1 Complexity Measurement

Complexity is a major driver of the cost, reliability, and functionality of software systems. In conventional software development, complexity can be defined as the

difficulty to analyze source code, modify and maintain its modules. Several metrics ha ve been proposed for measuring various aspects of complexity , such as size, control flow, data structures, and inter-module structure. Some of these metrics proposed for legacy and component-based systems are as:

1.3.1.1 Complexity Metrics for Legacy Systems

McCabe (1976) proposed a complexity metric, called Cyclomatic Complexity (CC), which is based on the number of nodes and connected components in a program graph. Kafura and Henry (1981) also proposed the complexity metrics based on the number of local information-flows entering (fan-in) and exiting (fan-out) in each procedure. Halstead complexity measurement (Halstead, 1977) was developed to measure a program module's complexity directly from source code, with emphasis on computational complexity. These measures are based on four scalar numbers (distinct and total number of operators and operands) derived directly from a program's source code. Weyuker (1988) proposed a set of properties for syntactic software complexity evaluation. Author evaluated some s tandard complexity measures against these properties. These properties help researchers to clarify the strength s and weaknesses of their proposed complexity measures, which ultimately lead to the definition of better proposals for complexity measurement.

1.3.1.2 Complexity Metrics for Component-based Systems

In CBS, development is restricted up to the component customization and integration. Rather than changes in the source code, its maintenance involves only replacing, adding and deleting components and then finally integrating the affected component into the system. The only information available to its users about the component is its interfaces. These interfaces can be used to measure the complexity of the components, which are finally to be integrated in the system. This will be helpful during analyzing, testing and maintenance of the system. This may also be used as a predictor of the efforts needed for maintaining the system.

Tullio Vernazza *et al*. (2000) extended the CK metrics (Chidamber and Kemerer, 1994). Authors proposed new metrics corresponding to each CK metric. Like, for number of methods (NOM), they proposed weighted class per component, for number of children (NOC), the proposed metric was number of children for a component and so on. The

proposed metrics are also validated against the theoretical properties pro posed by Briand *et al*. (1999). However, there is no empirical validation conducted for these metrics against any industry project, thus leaving the work incomplete.

Cho *et al.* (2001) proposed a suite of complexity metrics for software components. The approach considers the classes and their methods of each component and captures the dynamic complexity, based on analysis of source code of the component. However, it cannot be used to measure the complexity for black-box components, as the source code of these components is not available.

Pernilla (2002) suggested several factors that contribute to the complexity of large component-based software projects. These include: number of entities and relationships, number of types and software models, diversity of software representations and others. Component brokering, configuration management, testing are other issues which contributes towards the complexity of the system. However, the proposed work does not describe any methodology to measure these factors.

Gill and Grover (2003) suggested several metrics applicable to component-based development. These metrics include component interface complexity metric, component size metric, component portability metric, component integration complexity metric, component functionality metric and several others. Proposed work includes several metrics covering functional and non-functional behavior of components and CBS. However, the work does not give any methodology to measure and validate these metrics for components or component-based systems.

Bertoa *et al.* (2006) proposed usability metrics for software components. Usability is a quality criterion in ISO 9126 quality model, which covers five sub-characteristics, namely, understandability, learnability, operability, attractiveness and compliance. Out of these, only first three are considered relevant for software component. Several measurable concepts and attributes related with these three aspects of quality are explored. However, most of the work proposed here is subjective and very difficult to measure on a real time application.

Nael (2006) proposed several metrics for components, connectors between components, interface of each component and composition t ree. However, the proposed metrics are very basic in nature and are based on just the total numbers and does not

consider the complexity of individual interface. Gill and Grover (2004) proposed interface complexity metric, based on interface signatures, co nstraints on the interfaces and the packaging for different context of use. For each of these aspects, a definition is also proposed. However, work still lacks of any empirical evaluation and validation of the proposed metric.

Sharma *et al.* (2008a) proposed interface complexity metric for software components by considering interface methods and their associated properties, argument types and return types. Authors evaluated this metric on several Java Beans components and finally validated it against executi on time, readability and customizability. Results concluded that complex components take much time to execute and these are very difficult to maintain. The details of the work are explained in Chapter 2 of the thesis.

1.3.2 Component Dependency

Component-based systems are developed by integrating a number of components in the system. Due to the integration of these components , more and more interconnections exist in the system. Interactions among c omponents result in dependency, which leads to the complex system and results in poor understanding and high cost of maintenance.

As the systems are becoming larger and complex, it is very difficult to understand and track dependency relationship. Understanding dependency in a system helps inconsistencies to be treated and resolved during the syst em composition and evolution. The benefits of dependency analysis are clear in providing developer s/maintainers with significant knowledge and understanding of the system. Dependency analysis helps us to identify the critical components, isolated components, impact of change in one component on other components. To understand and model the dependency relationships, researchers have proposed different methodologies and frameworks. Some of these representations are discussed in next section.

1.3.2.1 Component Dependency Representation Techniques

Graph based approach has been widely used by researchers for representing dependency. Lisa and Delugach (2001) proposed the dependency representation in terms of conceptual graphs. Con ceptual graphs are formal, logic based, and semantic network

language and are used in domain modeling and requirement modeling. A CG is made up of concepts, relations and a possible value. Relations are connected to concepts with directed arrows. Hierarchies represent the subtype/supertype. Authors also implemented the proposed methodology by taking three examples and compared it against other existing methodologies, including UML. Yi and Nahrstedt (2001) categorized dependency into functional dependency and resource dependency and used directed graph based approach to represent these dependencies.

Guo (2002) suggested a category theory based framework for modeling component dependencies. Category theory is a branch of Mathematics, designed to describe various structural concepts from different mathematical fields in a uniform way. The work defined several definitions and represented component dependencies by using these definitions. The proposed model and definitions are very much similar to object-oriented data model to represent various elements like subtypes, attribute inheritance etc. However, the work does not implement the proposed framework empirically.

Vieira and Richardson (2002) represented the dependency relationships by using pomsets description. Pomsets define a set of labeled events in a sequence and is able to express what can take place after a particular component access point is called by another component. In other words, it is a sequence of one or more actions in the form of concurrent regular expressions. Pomsets are considered to be compact and low computational overhead and therefore can be used to describe the component dependency effectively.

Stafford *et al.* (2003) demonstrated a graph based representation for the dependency relationship between two or more components. This directed graph is further used to form an adjacency matrix $AM_{[i,j]}$. If there is a dependency between two components C_i and C_j, then $AM_{[i,j]}$ is 1, else it is 0. This representation is used to compute the total dependencies of a component and of the system. However, it stores only the presence of the dependency, and not the type of the dependency or the event/interface through which these components are dependent.

Li (2003) also described the dependency in terms of adjacency matrix and component dependency graph. He categorized dependency into eight categories , namely, data, control, interface, time, state, cause and effect, input/output and context

dependencies. Author considered these eight dependencies to measure the final dependency by using Boolean operators. By using this approach, several dependency relationships can be deduced. Like, if adjacency matrix obtained by all these dependencies is upper triangle matrix, it means that all the dependencies are unidirectional. Similarly, if adjacency matrix is a diagonal matrix, it means that there is no relationship and components are isolated. However, except these two information, it is failed to extract other important details like number of interaction parameters, type s of these parameters, interaction complexity and others.

Wu and Offutt (2003) performed static analysis to identify the interface events and the dependence relationship by using UML notation. The work provided a UML based framework to evaluate the similar ities among old and new components.

Sharma *et al.* (Sharma *et al.,* 2008) proposed a link-list based approach to represent the dependency relationship in CBS. Every component is represented by a node, consisting of all its required and provided interfaces. The provided interfaces of a component are accepted by other components as required interfaces. Component with required interfaces is called as dependent on component with provided interfaces. By using link-list based representation, along with the depende ncy, other information like name, number and type of interfaces, which are responsible for interaction can also be extracted. This information can further be used to measure the complexity of interaction, incoming and outgoing interaction density, most cri tical components and isolated components. All this information can be helpful in understanding, testing, debugging and maintaining the system. The details of the proposed methodology can be found in Chapter 3 of the present thesis.

1.3.3 Software Reuse and Reusability

Software reuse has been used as a tool to reduce the development cost and time of the software. The need for software reuse has become urgent as the size and complexity of software have started to escalate very fast. Software reuse is defined as the reuse of everything associated with a software project including knowledge (Basili and Rombach , 1988). It is the process whereby an organization defines a set of systematic operating procedures to specify, produce, classify, retrieve and adapt softw are artifacts for the purpose of using them in its development activities (Mili *et al.,* 1995). Reusability is the

degree to which a component can be reused and reduces the software development cost by enabling less coding and more integration (Wang, 2002).

1.3.3.1 Reusability Metrics for Legacy Systems

Selby (1989) studied several NASA projects, where software reuse was heavily used. He suggested that a reusable software module should be small in size with simple interfaces. It has few dependencies on other modules. Good documentation is also one of the properties. The reuse of the low level system and utility functions is more common than the reuse of human interface functions. He also validated these results statistically.

The ESPRIT-2 project, called as REBOOT (Reuse Based on Object-Oriented Techniques) developed a taxonomy of reusability attributes. As part of the taxonomy, they listed four reusability factors, a list of criteria for each factor, and a set of metrics for each criterion (Kitchenham and Kari, 1993). These factors include portability, flexibility, understandability and confidence. Although some of the metrics depend on subjective items such as checklists, an analyst can compute many of the metrics directly from the code, such as complexity, fan-in/out, and the comment-to-source-code into an overall value for reusability. Mili *et al.* (1995) considered two aspects, usability and usefulness to assess the reusability. Usability is the degree to which an asset is easy to use. Usefulness is the frequency of suitability for use. Paper also described the characteristics of usability and usefulness in order to better understand the concept behind these two aspects.

One major advantage of development with reuse is the lower cost of development in comparison to the development without reuse. Hence, one empirical factor for measuring reusability is the cost required to reuse a certain software artifacts. Caldiera and Basili (1991) proposed a model for reusability by considering three factors , namely, cost of reuse, usefulness of reusable components and quality of the reusable components.

Li (1998) proposed a cost benefit analysis for reuse, which helps to decide whether or not; reuse is a worth while investment. It proposed a metric, net cost savings, which is based on cost of project development from scratch, overhead costs associated with reuse and the actual cost of the software as delivered. Devenbu *et al.*, (1996) also considered cost benefit based approach, while proposing a metric, called, Reuse Ben efit

of a system, in terms of development cost. The work validated the proposed metric theoretically against standard Weyukar's properties (Weyuker, 1988).

1.3.3.2 Reusability Metrics for Software Components

Black box components have two main sources of information, the external documentation and the public interfaces. Gill (2003) suggested some guidelines for high reusability for software components. These guidelines include conducting reuse assessment, performing cost-benefit analysis for reuse, adoption of standards for software components, selecting pilot projects for deployment of reuse and finally identifying the reuse metrics.

Dumke and Schmietendorf (2000) proposed a set of reusability metrics for JavaBeans components. The metrics are adapted from structured and object-oriented design context. This work considers the source code to measure the metrics, therefore can not be applicable for black-box components.

Washizaki *et al.* (2003) proposed a metric suite for measuring reusability of black-box components. These metrics are based on the limited static information that can be obtained from the black box components. The work proposed that quality factors, namely, understandability, adaptability, and portability affect reusability of a component. Six metrics are proposed to measure these factors. 120 components were used to evaluate these metrics. Correlated correlation analysis is performed among these metrics. Results concluded that high understandability and adaptability lead to a high reusability of the component. However an independent validation performed by Goulao and Abreu (2004) showed the metrics to be unreliable for components with a number of features on their interfaces.

Boxal and Araban (2004) considered interfaces of the components to measure the reusability. The work assumed that understandability affects the level of reuse. Understandability of a component can be made through its interface properties. It also proposed some metrics by considering the size of the interface, argument count, argument repetition scale and others. These metrics give a better understanding of the properties of component's interfaces, which may help in measuring the reusability of the component. However, proposed approach does not consider the other aspects in the interface, such as, argument types and their complexity.

Rotaru *et al.* (2005) considered adaptability, compose-ability and complexity of a component to describe its reusability. Compose-ability and adaptability assess the ability of a component to be integrated as a part of a software system. Compose-ability is the easiness in combining a component with others, while adaptability is the ability of a component to accommodate changes in the environment. Paper also proposed metrics to measure these properties.

In order to develop an application by using existing components, it is essential to search the best suitable component from the repository. The developer wants to locate the highly reusable component that can be readily integrated to perform some clearly defined and distinct functionality within a large software system. Gui and Paul (2007) proposed coupling metrics for assessing and ranking the reusability of Java components retrieved from the internet by a search engine. An empirical comparison of the new metrics with several other existing metrics is also described, which concluded that the proposed metrics are superior with better results in ranking the components according to their reusability.

Sharma *et al.*, (2009) proposed a neural network based approach to measure the reusability of a software component. The work considered four factors, customizability, portability, interface complexity and understandability, which influence the reusability of black-box components. These four factors are considered as input parameters, while reusability is output parameter in order to train the network. Training and testing are performed by different number of hidden layers and neurons to get the best results. The results show that network is able to predict the reusability of the components with accepted precision.

1.3.4 Maintainability

Software maintenance, in general, refers to the set of activities that are performed to keep a system operational, as software changes after the system has been deployed. Software maintenance begins as soon as a system has been released to users for the first time in the case of incremental, evolutionary or spiral development. Maintainability is a high pre-requisite for reusable components and component-based systems, because there is no meaning in e.g. a long-live reusable component that is not maintainable. Therefore, maintainability in a system should be considered as one of the most important quality

aspect. A lot of research work has been carried out to measure the maintainability. To measure it, one need to identify the set of attributes/factors that bear on the efforts needed to make specified modifications. However, it may be very difficult to measure these factors and weighing/prioriting them against each other, and combine them to get final value of maintainability. Following section describes several metrics for maintainability for legacy as well as for CBS.

1.3.4.1 Measuring Maintainability

Khairuddin and Elizabeth (1996) proposed a maintainability model and included factors, namely, Modularity, Readability, Programming Language, Standardization, Level of Validation and Testing, Complexity and Traceability for evaluating the maintainability for software systems. Another model (Fioravanti and Nesi, 2001) was proposed for effort estimation/prediction of adaptive maintenance. It assumed that the system efforts for adaptive maintenance is typically spent performing several operations like understanding, addition and/or deletion of facts and modifications/changes of other system code portions. This metric is applied on several software projects and validated for predicting adaptive maintenance. The results shown that the proposed model is better ranked with respect to other models.

Bandini *et al.*, (2002) considered three independent factors, namely; design complexity, maintenance task and programmer's ability to predict the maintenance performance for object-oriented systems. To measure design complexity, interaction level, interface size and operation argument complexity were chosen. Perfective and corrective maintenance were selected to represent maintenance task. Correlation analysis was performed to conclude that the selected attributes were able to predict the maintenance efforts of the systems.

Another interesting work for measuring software maintenance project effort estimation has been done by Ahn *et al*. (2003). They proposed a software maintenance project effort estimation model (SMPEEM), which is based on the function point measure and 10 maintenance productivity factors. These factors are classified into engineer's skills, technology characteristics, and maintenance environment. Survey method was used to validate the proposed model. To find the relation of actual efforts and function points, regression analysis was performed by taking the data from 26

maintenance projects. Ardimento *et al.* (2004) made the assessment that if a component is difficult to understand then it will be difficult to maintain it and advised the trial usage of component before adopting it for the application.

Kajko *et al*. (2006) discussed several problems related with the maintenance of the software systems and proposed two maintainability models separately for product and process. Product aspects, consisted of common characteristics, variable characteristics, maturity level, traceability characteristics and others, while in process aspects the common tasks were to manage maintainability through out the whole software life-cycle. The work claimed that the commonly defined model of maintainability would substantially contribute to the overall success of the software.

1.3.4.2 Artificial Neural Network and Fuzzy Logic based Measurement of Maintainability

Artificial Neural Network (ANN) based approach is considered to be very helpful in estimating/predicting maintainability of the system. ANN is inspired from biological nerve system. ANN is characterized by its architecture, its learning algorithm, and its activation functions. A set of data (input/output) is used for training the network. Once the network is trained, it sets a relationship between input and corresponding output. Now this network can be used to predict the output for any input set of data. Researchers considered different factors as inputs for training the network. Some of the work is as:

Singh *et al.* (2004) considered readability of source code, documentation quality, understandability of software and average cyclomatic complexity as input or independent variable to measure the maintainability of the software systems by using the back propagation algorithm of Artificial Neural Network. Aggarwal *et al.* (2006) identified principal components of 8 object-oriented metrics as independent variables (input) to estimate the maintenance (output) for object-oriented systems. Back propagation algorithm was used for training the network. Shukla and Mishra (2008) used 14 factors as cost drivers for their study and trained the network by considering different number of hidden layers and neurons. Results concluded that neural network was able to predict the maintenance effort.

Aggarwal *et al.* (2005) used Fuzzy based approach to measure the software maintainability by considering average number of live variables, average life span of

variables, average cyclomatic complexity and the comments ratio. Grover *et al.* (2007, 2009) extended the ISO 9126 model to add one sub-characteristic, Trackability under Maintainability and proposed a Fuzzy Logic based approach to predict the maintainability of the CBS. Authors considered Interaction Complexity, Reusability, Testability, Understandability and Trackability as inputs to predict the maintainability as output. Inputs are designed by Fuzzy sets as Low, Medium and High, while maintainability as Very Low, Low, Medium, High and Very High. Total 243 rules were provided to fuzzy inference engine to get the output. The value of maintainability is measured then by using defuzzification process. The proposed model is then applied on a class room based project to evaluate its maintainability. The major findings of the paper are discussed in Chapter 5.

1.4 QUALITY OF COMPONENT-BASED SYSTEMS

On time, within the given budget and upto the level of satisfaction of the users are the ultimate requirements for any high quality software development project. It has to be noticed that software organizations invest some how 80% of their development resources for issues related to their product quality. In CBS, quality aspect becomes more important due to its architectural difference. Here, application developers have to rely on the component vendors or the developers. The quality of the component will have a very high impact on the quality of the final system. Following section discusses several quality models for legacy and for CBS.

1.4.1 Quality Models for Legacy Systems

Quality model is a set of characteristics and sub-characteristics, as well as the relationships between them that provide the basis for specifying quality requirements and for evaluating quality of the component or the system. Quality model establishes a framework to perform some kind of measurement of the specific desirable features that are needed in the final system and perceived by the end user (Losavio *et al*., 2002). Measurement of quality attributes is concerned with deriving the numerical values by using the appropriate metrics for that attribute.

There are several quality models proposed so far. The most well known model is McCall Model (McCall and Joseph, 1978). They presented a software quality framework

and classified the quality attributes into three groups, namely , product operation, product revision and product transition. Authors proposed 11 characteristics under these categories. Second model was proposed by Boehm and others (Boehm *et al.*, 1976), which in addition to most of the McCall factors; also include hardware characteristics and others like documentation, generality, economy, and others. Ghezzi *et al.* (1991) categorized quality into external and internal quality and list sixteen factors under these. FURPS model (Grady, 1992) decomposed quality into two different categories of requirements, namely, functional requirements defined by input and expected output and non-functional requirements like usability, reliability and others. Nagib and Callaos (1994) proposed the totality of quality concept for software development by giving a quality cube model. Dromey (1995) added two more characteristics, namely, reusability and process maturity to propose a new model. It categorized the characteristics into four categories: correctness, internal, contextual and descriptive.

ISO proposed a quality standard ISO 9126 (ISO, 2001) providing a generic definition of software quality, in terms of six main characteristics. These characteristics include functionality, reliability, performance, usability, portability and maintainability. These characteristics are further divided into 21 sub-characteristics, covering functional and non-functional behavior of the system. This model is a generic model. Researchers used this model by adding/removing characteristics/sub-characteristics from this, so that it can be used for component-based systems.

Bansiya and Davis (2002) proposed QMOOD, a hierarchical quality model, dedicated to the assessment of object-oriented design quality. The model indentified the specific design properties for object-oriented paradigm (e.g. polymorphism, inheritance, data abstraction etc.) and also introduces a set of new object-oriented metrics.

1.4.2 Quality Models for Component-based Systems

For component-based systems, most of the models, discussed above, may not fit as-it-is due to the difference in development approach. While some of the factors are appropriate for software components, others may not be well suited for those components. Therefore, one need to filter the characteristics and sub-characteristics for the existing model, which may fit for CBS and also some new characteristics specific to CBS, may also be added.

Woodman *et al.* (2001) considered the quality for the component user (component-based application developers) and suggested eleven quality attributes. These attributes include reusability, maintainability, accuracy, clarity, replaceability, interoperability, scalability, performance, flexibility, adaptability, and reliability.

Bertoa and Vallecillo (2002) modified the definition of characteristics and sub-characteristics proposed in ISO9126 model within the context of components. The paper classified the metrics for all these sub-characteristics into four categories: presence, ratio, time and level. The paper also suggested attributes and their associated metrics for these sub-characteristics for CBS.

Sedigh *et al.* (2001b and 2003) suggested the risk and quality management with the help of metrics and proposed a set of thirteen metrics, categorized into management, requirement and quality. Larsson (2004) collected several quality attributes for his proposed model. These attributes include business, nomadicity, and software system independence, footprint, schedulability, CPU utilization, latency, openness and several others.

Jung *et al.* (2004) conducted a survey on 200 users of a marketing department of a company. They used principal component analysis (PCA) to analyze the characteristics and sub-characteristics of ISO 9126 model. Results show that analyzability, changeability, stability and adaptability are related to maintainability and portability. The work also proposes that security, a sub-characteristic of functionality, should be considered as a separate characteristic and its sub-characteristics should also be proposed.

Voas and Agresti (2004) quantified the quality of software as a single variable by combining the contribution of all its attributes in units. All attributes may be measured by using different units; therefore these attributes need to be normalized so that they all appear in a range from 0 to 1. New weight values are proposed for each attributes to finally evaluate the quality of the software. Cai *et al.* (2005) proposed a generic quality assessment environment, named ComPARE (Component-based Program Analysis and Reliability Evaluation). It evaluates the quality of component-based software system and automates the collection of different metrics (including process metrics, static code metrics and dynamic metrics), selection of different prediction models, and formulation of user-defined models and validation of the established models.

Crnkovic *et al.* (2005) extended ISO 9126 to propose a new model for COTS components, which categorized the quality characteristics into runtime and life cycle. It removes portability, maintenance and reliability from the model and changes the definition of usability, learnability, understandability and operability in the context of components.

Rawashdeh and Matalkah (2006) added compatibility sub-characteristic under functionality and complexity under usability. It removed stability and analyzability from maintenance and added manageability to it. It also added new characteri stic, stakeholders to the proposed model. However, the proposed model did not specify any methodology to evaluate the characteristics and the quality of the system. Preiss *et al.* (2001) proposed several characteristics under observable at runtime and obser vable at development life cycle. These include integrability, deployability, dependability, safety, distributability and others.

Tawfiq *et al.* (2007) used quality attributes to propose a new cost estimation model. They used case-based reasoning to characterize the project for which estimate is to be made, relative to just enough quality attributes for that project. The description is then used to find other similar already finished projects, and an estimate for the new project is made based on the known ef fort values for those finished projects.

Sharma *et al.* (2008b) proposed quality model by adding complexity, reusability, flexibility and trackability and removing attractiveness and analyzability sub-characteristics from ISO 9126 model. Paper considered that, only the required characteristics/sub-characteristics to a particular domain should be considered for evaluating the quality. Like for throwaway applications e.g. a system developed to solve Y2K problem, maintainability will be of no use. Similarly, i f the application is intended to work only for a dedicated platform, portability aspect is not required at all. However, the financial applications will require more security, efficiency, reliability, fault tolerance and availability. Similarly, for an e-Commerce system, the important characteristics would be reliability, performance, availability, security, and maintainability (Voas and Agresti, 2004). Authors conducted a survey on 36 experienced software professionals from nine multi-national companies, working on the component-based technologies. Survey form consists of questionnaire about their preferences of quality characteristics. It

used Analytical Hierarchy Process (AHP) to assign weight values to these characteristics and sub-characteristics. Finally, a case study was performed on a live environment by taking the metrics for selected attributes to find the overall quality of the target component. This value can be used to compare and select the best component for the end product. The details of this work are presented in Chapter 6 of the present thesis.

1.5 CONCLUSION

It is evident from review of the literature that there exist no precise criteria to measure the various quality characteristics, like complexity, reusability, maintainability and others for component-based systems. Most of the work is either proposed theoretically without any evaluation and validation methodology or considers the source code of the components while proposing the metrics for these characteristics. Moreover, the relationships among these characteristics to obtain the overall quality as a single variable have not been explored.

The proposed study aims to estimate quality characteristics of black-box components and component-based systems. The work proposes and validates metrics for complexity, reusability and maintainability of the system. It also proposes the evaluation of quality for a component as a single variable. These estimates will help application developers to select the best quality component among others, which w ill eventually lead to the development of good quality product. The study uses Neural Network, Fuzzy Logic and Analytical Hierarchy Process based approaches to propose these measures.

Chapter 2

INTERFACE COMPLEXITY

2.1 INTRODUCTION

A component-based system (CBS) is integration centric with a focus on assembling individual components, to develop the application. In CBS, component source code information is usually unavailable. Each component introduces properties such as constraints associated with its use, interactions with other components and customizability properties. Customization can be performed through its interface methods and properties. Those components might be implemented by different programming languages and might be run on distributed machines; some of them might be developed in-house, while others may be third party or commercial off-the-shelf components (COTS). A key to the success of CBS approach to software development is its ability to use software components that are often developed by and purchased from third parties. Therefore, the selection of a component is of prime importance for helping a developer to choose the best component. One criterion for selecting the best component may be the complexity. It may also be an important indicator for estimating reusability, maintainability and overall software development efforts. If developer knows the method to evaluate the complexity of a target component, he can compare several available components and choose the appropriate one, which will be less complex than others.

Components are black-box in nature. The source code of these components is not available. Application may interact with these components only through their well-defined interfaces. Interface acts as a primary source for understanding, use and implementation for the component. Interfaces spell out the individual elements of a component in a systematic manner. Therefore, the complexity of these interfaces plays a lead role while measuring the overall complexity of the component. However, the overall

The major findings of the Chapter have been published in International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Vol. 18, Issue 6, November 2008, pp: 919 -931.

complexity of a component may also include integration complexity besides interface complexity (Narasimhan and Hendradjaya, 2004). Integration complexity is discussed in Chapter 3 of this thesis.

This chapter proposes an interface complexity metric for software components, which is based on complexity involved in the interface methods and properties used in the interface. These interface methods may have parameters and return values, which are the only source of information available to us for that interface method. Depending on the nature and the number of these parameters and return values and properties, weight values may be assigned to them, which can be used to measure the complexity of the target interface method. Finally, the sum of complexities of these methods will give the overall interface complexity of the target component.

2.2 COMPLEXITY CONCEPTS

IEEE defines software complexity as "the degree to which a system or component has a design or implementation that is difficult to understand and verify" (IEEE, 1990).

Basili (1980) defines complexity as a measure of the resources expended by a system, while interacting with a piece of software to perform a given task. If the interacting system is a program, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing or modifying the software.

There is no consensus on how to define software complexity, and the term complexity measure is a misnomer: The true meaning of the term software complexity is the difficulty to maintain, change and understand software (Zuse, 1991).

These definitions associate software complexity with the difficulty of performing a task on the software. An implicit assumption is that software complexity correlates well with reusability, maintainability or overall development efforts.

In conventional software development, complexity can be defined as the difficulty to analyze source code, modify and maintain its modules. However, in CBD, due to its black box nature, component may only be customized, added or removed. Customization can be performed through its interface methods and properties.

2.2.1 Complexity Metrics for Legacy Systems

Complexity is a major driver of the cost, reliability, and functionality of software systems. Several metrics have been proposed for measuring various aspects of complexity such as size, control flow, data structures, and inter-module structure. The most widely used complexity metric is Cyclomatic Complexity proposed by McCabe (1976). This metric is based on the program graph and is defined as :

$$V(G) = e - n + 2p \qquad (2.1)$$

where e is the number of edges, n is the number of nodes in the graph and p is the number of connected components. Author proposed that $V(G)$ can be used as a measure of procedure complexity of the program. Halstead (1977) complexity measurement was developed to measure a program module's complexity directly from source code (operators and operands in the module), with emphasis on computational complexity. These measures are based on four scalar numbers, derived directly from a program's source code. These numbers are:

n_1: the number of distinct operators, n_2: the number of distinct operands,

N_1: the total number of operators and N_2: the total number of operands.

From these numbers, five measures are derived as shown in Table 2.1. Among the earliest software metrics, these measures served as strong indicators of code complexity. Because they are applied to code, they are the most often used as maintenance metrics (Edmond, 2007).

Measure	**Symbol**	**Formula**
Program length	N	$N = N_1 + N_2$
Program vocabulary	n	$n = n_1 + n_2$
Volume	V	$V = N * (\log_2 n)$
Difficulty	D	$D = (n_1/2) * (N_2/n_2)$
Effort	E	$E = D * V$

Table 2.1: Halstead Complexity Measures

Kafura and Henry (1981) also proposed the complexity metric based on the number of local information-flows entering *(fan-in)* and exiting *(fan-out)* in each procedure. This metric is formulated in Equation 2.2.

$$Complexity = (Proc.\ Length) * (fan\text{-}in * fan\text{-}out)^2 \quad (2.2)$$

But in component context, these metrics may not be used to measure the complexity of entire component as these metrics measure the complexity for procedures only and do not consider the other aspects of the components like classes, attributes and interface methods, which may contribute a good amount of complexity to the component (Sharma *et al.*, 2007a).

Li (1983) proposed other metrics for complexity, based on size measures such as number of methods and attributes. In an object-oriented environment, Chidamber and Kemerer (1994) proposed a set of quality measures for class complexity. They proposed Depth of Inheritance Tree (*DIT*), Number of Children (*NOC*), and Weighted Methods Complexity (*WMC*), Coupling between Objects (*CBO*), Response for a Class (*RFC*) and Lack of Cohesion in Methods (*LCOM*). Out of these metrics *DIT, NOC, CBO* and *RFC* evaluate the external complexity of the relation between classes and do not depend upon the code complexity of the methods. *WMC* measures the complexity of the methods and is defined as:

$$WMC(C) = \sum_{i=1}^{n} C_i \quad (2.3)$$

where C_i is the static complexity of the corresponding method M_i. This metric evaluates the complexity of methods for a class.

2.2.2 Complexity Metrics for Component-based Systems

An extension to the metrics (Li, 1983) is proposed by Vernazza *et al.* (2000) for software components. They consider that a component consists of a group of classes, so it is reasonable to expect that the complexity of the various classes influence the complexity of the resulting component. They proposed a metric called weighted class per component (*WCC*) as:

$$WCC = \sum_{i=1}^{n} NOM(C_i) \quad (2.4)$$

where *NOM* is the number of methods available in the component. The other extended metrics are:

- Number of Children for a Component (*NOCC*), which counts the number of children of all the classes in the component.
- External Coupling between Objects (*EXTCBO*) counts the number of external classes, coupled to it.
- Response Set for a Component (*RFCOM*) is the number of all the methods in the member classes and the methods called by those classes.
- Maximum of the DIT (*MAXDIT*) and Mean DIT of unrelated trees (*MUT*).

The paper also performs a validation of these proposed metrics by using the properties defined by Briand *et al.* (1999). This validation is based only on some theoretical properties and no validation, even empirical has been performed so far.

Cho *et al.* (2001) propose various metrics for complexity, customizability and reusability for component-based systems. Authors classified the proposed metrics into design time metrics and implementation metrics. The work considers classes, interfaces and relationships among classes to measure the complexity metrics. These metrics combine cyclomatic complexity with the sum of classes and interfaces to give plain, static and dynamic complexity metrics, which may eventually help to estimate the component's size estimates. These metrics consider only the number of constituents like classes, methods and interface methods etc., but do not consider the technical complexities of these constituents.

Nael (2006) considered that structural complexity of component-based software systems is due to the components, connectors between components, interfaces of each component and composition tree. Author proposed several metrics for each of these categories. For interfaces, they proposed metrics, namely, total number of interfaces and average number of interfaces per component. However, the proposed metrics are very basic in nature and are based on just the numbers and does not consider the complexity of individual interface.

Most of the metrics discussed above, are based on the source code of the components and therefore cannot be used at the system level. Therefore, there is a strong need for designing of some complexity metric for black-box components, which can be used by the application developers to choose the best component and finally can produce the better quality systems. For black-box components, complexity is mainly due to the

interface, integration and semantics. Therefore, to measure the overall complexity of the component, we may have to measure all these. Interface measures are the estimates of the complexity of interfaces. Interface defines component's provided services and acts as a basis for its use and implementation. It acts as one of the primary definitive source for understanding component and often may be the only available source. An interface comprises of a set of operations, which act as access points for interaction with the outside environment. Integration measures are the measures of efforts required in the integration process of components. Lastly, semantic measures estimate the complexity of the relationship of components to application. Mahmood and Lai (2005) consider these three into consideration while designing the complexity for UML Component-based System Specification (CBSS). As the proposed metrics measures the CBSS complexity early in the specification phase to identify the complex components, they can direct the appropriate amount of effort to test and maintain these components to produce more reliable CBS.

Boxall and Araban (2004) define Interface Textual Complexity Metrics through a set of mathematical expressions. These metrics considers various measuring aspects of component's interfaces, such as interface size, number of distinct arguments in operations, level of repetition of such arguments, the commonality in identifiers, identifier's length and the density of reference arguments. The proposed metric gives a very rich analysis for the arguments used, however it considers all the arguments of same type and does not differentiate them on the basis of their type and complexity involved in that argument due to its data type.

Gill and Grover (2004) proposed a metric, called Component Interface Complexity Metric (*CICM*). The work discusses the interface characterization of software components and assumes that the complexity of a component is mainly due to interface data signature, interface constraints and interface packaging and configurations. Interface signature characterizes the functionality of the component and consists of properties, operations and events. Interface constraints involve the individual elements and the relationships among these elements. Last is the Interface packaging and configurations, which deals that how a component will be used in an application or in another component. First two parts of this metric deal with the internal functioning of the

component and depend on the coding involved during the development while the third one deals with the use of the component after the development. Although, the proposed metric seems to be a very strong candidate for measuring interface complexity but it lacks any sort of empirical evaluation.

2.3 PROPOSED COMPLEXITY METRIC

Rotaru *et al.* (2005) considered the interface-methods based approach to measure composability degree of a software component. It considers the parameters and return values of its interface methods to measure the proposed metrics. It justifies that software components interfaced only by methods with no parameter and no return value has the biggest composability degree because it does not have any external dependencies. While interface methods with no parameter with return value will have lower composability degree. Lastly, interface methods with both, parameters and return values have the lowest composability degree.

We extended the approaches described in (Rotaru *et al.*, 2005; Gill and Grover, 2004; Boxall and Araban, 2004) while proposing a new interface complexity metric for components. Proposed metric uses the signature or the behavior of the component through its interface methods and properties, which are available even without going into the internals of the component. For the proposed metric, we consider the events and their listeners similar to the methods. We propose that the interface complexity metrics for the component will be due to the complexities involved in its interface methods and properties described above and define Interface Complexity Metric (*ICM*) for Component *C* as:

$$ICM(C) = a\sum_{i=1}^{n} CIM_i + b\sum_{j=1}^{m} CP_j \qquad (2.5)$$

where CIM_i is the complexity of i^{th} interface method and CP_j is the complexity of j^{th} property. *A* and *b* are weight values for methods and properties respectively, as complexity of a interface method may have different weight value than the complexity of a property. However, in this work, we are taking these weight values same and equal to 1. Interface methods are divided into the following categories:

- Interface methods without return values and without parameters.
- Interface methods with return value but without parameters.
- Interface methods with no return value but with parameters.
- Interface methods with return value and with parameters.

Complexity of interface methods may be measured based on its return type and arguments passed to it. We assign different weight values to these methods based on the nature (data type) of arguments/return values, used in the method. Arguments/Return types may be of primitive data types like integer, structured data types like date, string, array list, vector and complex data types like class type, built-in and user-defined components, pointers/reference and others. Therefore, based on the complexities involved in these data types, different weight values are assigned to the methods. We classify these data types in four categories, namely, simple, medium, complex and highly complex. Similarly based on the data type, we can categories properties into simple, medium, complex and highly complex and can assign the corresponding weight values to them.

Our assumption in assigning weight values is that a method having no argument (e.g. constructor) may be considered as simplest method and we assign the weight value to these methods 0.05. All other interface methods are assigned weight values depending on the type and total number of arguments and return types. The following table shows these weight values:

Data Type → / No. ↓	Simple	Medium	Complex	Highly Complex
1-3	0.1	0.15	0.2	0.25
4-6	0.2	0.3	0.4	0.5
7-9	0.3	0.45	0.6	0.75
>=10	0.4	0.6	0.8	1.0

Table 2.2: Weight Values for Interface Methods

Same table can be used for getting the weight values for properties used in the component. Based on these tables, the complexity of each interface method and property can be measured and finally by taking appropriate values of a and b, we can measure the interface complexity of the entire component by Equation 2.6. The metric seems to be fit

for empirical evaluation. This complexity metric can be correlated with other quality criteria such as performance, customizability and others. Customizability can be used to measure the reusability and maintainability for CBS, as direct method to measure maintainability may not be possible **(**Sharma *et al.*, 2007b).

2.3.1 Theoretical Evaluation of Proposed Metric using Weyuker's Properties

Weyuker (1988) proposed an axiomatic framework in the form of several properties for evaluating complexity aspect. The proposed interface complexity metric in this chapter is evaluated against these properties for compatibility. The properties are:

Property 1: There are programs P and Q for which M(P) ≠ M(Q).

Property 2: If c is non-negative number, then there are finitely many programs P for which M (P) =c

Property 3: There are distinct programs P and Q for which M(P)=M(Q)

Property 4: There are functionally equivalent programs P and Q for which M(P) ≠M(Q)

Property 5: For any program bodies P and Q, we have M(P)<= M(P;Q) and M(Q) <= M (P; Q).

Property 6: There exist program bodies P, Q and R such that M(P)=M(Q) and M(P;R) ≠ M (Q; R).

Property 7: There are program bodies P and Q such that Q is formed by permuting the order of statements of P and M(P) ≠M(Q).

Property 8: If P is a renaming of Q, then M(P) = M(Q).

Property 9: There exist program bodies P and Q such that M(P)+M(Q) < M(P;Q).

These properties are evaluated for the proposed metric as:

- There may be two different components with different complexities, thus satisfying the first property.
- As component will have at least one business method with some functionality, therefore its complexity will always have some positive value. It confirms the second property.

- For two different components with different functionality, complexity metric value may be same, as these methods may have same interface structure but with different functionality. Property 3 is satisfied on the proposed metric.
- Even if the functionality of the two components is same, both may have different complexities as these components may be designed by using different technologies and programming concepts. It confirms 4th property.
- If a component is integrated in another component to get an assembly for enhanced functionality, the complexity of these two individual components will be lesser than the complexity of the assembly, which is as per the 5th Weyuker property.
- Two components with the same complexity means both will have same no. of interface methods with same arguments and return types. However, they may be developed by using different programming methodologies and therefore when integrating in the system/component, both may have different integration code and implementation thus resulting in different complexities of the system in both the cases. Therefore this property is satisfied for the proposed metric.
- The ordering of interface methods and return type and arguments in a component will not change the complexity of the new component. Thus this property is not satisfied by our metric.
- It is obvious that renaming a method or a component will not affect the complexity of that interface or the component, thus satisfying this property.
- When two components are assembled then besides the methods used in these components, we may have to write some methods related with the integration also. This will increase the complexity of the assembled component. This satisfies the last property.

Out of nine properties, eight properties are satisfied by the proposed metric.

2.3.2 Empirical Evaluation of the Proposed Metric

Above proposed metric is evaluated on a number of Java Bean components, collected from several websites (Jars, Elegantjbeans, Oreilly, Java). These Java Beans vary from very simple and small to complex and very large. The parameters passed to

and value returned from these components varies from simple *int* to complex object or component type. Also, the number of properties varies from 2 to 75. The values obtained for complexity from these Java Bean components by using Equation 2.5 are shown in the Table 2.3.

Java Beans	Methods	Properties	Interface Complexity
Temperature	11	27	8.45
Boiler	16	35	11.95
AcmeLabel	6	2	1.30
WordCount	3	2	1.40
TableBean	19	76	15.80
ConvertBean	6	4	2.20
AcmeSizing	9	4	4.35
TempModi	21	13	13.80
Tree	16	22	22.65
Calendar	15	10	18.75

Table 2.3: Complexity Metric Values

Table 2.3 shows that the Java Beans considered for the experimentation vary from very simple (only three simple methods and 2 properties) to complex (around 20 methods and 75 properties). The complexity of these components varies from 1.30 to 22.65.

2.3.3 Validation of the Proposed Metric

Correlation analysis is performed for the proposed metric with several other quality characteristics, like, performance, customizability and readability on same Java Bean components used for proposed complexity metric. For performance, time taken by these components (with default values of parameters) to execute in seconds is measured. Customizability is defined as the ability to modify a component as per the application requirement. Better customizability will lead to a component with better reusability. It will also help in maintaining the system in the later phases. Therefore, it can be used to measure the maintainability and reusability for CBS. It may be measured on the basis of writable properties available in the component. Writable properties in Java Bean

components may be recognized by *set* methods (Washizaki, 2003). The following formula is used to evaluate this metric:

$$Customizability = \frac{No.\ of\ Set\ Methods}{Total\ number\ of\ Properties} \qquad (2.6)$$

Similarly, readability can be measured by getting the observable properties from the component. Readability will help an application developer to understa nd the component. If a component is understandable, it will be easier to use it and maintain it. Therefore readability will improve the usability, reusability and maintainability of the component. It may be measured on the basis of readable properties avai lable in the component. Readable properties in Java Bean compone nts may be recognized by *get* methods. The following formula is used to evaluate this metric:

$$Readability = \frac{No.\ of\ Get\ Methods}{Total\ number\ of\ Properties} \qquad (2.7)$$

Table 2.4 shows the values of three metrics i.e. Execution Time (ET), Customizability (Cust) and Readability (Read) for the same Java Bean components.

Java Beans	**ET**	**Cust.**	**Read.**
Temperature	0.031	0.185	0.148
Boiler	0.109	0.31	0.20
AcmeLabel	0.010	0.50	0.5
WordCount	0.016	0.0	0.0
TableBean	0.90	0.04	0.07
ConvertBean	0.011	0.35	0.5
AcmeSizing	0.019	1.0	1.0
TempModi	0.912	0.05	0.08
Tree	0.516	0.0	0.09
Calendar	0.39	0.02	0.10

Table 2.4: Other Metrics Values

Now, Correlation theory is applied to find the relationships among proposed complexity metric and these three metrics. For this purpose Karl Pearson's coefficient of correlation is used. Table 2.5 shows the result obtained.

Characteristics	**Correlation Coefficient**
Complexity vs. Exec. Time	0.78
Complexity vs. Customizability	-0.51
Complexity vs. Readability	-0.56

Table 2.5: Correlation Coefficients among proposed and other metrics

Table 2.5 shows that there is a strong positive correlation between complexity and execution time. Complex components take more time to execute, which is self evident and is proved by our proposed metric. Also, there are negative correlation between complexity and customizability and complexity and readability, which confirms that highly complex components are hard to customize and to understand which leads towards poor reusability and maintainability. These correlation coefficients and their interpretation validate the proposed complexity metric for components.

2.4 CONCLUSION

The chapter discusses various complexity metrics proposed by researchers especially for component-based systems. Most of the metrics proposed so far are based on the source code of the component and therefore cannot be used by the application developers, who do not have the source code of these components. We proposed an interface complexity metric for black-box components, which is based on the complexity of the interface methods and properties existed in the component. Work theoretically evaluates the proposed metric against Weyuker's properties and then empirically evaluates it on several Java Bean components. It also conducts validation through other metrics, namely, performance, customizability and readability and verifies the facts that complex components take more time to execute and are hard to maintain and reuse. The proposed metric can be used by the system developers to evaluate the complexity of the component at an early stage before using it in the system for ensuring the good quality end-product.

Chapter 3

DEPENDENCY ANALYSIS

3.1 INTRODUCTION

Different programmers develop components, often working in different groups by using different methodologies. Interaction among these components can be characterized by the use of component's interface or through other components interactions. In other words, interaction happens when a component provides an interface and other components use it, and also when a component submits an event and other component s receive it. Interactions promote dependencies. Integrating a component can affect the composite functionality of the system. It is hard to create robust and efficient systems, if they do not understand the dynamic dependencies among components. Thus, it is very common to find cases, in both legacy and component-based systems, in which a module/component fails to accomplish its goal because the system does not properly resolve an unspecified dependency. Sometimes, other modules or components do not properly detect the graceful failure of one module or component, which leads to a total system failure.

Because administrators must continuously update and modify current systems, dependency conflicts may arise. For example, operating system administrators must monitor security announcements daily and be prep ared to update their operating system kernels with the appropriate security patches. In addition, users demand new versions of applications such as web browsers, text editors, software development tools, and so forth. Often, building and installing a new s oftware package requires updates to a series of other components as well (Kon and Campbell, 2000). Recent research suggests that most faults are found due to the poor interactions among components. If we can identify these possible faults early at specific ation, then precautionary actions can avoid the likely failure causes and costly maintenance (Mahmood and Lai, 2005).

The major findings of this Chapter have been published in ACM SIGSOFT Software Engineering Notes, Vol. 34, Issue 4, July 2009, pp: 1-8.

Understanding and tracking dependence relationships among components is increasingly difficult in large and complex systems. The problem is intensified since CBS encompasses both components developed in-house and components made available by a third party (e.g. COTS), often deployed with insufficient documentation (Vieira and Richardson, 2002). Undocumented dependencies can prevent the ability to evolve a CBS when it is required to do so. In addition, unidentified dependencies can cause performance related problems and failures which are extremely difficult to find in large systems. Thus, it is important to provide ways to comprehensively identify dependencies of an individual software component and to analyze and manage its influence in a CBS (Vieira and Richardson, 2001). Dependency analysis helps to answer the following questions in a component-based system:

- If a component is updated, which other components in the system are affected? It will help to predict the maintainability efforts.
- If a new system of component is installed, what is the effect on the system?
- Which components are more important/critical than others? and which components are isolated?
- What is the minimal set of components of the system that must be inspected when a failure of the system occurs?
- What is the difference between two configurations?

To address all these issues, it is important that dependencies among components should be represented by an efficient and effective manner, which can retrieve the information in a faster way. The present chapter proposes a new link-list based representation for storing dependencies among components. This representation is implemented by using HashMap concept in Java. HashMap based representation may store the component key and the complex objects, like class. Through this representation, we can store information like incoming/outgoing interactions, type of interactions (simple/complex) etc. This representation helps us to analyze the dependency level, which is an indirect measure of complexity. This information can also be used to evaluate interaction density for the component and finally for the whole system.

3.2 DEPENDENCY

Components provide system functionalities by interacting, cooperating and coordinating. These will produce dependencies among them. Usually, a group of components depend on each other in order to supply complex system functionality. Any modification to a component can cause the change of composite functionality, because the composite functionality is reflected in different components. In addition, the replacement of a new version component will also cause the change of dependency among components (Li, 2003).

Dependency may be defined as (Lisa and Delugach, 2001):

Dependency is a relationship involving two or more components, where a change of state in one or more component leads to a potential for a change of state in one or more other components.

In the simplest case of dependency, a unidirectional dependency between two entities, *d(A, B)*, implies that *A* depends upon *B*. If *A* depends upon *B*, then a change in *B* implies a potential or possible change in *A*. *A* is referred as dependent and *B* as the antecedent (Keller *et al*., 2000).

3.2.1 Dependency Analysis and Component-based Systems

Dependency analysis involves the identification of interdependent components of a system. These interdependent components work as subsets of the system. It has been widely studied for purposes such as code restructuring during optimization, automatic program parallelization, test-case generation, and debugging. Dependence relationships at the architectural level arise from the connections among components and the constraints on their interactions (Stafford *et al.*, 2003).

The benefits of dependency information are clear in providing the developers and maintainers with significant knowledge about the system and its maintenance. It is important to identify possible dependencies not only to understand which parts of a CBS can be affected by the evolution, but also to distinguish sources (roots) of potential problems, where more attention is needed. There are reported cases of failures in CBS because dependence relationships were not identified or properly resolved during the

system's evolution. Particular problems exist for mission-critical systems with high-dependability requirements (Vieira and Richardson, 2002).

3.2.2 Existing Dependency Representation Techniques

Traditional graph theory can be used to monitor the dependencies among components. It is also useful to represent a graph with a matrix or adjacency lists to be able to determine if components are dependent or not. Dependencies among components are to be tracked and stored for further management. By using this approach, it is possible to analyze what has been affected in the system and to create determinism when updating the system with new components (Larsson, 2007).

Stafford *et al.* (2003) represented dependency among the components by graph G where $G=(N, E)$ is a directed graph which has a finite set of vertices N (called components) and E the set of dependencies (path) between the components. An adjacent matrix AM_{n*n} may be used to represent the directed graph, where

$$AM_{[i,j]} = \begin{cases} 1, \text{ if component } C_i \text{ is directly connected with } C_j, \\ 0, \text{ if component } C_i \text{ is not directly connected with } C_j \end{cases}$$

$$= \begin{pmatrix} d_{11} & d_{12} & d_{13} \ldots & d_{1n} \\ d_{21} & d_{22} & d_{23} \ldots & d_{2n} \\ \ldots & & & \\ \ldots & & & \\ d_{n1} & d_{n2} & d_{n3} \ldots & d_{nn} \end{pmatrix} \qquad (3.1)$$

where d_{ij} $(i,j = 1, n)$ is the $(i, j)^{\text{th}}$ state with value 1, if j^{th} component is dependent on i^{th} component, otherwise 0.

For example, dependencies in a component-based system consisting of five components, namely, *A, B, C, D,* and *E* (shown in Fig 3.1) can be represented by the matrix as:

$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}
\begin{array}{c}
\begin{array}{ccccc} A & B & C & D & E \end{array} \\
\left(\begin{array}{ccccc}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{array}\right)
\end{array}
\qquad (3.2)
$$

From this representation, the dependencies among components may be computed. Dependency of a component C_i to other components is the number of all paths in the graph from C_i to other component.

$$D_i \text{ (Dependency of } C_i) = \text{Sum of all paths (i to j)} \qquad (3.3)$$

where *j=1 to n* and *Path (i,j)* is the total number of paths from component C_i to C_j.

The dependency graph, which is represented by a matrix, can easily be used to determine what has been affected when a new version of a component is installed.

Gill and Balkishan (2008) used adjacency matrix based representation for CBS and introduced a set of component-based metrics, namely, Component Dependency Metric and Component Interaction Density Metric, which measure the dependency and coupling aspects for the software components respectively. The work concludes that higher interactions among components increase the complexity because of more coupling among components. Higher complexity means more expensive and less maintainable software. The proposed work still needs an empirical validation using data from real life applications. Moreover, the work uses graph theory based notations and adjacency matrix to show dependency. However, this approach only considers the presence of interactions (1 or 0) and does not consider the type of interaction while measuring the complexity. For example, an interaction involving only simple interface like parameter of type *int* will have low interaction complexity in comparison to interaction involving complex interface as when parameter is of component type. The type of interaction among components may have a significant contribution in the interaction complexity of the component.

Narasimhan and Hendradjaya (2004) proposed static and dynamic metrics for component integration. Static metric covered the complexity and the criticality within an integrated component and is intended to be used early during the design stage. The work

proposed two metrics called Component Packing Density (CPD) and Component Average Interaction Density (CAID) to represent the complexity of the system. CPD is the ratio of number of constituents like modules, classes, operations etc. and the total number of components; while interaction densities are measured by dividing the actual incoming interactions to the total number of incoming interactions and similarly dividing the actual outgoing interactions to the total number of outgoing interactions. The results deduced are shown in Table 3.1.

CPD	**CAID**	**Result**
Low	Low	Low data processing and low computation (Simple transaction systems)
Low	High	Low data processing and high computation (Compute-intensive real time systems)
High	Low	High volume of data with many components but with low interaction among components (Transaction processing systems)
High	High	Many classes or constituents with the components and high interactions among components (Complex Systems)

Table 3.1: Density Metrics

These metrics are further evaluated theoretically against standard Weyuker properties. Most of these properties are satisfied by the proposed metrics. This paper also discusses the relationship of these metrics with quality attributes defined in McCall (McCall and Joseph, 1978) model. The proposed metrics can be used to measure usability, efficiency, reliability, maintainability and other quality factors. However, these metrics are required to be empirically evaluated and validated against some live component-based applications.

A similar approach is adopted by Kharb and Singh (2008). Authors proposed several interaction metrics for component-based systems to finally measure the complexity of the system. These metrics are based on actual and total number of incoming and outgoing interactions among the components. However, these metrics are just proposed theoretically, without any correlation with any quality characteristic, with no evaluation and validation on real-life applications.

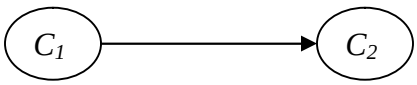
3.3 PROPOSED COMPONENT DEPENDENCY REPRESENTATION USING LINK-LIST

The main idea of our approach is to provide mechanisms for:

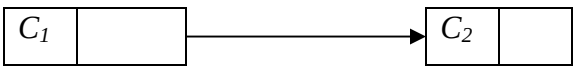
- Describing dependencies of an individual component with the overall system and
- Analyzing the influence of such dependencies in terms of type of interactions, dependent components in a CBS in which the component is assembled, by deploying the link-list representation strategy.

In the proposed representation, each component has a list of components which are dependent on it. The length of a dependency list gives the number of dependent components and can be used as a measurement of dependency complexity. The higher the number, the more complex is the relation with other components.

Consider a simple case involving two components C_1 and C_2. C_2 requires some functionality from component C_1. Therefore C_2 is dependent on C_1.



This can be represented by linked list as follows:



Use of this approach is though resource consuming but allows faster searches for all components dependent on other components. This will result in better understanding of the system and will decrease the overall maintenance efforts.

3.3.1 Implementation of the Proposed Methodology

We propose a HashMap based approach for the above representation, which stores the dependency of components, represented by linked-list as discussed above. Map is a class that stores key/value pairs. The HashMap class provides the primary implementation of map interface. HashMap class uses a hash table for implementation of the map interface.

Each component is added as a key and all the inbound (required interfaces) and outbound (provided interfaces) interactions are encapsulated in a class object in HashMap. Outbound dependencies can be used to measure the interactions for a

component in the system, which, in turn can be used to estimate the complexity of the system. HashMap based approach is better than the traditional matrix-based approach in terms of efficiency and storage of additional information like incoming and outgoing interactions, their number and types etc. The components for which the interactions are to be used are stored as a key and the dependent components are stored in a list that contains the multiple objects of dependency class. The structure of the class will be as shown in Table 3.2.

```
public class Dependency
{
      Object  componentAsKey;
      List parametersUsedForInteractions;
        public void addComponentForInteraction(Object component)
        {
            this.componentAsKey = component;
        }
        public Object getComponentForInteraction(){
            return componentAsKey;
        }
      public void addParametersUsedForInteractions(Object parameter)
        {
            parametersUsedForInteractions.add(parameter);
        }
        public List getParametersUsedForInteractions()
        {
            return parametersUsedForInteractions;
        }
}
```

Table 3.2: Pseudo code for designing Link-List based representation

To find out the inbound interactions, we need to write a method that will traverse the HashMap to find out the number of inbound interactions. The structure of the method is given below in Table 3.3.

```
public int countInboundInteraction
                        (HashMap componentGraph, Object keyComponent)
    {
       Iterator it = componentGraph.keySet().iterator();
       int inboundCount = 0;
       while (it.hasNext())
        {
           List associatedComponents =(List)
```

```
componentGraph.get(it.next());
                if(associatedComponents.contains(keyComponent))
                {
                    inboundCount++;
                }
            }
           return inboundCount;
    }
```

Table 3.3: Pseudo code for Storing Inbound Interactions

Similar pseudo-code can be developed by traversing HashMap to trace the outbound interactions. The count of occurrences in values (from HashMap key/value pair) will give the number of outbound interactions. For each outbound interaction, the number of parameters can be found from Dependency class object. Total interactions, available incoming interactions, used interactions and other valuable information can be measured by using the same approach. These values can then be used to ascertain various interaction complexity measures for the component-based systems. Such metrics are summarized below:

a) For Individual Component

Based on the proposed methodology, incoming and outgoing interaction density can be measured for a component. The dependent components on a parent component can also be identified. Interaction densities may be used to measure the integration efforts for the system, while dependency level helps in identifying critical and isolated components in the system.

i) Incoming Interaction Density

Incoming Interaction Density (*IID*) for a component *C* can be measured by dividing Used Incoming Interactions *UII (C)* to the Available Incoming Interactions *AII (C)*. *AII (C)* and *UII (C)* can be measured as:

AII (C) = Sum of all provided services of Parent Components of C

UII (C) = Sum of all required services for Component C

Therefore,

$$IID(C) = \frac{UII(C)}{AII(C)} \qquad (3.4)$$

IID may be used to measure the integration efforts for that individual component. Higher value of *IID* results in complex integration efforts, which will increase the maintenance effort also.

ii) Outgoing Interaction Density

Similarly, Outgoing Interaction Density (*OID*) for a component *C* can be measured by dividing Used Outgoing Interactions *UOI (C)* to the Available Outgoing Interactions *AOI (C)*. *AOI (C)* and *UOI (C)* may be measured as:

AOI (C) = Sum of provided services of component C

UOI (C) = Sum of required services of child components of C

Therefore,

$$OID\ (C) = \frac{UOI(C)}{AOI(C)} \qquad (3.5)$$

OID may be used as a measure of usability of component in the system. Higher value of this metric result in higher possibility of using this component by other child components, which is an indication of high dependability of this component within the system. Also, it may be used for measuring the service utilization. If all the provided interfaces of a component are utilized by other dependent components then it may be termed as efficient component in terms of service utilization. On the other hand, if some of the provided interfaces are not used by any of the dependent components; it means that the functionality provided by the component is not fully utilized by other components.

iii) Dependency Level

Dependency level (*DL*) of a component is the sum of all the child components of *C*.

DL (C) = Sum of child components of C (3.6)

This measure can be used to identify the critical components and isolated components in the system. Highest value of *DL* will be referred as the most critical component of the system. Any change in this component may require several possible changes in other dependent components also. On the other hand, 0 dependency level for a

component means an isolated or independent compo nent. It can accommodate any change without affecting other components of the system.

b) For the System

Interaction densities (incoming as well as outgoing) may also be measured for the whole system. Incoming Interaction Density for the system *S*, *IID (S)* may be defined as:

$$IID(S) = \frac{Sum\ of\ Incoming\ interactions\ for\ all\ components}{Number\ of\ Components} \qquad (3.7)$$

Similarly, Outgoing Interaction Density for the system *S*, *OID (S)* may be defined as

$$OID(S) = \frac{Sum\ of\ Outgoing\ interactions\ for\ all\ components}{Number\ of\ Components} \qquad (3.8)$$

Also, Average Interaction Density (*AID*) for the system *S* may be defined as:

$$AID(S) = \frac{Sum\ of\ Interactions\ for\ all\ components}{Number\ of\ Components} \qquad (3.9)$$

These measures can be used to measure the interaction complexity of the system , which may be a measure of integration efforts for the syste m.

3.3.2 Experimentation

We consider a simple system (Fig. 3.1) to illustrate the above discussed concepts and measures. There are total five components in the system. Every component is having some incoming interactions, referred as required interfaces and some outgoing interactions, referred as provided interfaces. For example, component *A* has x_1 as required interface and x_2 and x_3 are provided interfaces. These provided interfaces in turn can be used by other components as required interfaces. For compo nent *A*, out of two provided interfaces, only one is utilized (by component *B*). Similarly, for component *E*, only x_{10} is used by component *D*, leaving x_{13} as unused, although available.

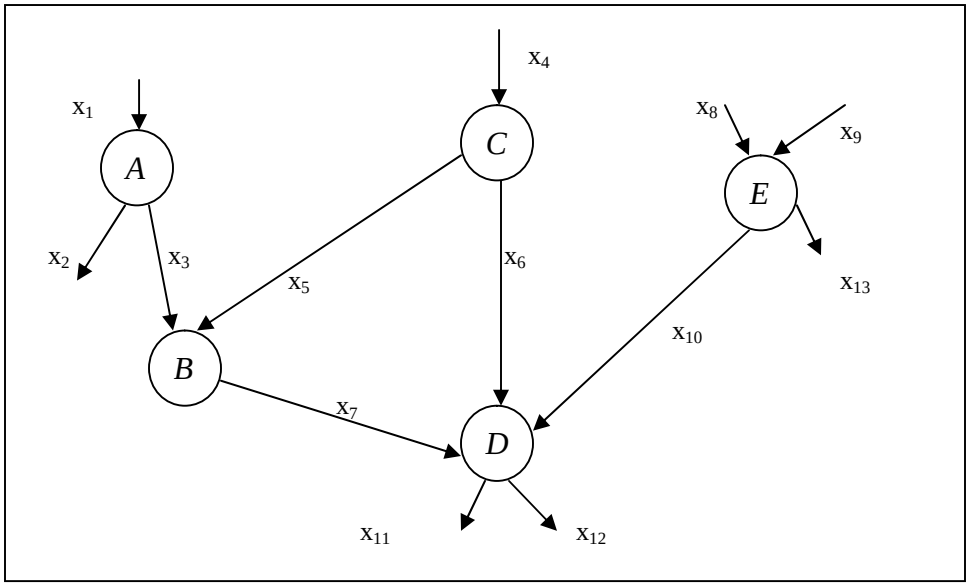


Fig 3.1: CBS with five Components *A, B, C, D* and *E*

This system can be represented as shown in Fig. 3.2 by using our proposed link-list based approach. Components shown in left most side are the parent components, while right side components are their dependent child components. Every component is having required interfaces *(Req. Int.)* on its left side, while provided interfaces *(Prov. Int.)* on the right side.

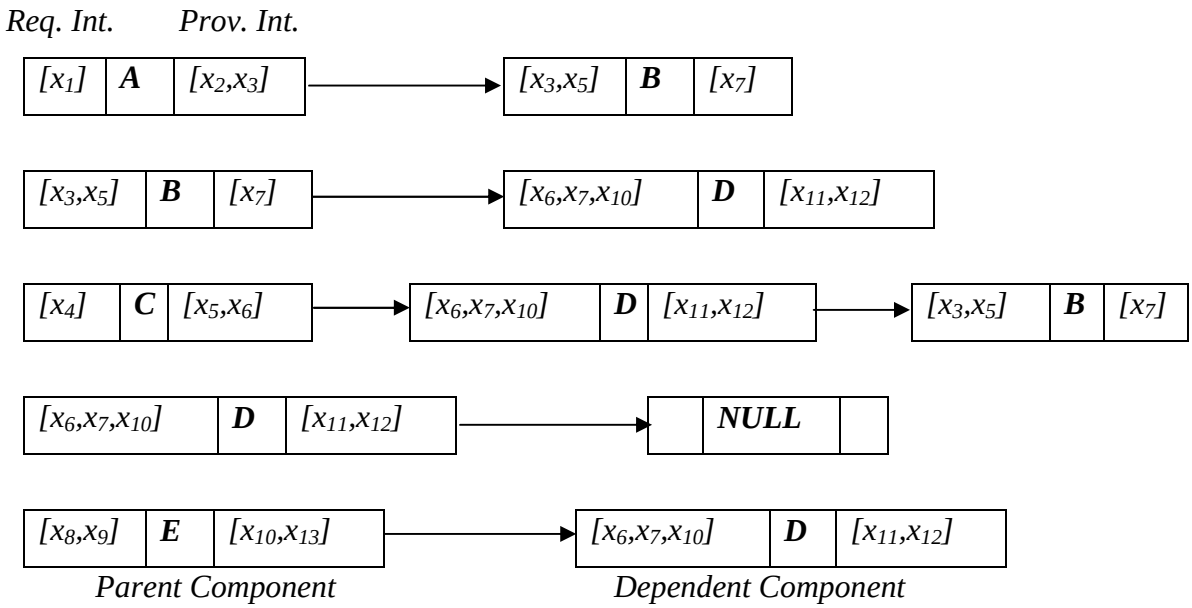


Fig. 3.2: Link-List Based Dependency Representation

Now HashMap based implementation of this representation can be used to add components in this list along with its parent and child components and incoming and outgoing interactions. This representation can be used to analyze the dependency levels (*DL*) and the dependent components for a particular component. This may also be used to analyze the impact of change on all components which are dependent on changed/replaced component. The values of various measures for these components are shown in Table 3.4.

Component	***IIC***	***OIC***	***DL***
A	0	½=0.5	1
B	2/4=0.5	1	1
C	0	2/2=1	2
D	3/5=0.6	0	0
E	0	0.5	1

Table 3.4: Dependency Metrics Values

Here in this example, D is having the highest IIC, which means that integration efforts for using this component will be more than that of other components in the system. Also, component B and C are fully utilized by their respective child components. Dependency level of D is minimum i.e. 0. If we need any change in D component, it will not affect other components. However, the most critical component is C, which has dependency level 2 and has two child components B and D. Any possible change in this component may require corresponding changes in both components.

3.4 CONCLUSION

The present work proposes a link-list based approach to represent dependency among components in component-based systems. This approach is more efficient than matrix based approach as it can store several type of information like, provided and required interfaces of components and their types, dependent components etc. This approach is expected to give faster searches for the components, dependent on other components. This chapter also discusses several metrics related with the interactions among multiple components in a system. The information may be used to measure the

interaction complexity of the system. Also, we may analyze several interaction and dependency related issues with the proposed approach.

Chapter 4

ESTIMATING REUSABILITY

4.1 INTRODUCTION

Software programming is a highly specialized and intellect-oriented task, mainly due to the complexity involved in the process. Nowadays, this complexity is increasing to levels, in which reuse of previous software designs are helpful to reduce the efforts and total development time. The main idea of software reuse is to use previous software components/artifacts to create new software systems. Thus, software reuse is software design, where already developed components are the building blocks for the creation of new systems. The main objective of software reuse is to minimize repetition of work, development time, cost and efforts and increase reliability of the systems. It also improves the maintainability and portability of the system.

In case of component-based development, software reuse refers to the utilization of a software component with in a product, where the original motivation for constructing this component was not known. Here, reuse is seen as black-box reuse, where the application developer sees the interface, not the implementation of the component. The interface contains public methods, user documentation, requirements and restrictions of the component. If there is a change in the code of a black-box component, compiling and linking the component would propagate the change to the applications that reuse the component. As the users of the component trust its interface, changes should not affect the logical behavior of the component. The clients will get what the contract promises only if the post condition is true after the changes to the internal implementation (Judith and Audrey, 1998).

One of the essential problems in software reuse is the retrieval and selection of suitable software components from a large library of components. Gill (2006) discusses the importance of component characterization for better reusability. It discusses several

The major findings of this Chapter have been published in ACM SIGSOFT Software Engineering Notes, Vol. 34, Issue 2, March 2009, pp: 1-6.

benefits of component characterization, which includes improved cataloguing, improved usage, improved retrieval and improved understanding eventually for better reuse.

For CBD, there are two broad reuse developments. One is development of systems with reuse and another is development of components for reuse. In first case, application is developed by reusing several already built-in components. These components are already tested thoroughly, which will enhance the quality of the end product and will save time and cost. Another approach is the development of components for reuse. Here, components are developed keeping in mind the high reusability. These components need to be compatible for wide range of applications developed in variety of languages for different platforms. Present chapter emphasizes on the first aspect of reuse and proposes a methodology to estimate the reusability of the target component before using it in the final system. It will help application developers to compare and chose the highly reusable component among others and will eventually lead to the application development with low maintenance efforts.

4.2 REUSABILITY METRICS

As the number of components available in the market increases, it is becoming more important to devise software metrics to quantify the various characteristics of components. Among several quality characteristics, the reusability is particularly important when reusing components. Reusability can measure the degree of features that are reused in building applications. There are a number of metrics (Banker *et al.*, 1991, Karunanithi and Bieman, 1993; Devanbu *et al.*, 1995; Frakes and Terry, 1996; Barnard, 1998; Kamiya *et al.*, 1999; Aggarwal *et al.*, 2005) available for measuring the reusability for object-oriented systems. These metrics focus on the object structure, which reflects on each individual entity such as methods and classes, and on the external attributes that measure the interaction among entities such as coupling and inheritance. But there are some difficulties in applying existing object-oriented metrics into the development of components and CBS. Object-oriented metrics cannot be used to measure the component's quality. The reason may be that in OO development, reuse is only limited up to class level and with in the same application, while in CBSD, we can reuse even the whole component and also in multiple applications.

An important issue in choosing the best component for reusability is deciding which components are easily adapted. Generally, good guidelines for predicting reusability are: small size of code, simple structure and good documentation. Gill (2003) discusses the various issues concerning component reusability and its benefits in terms of cost and time-savings. This work also provides some guidelines to augment the level of software reusability in component-based development, which are summarized below for completeness:

i) Conducting thorough and detailed software reuse assessment to measure the potential for practicing reuse in an organization so that it can be ensured that the organization can get maximum benefits from already practicing reuse.
ii) Performing cost-benefit analysis to decide whether or not reuse is a worthwhile investment. This analysis can be performed by using well-established economic techniques like Net Present Value (NPV) and others.
iii) Adoption of standards for components to facilitate a better and faster understanding of a component and a faster integration into a system.
iv) Selecting pilot projects for wider development of reuse.
v) Identifying reuse metrics.

Poulin *et al*. (1993) presents a set of metrics used by IBM to estimate the efforts saved by reuse. The study suggests the potential benefits against the expenditures of time and resources required to identify and integrate reusable software into a product. Study assumes the cost as the set of data elements like Shipped Source Instructions (*SSI*), Changed Source Instructions (*CSI*), Reused source Instructions (*RSI*) etc. It proposes several reusability metrics in terms of cost and productivity like Reuse Cost Avoidance, Reuse Value Added and Additional Development Cost, which can be used significantly for business applications. These metrics are designed by considering the source code of the component and, therefore, cannot be applied to black-box components.

Cho *et al.* (2001) proposes a set of metrics for measuring various aspects of software components like complexity, customizability and reusability. The work considers two approaches to measure the reusability of a component. The first is a metric, Component Reusability (*CR*), which is calculated by dividing sum of interface methods providing commonality functions in a domain to the sum of total interface methods. The

second approach is a metric, called Component Reusability level (*CRL*) to measure particular component's reuse level per application in a component-based software development. However, the proposed metrics are based on lines of codes and can only be used at design time for components.

Washizaki *et al.* (2003) discussed the importance of reusability of components in order to realize the reuse of components effectively and proposed a component reusability model for black-box components from the viewpoint of component users or application developers. The model identified factors affecting reusability on the basis of an analysis of the activities carried out when reusing a black-box component. The factors considered are:

- Understanding the functionality of the component
- Adapting the component to the specific functional requirements of the new system
- Porting the component to a new environment

Authors (Washizaki *et al.*, 2003) also proposed several metrics related to these factors, namely, Existence of Meta-Information (*EMI*), Rate of Component's Observability (*RCO*), Rate of Component's Customizability (*RCC*), Self-completeness of Component's Return Value (*SCCr*) and Self-completeness of Component's Parameter (*SCCp*). *EMI* and *RCO* metrics indicate that high value of readability will help user to understand the behavior of a component from outside the component. High value of *RCC* metric indicates the high level of customizability of component as per the user's requirement and thus leading to high adaptability. High values of *SCCr* and *SCCp* will lead to self completeness of a component and thus lead to high portability of the component.

This work also performs an empirical evaluation of these metrics on various Java Bean components and set confidence intervals for these metrics. It also establishes a relationship among these proposed metrics. These metrics are applied to only a small number of Java Bean components and thus need to be validated for other component technologies like .NET, ActiveX and others. Though, this work considers some of the important factors like customizability and understandability, and discusses their impact

on reusability of components, it ignores other factors like understanding the component from outside, which may also make important contribution towards the reusability.

Boxall and Araban (2004) considered that understandability of the component affects the level of reuse. Understandability of a component can be made through its interface properties. Authors proposed some metrics for better understanding of the component interfaces by considering the size of interface, argument count, argument repetition scale and others. However, the proposed approach does not consider the other aspects for interface, such as, complexities of the arguments and the return types. Also, empirical validation of the work is still unexplored.

Several other researchers also considered similar factors for estimating reusability. Like, Rotaru *et al.* (2005) considered adaptability, compose-ability and complexity of a component to describe its reusability. Mili *et al.* (1995) considered two aspects, usability and usefulness while *REBOOT* (Reuse Based on Object-Oriented Techniques) proposed by Sindre *et al.* (1995) considered factors, namely, portability, flexibility, understandability and confidence to assess the reusability. In this present study, we have taken account into the understanding of the components as well, along with other important factors required for estimating the reusability of software components.

4.3 PROPOSED METHODOLOGY FOR ESTIMATING THE REUSABILITY FOR COMPONENTS

In the present work, we have identified factors contributing to reusability and then proposed Artificial Neural Network (ANN) model for estimating it for software components. Neural networks have been established to be an effective tool for pattern classification and clustering (Haykin, 2003; Mayrhauser *et al.*, 1995). Neural network has been chosen because:

- ANN has pattern classification ability and is adaptive in nature.
- ANN adjusts the complexity of the network with the complexity of the problem, therefore, gives better results than other analytical models.

- In software industry, the software personnel keep on changing at fast rate. So there is vital need to collect and automate the expertise knowledge. Expertise can help in training the network and after training, it can automate outputs.

The major advantage of ANN is that it learns by examples and has the ability to generalize from its training data to other data. By learning from noisy and incomplete training data, it can produce correct outputs. ANN is relatively inexpensive to build and train (Hudson, 2003).

By and large, the neural network models have been used for engineering predictions, economic predictions, data mining and medical diagnoses (Widrow *et al.*, 1994). However, adoption of neural networks to software engineering paradigm is relatively slow, especially in component-based development. The major studies reporting the application of neural networks to software development have been carried out in the prediction of maintenance efforts and overall development efforts. Keeping this in view, we adopted this approach to predict the reusability for software components.

4.3.1 Artificial Neural Network

An artificial neural network is characterized by its architecture, its learning algorithms and its activation functions. Learning in the present context may be defined as a change in connection weight values that result in the capture of information that can later be recalled. Generally, the initial weights for the network prior to training are set to random values with in a predefined range. There are broadly two paradigms of neural learning algorithms, namely, supervised and unsupervised.

Supervised learning is the most common type of learning in neural networks. It requires many samples to serve as exemplars. Each sample of this training set contains input values with corresponding desired output values (or target values). Then the network will attempt to compute the desired output from the set of given inputs of each sample by minimizing the error of the model output to the desired output. It attempts to do this by continuously adjusting the weights of its connection through an iterative learning process called training (Beizer, 1990; Mayrhauser *et al*., 1995). The supervised learning approach is followed in the present study.

Unsupervised learning is sometimes called self-supervised learning and requires no explicit output values for training. Each of the sample inputs to the network is assumed to belong to a distinct class. Thus, the process of training consists of allowing the network to uncover these classes.

There are two main advantages when using estimation by artificial neural networks. First, it allows the learning from previous situations and outcomes. Second, it can model a complex set of relationships between the dependent variable (such as, cost , customization or effort) and the independent variables (such as reusability and maintainability). However, there are some shortcomings of this approach also. Neural networks approach may be considered as 'black-box'. Consequently, it is not easy to understand and to explain its process to the users. Moreover, there is no guidelines for the construction of the neural network topologies, number of layers, number of units per layer, initial weights etc.

4.3.2 Neural Network Architecture

Neural network architecture (or network topology) refers to the types of interconnections between neurons. A network is said to be fully connected if the output from a neuron is connected to every other neuron in the next layer. A network with connections that passes outputs in a single direction only to neurons on the next layer is called a feed forward network. A feedback network allows its outputs to be inputs to preceding layers. Networks that work with closed loops are known as recurrent networks. Feed forward networks are faster than feedback networks as they require a single pass only to obtain a solution. The former networks are most commonly used for prediction problems (Mukherjee and Deshpande, 1995).

An elementary neuron with R inputs is shown in Figure 4.1. Each input is weighted with an appropriate w. The sum of the weighted inputs and the bias forms the input to the transfer function f. Neurons may use any differentiable transfer function f to generate their output.

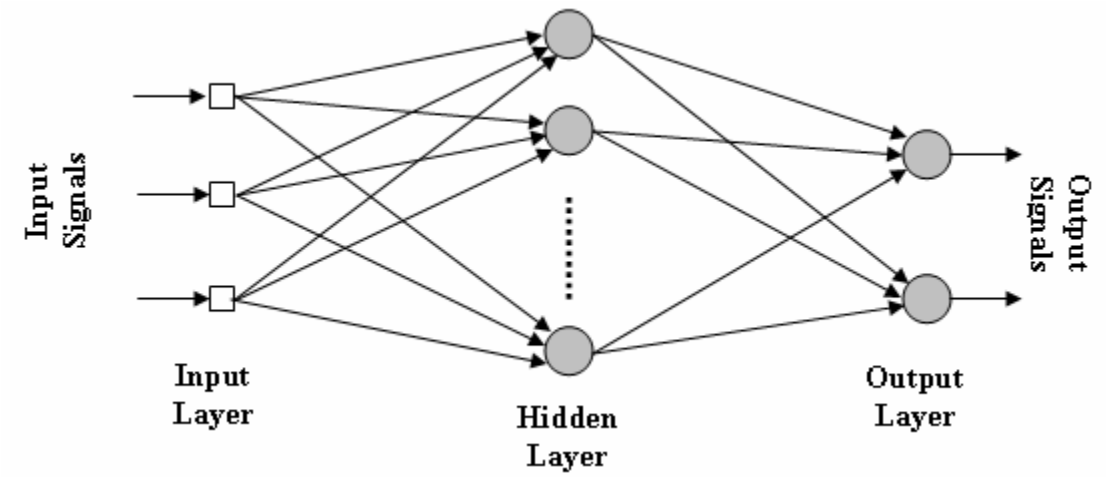


Fig. 4.1: General Architecture of Artificial Neural Network

4.3.3 Steps for Designing a Neural Network

The process to devise a neural network model involves the following steps

- The data to be used should be defined and presented to the neural network as a pattern of input data with the desired outcome or target. Then data are selected to be either in the training set or test set. Training set is used in learning process in developing the model, while test set is used to validate the model for its predictive ability and when to stop the training of the neural network.
- The neural network structure is defined by selecting the number of hidden layers to be constructed and the number of neurons for each hidden layer. All the neural network internal parameters are set before starting the training process.
- Now, the training process is started. It involves the computation of the output using the input data and the weights. A learning algorithm (such as back-propagation) is used to 'train' the neural network by adjusting its weights to minimize the difference between the current neural network output and the desired output.
- Finally, an evaluation process is carried out in order to determine if the neural network has 'learned' to solve the task at hand. This evaluation process may involve periodically halting the training process and testing its performance until an acceptable accuracy is achieved. When an acceptable level of accuracy is obtained, the neural network is then deemed to have been trained and is ready to

be utilized.

- As there are no fixed rules in determining the neural network structure or its parameter values, a large number of neural networks may have to be constructed with different structures and parameters before determining an acceptable model. Determining when the training process needs to be halted is of vital significance to obtain a good model.

4.3.4 Learning Algorithms

A prescribed set of well-defined rules for the solution of a learning problem is called learning algorithm. Several learning algorithms exist for neural networks learning. In this study, back-propagation neural networks (BPNNs) supervised algorithm is used. Backpropagation is the generalization of the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding target vectors are used to train a network until it can approximate a function and associate input vectors with specific output vectors.

Properly trained backpropagation networks tend to give reasonable answers when presented with inputs that they have never seen. Typically, a new input leads to an output similar to the correct output for input vectors used in training, that are similar to the new input being presented. This generalization property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all possible input/output pairs.

The network in the present study is trained by using *Trainlm* and *Trainbr* functions separately of Feed Forward Back Propagation algorithm and then result is compared to get the best output. *Trainlm* is a network training function which is based on Levenberg-Marquardt optimization. *Trainlm* is often the fastest backpropagation algorithm available in MatLab, although it does require more memory than other algorithms. *Trainbr* is also a network training function that updates the weight and bias values according to Levenberg-Marquardt optimization. It minimizes a combination of squared errors and weights, and then determines the correct combination so as to produce a network that generalizes. We have used the linear transfer function *Tansig* for the experimentation.

4.4 EXPERIMENTATION

Component is developed by keeping in mind the reusability aspect. As it has to be used in multiple applications, it should be highly portable and flexible enough to be customized as per new requirements. In other words, reusability of a component depends on the fact that how much customization can be implemented on that. Complex interface of the component will lead towards high efforts in customizing it. Therefore, a reusable component must be developed with simple interfaces. Also, good quality documentation has to be prepared for these components so that application developers, who are using these components, will have thorough understanding of these components. In summary, we identified the following four factors on which the reusability of the component may depend:

i) Customizability
ii) Interface Complexity
iii) Understandability
iv) Portability

i) Customizability

Customizability is defined as the ability to modify a component as per the application requirement. Better customizability will lead to a component with better reusability in applications and thus help in maintaining the component in the later phases. It may be measured on the basis of writable properties available in the component. Writable properties in Java Bean components may be recognized by setter methods. The following formula is used to evaluate this metric (Washizaki, 2003):

$$Customizability = \frac{No.\ of\ Set\ Methods}{Total\ number\ of\ Properties} \qquad (4.1)$$

By using this metric, one can measure that how much an interface method can be customized. Therefore, it may be used to measure the reusability of the component. Customizability of a component may vary from 0 to 1.

ii) Interface Complexity

Components are black box in nature. The source code of these components is not available. Application may interact with these components only through their well-defined interfaces. Interface acts as a primary source for understanding, use and implementation and finally maintenance for the component. Therefore, the complexity of these interfaces plays a lead role while measuring the overall complexity of the component. Complex interfaces will lead to the high efforts for understanding and customizing the components. Therefore for better reusability, interface complexity should be as low as possible. We use the methodology discussed in Chapter 2 for measuring the interface complexity of software components.

iii) Understandability

As the source code of the component is not available to the application developer, documentation is the only source from where he/she can understand the component. Documentation provides the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component.

Here, the term documentation of component refers to component manuals, demos, help system, and marketing information. Functional elements may be referred as the interfaces, operations, or events that a component may support or require from other components to achieve its functionality, i.e., to implement its services. A good quality document must include functional description, installation details, syst em administrator's guide, system reference manuals etc. It may also require non-functional details, like performance, security issues, and previous maintenance activities, if any. It will help in understanding the component and can be reused/integrated easily in the final system. It will also help in implementing the maintenance activities with less effort. For present work, we categorize documentation quality into very low, low, medium, high and very high categories.

iv) Portability

It is the ability of a component to be transferred from one environment to another with little modification, if required. It is typically concerned with reuse of component on new platforms. The component should be easily and quickly portable to specified new environments if and when necessary, with minimized porting efforts and schedules. For

better reusability, component should be highly portable, means; it should be supported by several platforms. Here, for the proposed work, portability is divided into very low, low, medium, high and very high categories.

The goal of our work is to develop a tool for estimating reusability of the software component. We consider that reusability is a measure of factors mentioned above. The values of these individual factors can be measured by using the appropriate metrics. For our study, interface complexity metric is measured by the methodology discussed in Chapter 2. Customizability metric is the ratio of writable properties to the total number of properties. Documentation and portability can be classified from very low to very high categories.

4.4.1 Development of Neural Network Model to Measure Reusability

The training set consisted of 40 exemplars of Java Bean components, collected from *www.jars.com* and *www.elegantjbeans.com*. The components were divided into 5, 8, 13, 10, 4 cases of very low, low, medium, high and very high categories respectively. Further, 12 exemplars were selected for test set, to validate the network. The values of all four factors were measured by using the above discussed metrics. All these metrics are normalized by using *Min-Max* normalization. *Min-Max* normalization is a linear transformation on the original data. It transforms the original input range into a new data range (typically -1 to +1 range). Suppose that *minA* and *maxA* are the minimum and maximum values of an attribute *A*. It maps value *v* of *A* to *v'* in the target range -1 (min_{target}) to 1 (max_{target}) by using the formula given in Equation 4.2.

$$v' = (max_{target} - min_{target}) * \left[\frac{v - minA}{maxA - minA} \right] + min_{target} \qquad (4.2)$$

The network is investigated empirically containing different numbers of neurons, i.e., 5, 10, 12, 15, 18, 20, 22, and 25 in hidden layer(s).

Network is trained by adjusting different weights of neurons and hidden layers, so that the difference between desired output from the network and actual output is minimized. The network learns by finding a vector of connection weights that minimizes the desired error on the training data set. We used training functions *trainlm* and *trainbr* separately and compared the results. The adaptation learning function selected for this

experiment was *learngdm*. We used the transfer functions, '*tan-sigmoid*' in both the layers, which converts the input into net output. Each of the networks was trained with 1000 epochs, keeping the goal as 0. Other parameters are set to their default values.

In order to establish the success and sufficiency of supervised training for the neural network models, it is necessary to have some quantitative measure of learning. Root mean squared error (*RMSE*) is an adequate and commonly used error measure. *RMSE* is a useful measure of how close a network is getting its predictions to its target output values. For successful training, *RMSE* will decrease significantly in the initial stages of training and converge after a sufficient number of iterations have been completed. For this study, it is calculated as:

$$RMSE = \sqrt{\frac{\sum (ActualValue - EstimatedValue)^2}{n}} \qquad (4.3)$$

where *n* is the number of observations

Experiments were conducted with an ensemble of networks, starting from network with 5 hidden neurons and going up to 25 neurons to find out which of these networks gives the best performance, keeping the output layer neurons constant in each of the networks i.e. 1 representing one of the five target values. After the training, simulation was done on the network. The network is tested against test set, consisting of 12 exemplars from all five categories. Simulation was done with both the training functions to see the difference in performance. The optimum performance of the BPNN model is found to be by *trainlm* function at 15 neurons with *RMSE* equals to 0.1648. The results obtained with different neurons are shown in Table 4.1

Therefore, we can say that the proposed ANN based approach is able to predict the reusability of the software components and can be used to choose the better reusable component to be integrated in the system.

Reusability		
Neurons in the Hidden Layer	*RMSE* for *Trainlm* function	*RMSE* for *Trainbr* function
5	0.1929	0.1793
10	0.1821	0.1782
12	0.1732	0.1763
15	**0.1648**	0.1701
18	0.1783	0.1783
20	0.2114	0.1783
22	0.1883	0.1784
25	0.1826	0.1801

Table 4.1: Accuracy for the Network

4.5 CONCLUSION

Reusability is the most important criteria for selecting a component for component-based systems. A highly reusable component will help in better understanding and low maintenance efforts for the application. Therefore, it is necessary to estimate the reusability of the component, before integrating it into the system. Present chapter adopts artificial neural network based approach to estimate the reusability of component. Network is trained on training data by considering different number of hidden neurons for two training functions, namely, *trainlm* and *trainbr*, to get the best results. This network is further validated by applying the proposed approach on test data. Results obtained show that network is able to predict the reusability of these components with an acceptable accuracy. However, it is also possible to train the network by using different combinations of other training functions, transfer functions, and number of hidden layers etc. Also, it is possible to explore alternative neural network models like cascade correlation model that dynamically build the neural network architecture model. In the present study, data used to train the network is limited to 40 components. More number of components for training may produce better results.

Chapter 5

PREDICTING MAINTAINABILITY

5.1 INTRODUCTION

Software maintenance is a very broad activity in software development that includes error corrections, enhancement of capabilities, optimization, deletion of obsolete capabilities and so on. Maintenance includes all changes to the product, once the client has agreed that it satisfied the specified document. Maintenance may be corrective, adaptive or perfective. Corrective maintenance consists of the removal of residual faults. Adaptive maintenance changes functionality to meet another requirement in place of an existing requirement due to changes in environment, while perfective maintenance refers to enhancement and optimization. Corrective maintenance is important and critical because system will remain idle during this time whereas other types of maintenance activities can go in parallel with the software operation.

The maintenance of existing software can account for 70% of the total efforts put-in application development (Pressman, 2005). It is also estimated that this may go up to 80%, if the approaches used in the development are not improved. A survey by Lientz and Swanson (2000) discovered that about 65% of maintenance was concerned with implementing new requirements, 18% with changing the system to adapt it to a new operating environment and 17% to correct system faults. Because change is inevitable, mechanism must be developed for evaluating, controlling and making modifications. Therefore a different and effective approach is needed to reduce the overall maintenance efforts.

Maintenance of CBS may require several different activities than normal applications, such as, upgrading the functionality of black-box components (for which code may not be available), replacement of older version components with the new ones

The major findings of this Chapter have been published in ACM SIGSOFT Software Engineering Notes, Vol. 32, Issue 5, September 2007, pp: 1-5.

for better and improved functionality, tracing the problem of compatibility between the new components with system, and so on.

Present chapter discusses various issues and challenges, related with the maintainability of CBS. It studies maintainability aspect of ISO 9126 quality model and proposes to introduce trackability aspect under that. Trackability is expected to ease the overall maintenance activity. The present work proposes a Fuzzy Logic based approach to estimate the maintainability of CBS. We identified the factors influencing maintainability and then categorized them into different fuzzy sets. Classroom projects are considered to estimate and validate the proposed maintainability model.

5.2 MAINTAINABILITY

Maintainability is a prediction of the ease with which a system can evolve from its current state to its future desired state. It is termed as the most difficult and costliest activity due to it's inherently involvement in making predictions about the future.

Maintainability is defined as (IEEE, 1998):

"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment"

Another definition is (ISO, 2001):

"The capability of the software product to be modified". Modifications may include corrections, improvements or adoptions of the software to changes in environment, and in requirements and functional specifications.

5.2.1 Maintainability Challenges for CBS

In traditional software systems, we relate maintainability with the term MTTR (mean time to repair), which should be as low as possible for good maintainability. However, this traditional measure, which relies on source code visibility, is insufficient to contend the maintenance demands of CBD. Component-based systems consist of black-boxes; the maintainers of these systems do not have access to the source code of the components. Here the main difficulty is to identify whether the problem is in the component itself or in the system or may be due to the interaction between the two. Moreover, if system needs to be upgraded for some advanced additional features, the

compatibility of the new system with the existing components may vanish. In this case, components will also need the upgradation or may require installation of new components with the desired compatibility. Therefore, we may have to consider both components and component-based systems separately while dealing with the maintenance issues. Voas (1998) discusses several aspects of maintainability of CBS, which includes difficulties in maintenance activities due to frozen functionality, incompatible upgrades, defective and complex components or defective middleware such as wrappers or the glue code. The work also suggests some guidelines for maintenance activities, which are:

- Use CBD approach only for large and complex systems where reusability may be a great concern. For small systems, CBS may not be the right choice.
- For better understanding of the component, keep detailed documentation for each component.
- In CBS, suitable component selection with better performance and functionality is the main challenging job. Component repositories (within and outside the organization) should have the adequate number of suitable components so that developer can pick the best according to its requirements.
- If multiple applications share a component, but cannot tolerate changes any one of these applications need, keep separate similar components in the repository.

Voas's study on maintenance of CBS, though theoretical, but is a step ahead towards our understanding and measuring maintainability for software components. The major challenges with the maintenance of components and component-based systems are: (Judith and Audrey, 1998):

- Upgradation/Modification in component functionality may not be compatible with other existing components as well as the system where the components are used, thus necessitating updating of these components and finally software system also.
- Upgradation/replacement of component may also lead to the incompatibilities in integration process, which may require a substantial modification in interface code also.

- The new component may be incompatible with the hardware environment of the application. Also, the modified component may consume time or memory, which may not be acceptable by the system due to performance requirements. This , in turn, may require necessary modifications in the system to adjust the changes.
- If the modified component requires changes in tools used and languages, it may not be supported by the software system. For example, a component designed by using Microsoft technology (say VB .NET or ASP .NET component) may not work perfectly if used in an application developed in Java technology or may require additional plug-ins for proper functioning.
- Modified components may require dealing with the modifications in security concerns for the software system.

To overcome all these challenges, component developers need to develop components, which are visible, extensible and easily integrated into a wide spectrum of systems. The more open components, easier for maintainers to monitor, manage , extend, replace, test and integrate.

5.2.2 Quality Models and Components Maintainability

There are several quality models (McCall and Joseph, 1978; Boehm *et al.*, 1976; Grady, 1992; Dromey, 1995; Sedigh *et al.*, 2001b; ISO, 2001) and others proposed by researchers for general software applications, but none of them , addresses the concerns of component-based systems directly in detail, though, maintainability characteristic is supported by almost all. Out of all these, ISO 9126 caters the need of software industry to standardize the evaluation of software products using quality models in more promising and suitable way. We concentrate on ISO 9126 model to further investigate the maintainability characteristic and its sub-characteristics for CBS. As this model is a generic model for any software product, we need to restrict it to component-based systems. The model defines maintainability in terms of four sub-characteristics: Customizability, Stability, Analyzability, and Testability.

i) Customizability

It is the capability of the software product to enable a specified modification to be implemented. Although the user (application developer) may not have the possibility of

making modifications on the component, he can create a wrapper around it. Wrapping can be made to control the component's input data as well as to filter the output generated by the component. For example, in Java Bean components, customization can be performed by using *set* methods.

ii) Analyzability

Analyzability may be defined as the capability needed for diagnosis of deficiencies or causes of failures or for identification of parts to be modified. In case of components, it is very rare to develop some functionality for the auto-analysis purpose or which identifies the parts of the components to be modified in the later stages (Voas, 1998). Therefore, it may be excluded from the maintainability characteristic.

iii) Stability

Stability is the capability to avoid unexpected effects from modifications to the software. In case of component-based systems, it may be defined as the degree to which software is composed of discrete components such that a change to one component has the minimal or zero impact on the other components or the system. Therefore, for better stability, dependencies among components should be as low as possible.

iv) Testability

It is the capability of the component to be tested with simplified test operations and reduced test cost. For better testability, it is important to test how easy it is to observe the component in terms of its behavior, input parameters and outputs. To achieve all these, proper test suites and test cases have to be provided to test the component thoroughly before being used in the system. Here the maintainers have to keep in mind also the environment or the platforms in which the component has to be tested for the compatibility purpose.

5.2.3 New Sub-Characteristic of Maintainability

Apart from the above mentioned changes in maintainability characteristic, we propose to add a new maintainability sub-characteristic, called trackability, described as:

Trackability

When reconfiguring a component for changed/improved functionality, maintainers must perform a full cycle of product evaluation, integration and testing. Even

if the new release of a component claims to be functionally backward compatible with an older version, there may undoubtedly be differences such as resource usage, performance, and other non-functional requirements (Vigedar and Kark, 2006). Therefore, for better understanding of the previously performed maintenance tasks, it is very necessary to keep track of all the parameters of the older version so that these details can be com pared with the enhanced or changed parameters. Thus, maintenance activities may be extended to include provisions for keeping a proper track of various system properties during the maintenance activities. These system properties may include tracking system performance or resource utilization, before and after any maintenance activity, like replacement of an old component with a new version with some added/modified functionality. This may also include the possible security violations due to some maintenance activities. This section provides a detailed study of the non-functional activities performed during maintenance of the CBS.

- **Performance Tracking:** It is observed that component vendors usually do not provide application developers with any performance info rmation. Hence, system testers and integration engineers must spend a lot of efforts to identify the performance problems and the components that cause the problems. Therefore, we need to check the performance aspects of the system after any change is made. These performance aspects can include various sub attributes like Resource Utilization, (CPU Cycle, Execution Time), Audit logs, Average/Min/Max Speed, Scalability aspect (number of users, a system can scale to without sensible decrease in response time) etc. The information recorded before and after maintenance can be used for any comparative analysis, especially in the case when a component is replaced by a new one with changed development environment (programming language or operating system).
- **Visibility Tracking:** It is the tracking of public visible data and data states of components, needs to be changed or replaced. It includes the recording of properties, methods/interface methods, arguments, return types, exceptions, annotations etc. It may also rec ord the GUI events available in the components.

- **Portability:** If the replaced component is developed under different operating system environment or by using different programming language then we need to address the issues of compatibility also. Also the new component may need different glue/integration code to be embedded in the system, may be due to its changed environment or functionality. This information needs to be stored in the maintenance documentation for future use.
- **Confidentiality and Integrity:** Confidentiality is defined as a measure of the absence of unauthorized disclosure of information while integrity is defined as the absence of improper system state alterations. This information can be tracked at the application level for secure and authorized access and not at the component level. We need to track whether there is any violation of these two aspects in the application or in the component after the maintenance phase.
- **Configuration Management and Licensing Issues:** If we need to upgrade the components used in the system for better functionality or for other aspects, then we need to track the versions and licensing issues of these components for backward compatibility, which may arise at any later stage.
- **Quality affected:** Any effect on some of the other quality criteria after maintenance like reusability, customizability, complexity etc. must also have to be recorded for better maintenance. For this, we may use various metrics available in the literature for measuring these aspects.
- **Other Issues:** Tracking may also involve the other aspects like documentation, total cost for the present maintenance activity, conformance to various standards, Installability etc.

Tracking, not only will validate any improvement efforts, but also once an information process is presumed to be stable, tracking may provide insight into maintaining statistical control. This, in effect, will ease the overall maintenance process.

5.3 ESTIMATING MAINTAINABILITY

Quantitative measurement of an operational system's maintainability is desirable both as an instantaneous measure and as a predictor of maintainability over the time.

Efforts to estimate and track maintainability are intended to help reduce or reverse a system's tendency towards degraded integrity and to indicate when it becomes cheaper and/or less risky to rewrite the code than to change it.

Khairuddin and Elizabeth (1996) proposed a maintainability model by considering two separate qualities; reparability and evolvability. Reparability involves corrective maintenance, while evolvability involves preventive and adaptive maintenance. The model describes software related factors affecting maintainability of software components. These factors include Modularity, Readability, Programming Language, Standardization, Level of Validation and Testing, Complexity and Traceability.

Arsanjani *et al.* (2002) addresses the issue of software maintenance of component based systems by identifying encapsulating and externalizing the variations around design decisions. The approach is based on the notation of Enterprise Component (EC). EC is defined as an architecture patterns that provide a uniform mechanism for management of component boundaries between systems. The process includes the identification of requirements for the system and components, formalizing and abstracting them into a domain-specific language's grammar. This approach enables a highly re-configurable architectural style to help build and maintain reusable components that are responsive and resilient to changing requirements .

UML is a language for specifying, constructing, visualizing and documenting artifacts of software-intensive systems. It can be used to represent key parts of the internal structures of the components without relying on the source code and is now the industry standard for software modeling. Wu and Offutt (2003) present a new UML-notation based approach for maintaining CBS. Authors used a static analysis to identify the interface events and dependence relationship that would be affected by the modification in the maintenance activity. The work discusses UML-based infrastructure and corresponding regression testing for corrective maintenance activities. It also provides a UML-based framework to evaluate the similarities among old and new components. The given approach seems to be quite reasonable but needs further study in real environment to check the validity and assumptions made.

Ardimento *et al.* (2004) reports the results of empirical study aimed at understanding how characterization of components affect the maintenance effort of the component- based systems. They have made the assessment that (i) functionality of each component should be as concentrated as possible over a single aspect of the application domain; (ii) the training time offered by the component's producer usually indicates the complexity of understanding it and if a component is difficult to understand, then it is also difficult to maintain; and (iii) a deep knowledge of the component is necessary for the organization before its adoption, therefore, a trial usage of components is advised before the final decision about their adoption.

Kajko-Mattsson *et al*. (2006) discussed the problems faced by software community towards maintenance and elaborate some concepts for proposing a maintainability model. This model considered both, the product aspects as well as process aspects related with the maintenance.

Artificial Neural Network based approach is adopted by Singh *et al*. (2004) to predict the maintainability of the systems. They considered Readability of Source Code (*RSC*), Documentation Quality, Understanding of Software (*UOS*) and Average Cyclomatic Complexity (*ACC*) as independent variables to measure Maintainability, which is considered as dependent variable. These variables were used to train the Artificial Neural Network by using MatLab. The results obtained were quite appreciable with the prediction quality of 91.42%.

Similar approach is adopted by Aggarwal *et al*. (2006) to predict the maintainability of the object-oriented systems. They considered principal components of eight OO metrics as independent variables. These include Lack of Cohesion (*LCOM*), Number of Children (*NOC*), Depth of Inheritance (*DIT*) and others. Maintainability was considered as dependant variable. Results obtained by training the network using back propagation algorithm show that independent variables chosen for the study were able to predict the maintenance efforts with a mean absolute relative error (*MARE*) of 0.265.

Shukla and Mishra (2008) also used Neural Network based approach to estimate software maintenance efforts. They chose 14 factors as cost drivers for their study and conducted the experiment by taking various options of number of hidden layers and number of hidden nodes. Input data selected for training was 60% of the total, while 20%

each were used for validation and testing. *MRE* obtained from the experiment was around 5%. Results concluded that neural network was able to successfully model the maintenance effort.

Aggarwal *et al.* (2005) used Fuzzy model to measure the maintainability of the software system. The authors considered four factors affecting maintainability, namely, average number of live variable, average life span of variables, average cyclomatic complexity and the comments ratio. All inputs are classified into fuzzy sets viz. low, medium and high, while maintainability is classified as very good, good, average, poor and very poor. All the inputs and outputs are fuzzified and in total 81 rules are proposed for the model. Model is validated against the software projects developed by undergraduate engineering students. However, the model is not validated on real life complex projects.

5.4 PROPOSED FUZZY-BASED APPROACH FOR ESTIMATING MAINTAINABILITY FOR CBS

In this chapter, we proposed a Fuzzy Logic based approach for estimating maintainability for component-based systems.

It is often impossible to estimate software quality attributes directly. For example, attributes (say, maintainability) are affected by many different factors, and there is no straightforward method to measure them. To estimate maintainability of CBS, one needs to establish a relationship of the factors with maintainability to achieve the desired goal.

As discussed earlier, maintenance of CBS may require a change either in the component or in the system or in the integration code. Emphasis in the present work is to estimate maintainability, if the changes require customization/replacement of the component or integration code has to be modified. Following factors have been identified, which will influence maintainability of CBS:

i) Interaction Complexity among components
ii) Reusability of the component
iii) Testability
iv) Understandability
v) Trackability

Interaction Complexity among components and Reusability of a component have been discussed in Chapter 3 and Chapter 4 of this thesis respectively. Testability is discussed under section 5.2 of this Chapter. Documentation of the component is considered for understanding the behavior of the component, as source code of these components are not available to the maintainers. Documentation provides the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system. A good quality document must include functional description, installation details, system administrator's guide, system reference manuals etc. Finally, trackability may be estimated on the basis of initial information stored for performance, portability, conformance, security issues and other non-functional aspects. It will help in understanding and implementing the maintenance activities with less effort.

5.4.1 Fuzzy Logic

Fuzzy Logic is a mathematical tool for dealing with uncertainty and also it provides a technique to deal with imprecision and information granularity (Sivanandam, 2007). Fuzzy logic offers a particularly convenient way to generate a mapping between input and output spaces by using natural expressions (Zadeh, 2002). In direct contrast to neural networks, which take training data and generate opaque models, fuzzy logic is based on if-then rules, which are designed by considering the opinion of experts from that domain. It has been found that the most accurate prediction models are based on analogy and experts opinion. Expert-based estimation was also found to be better than all regression-based models (Musilek *et al.,* 2000). Henceforth the use of fuzzy logic in maintenance prediction is desirable since expert knowledge can be incorporated into the fuzzy maintenance prediction models.

Major advantage of this approach is that it is less dependent on historical data. Fuzzy logic models can be constructed without any data or with little data (MacDonell *et al.*, 1999 and Ryder, 1998). This makes fuzzy logic superior over data-driven model building approaches such as neural network, regression and case-based reasoning. In addition, fuzzy logic models can adapt to new environment when data become available (Sailu *et al*., 2004).

5.4.2 Implementation of Fuzzy Logic

Implementing a fuzzy system requires that the different categories of the different inputs be represented by fuzzy sets which, in turn, is represented by membership functions. The domain of membership function is fixed, usually the set of real numbers, and whose range is the span of positive numbers in the closed interval [0, 1]. There are total 11 membership functions available in MatLab. We considered Triangular Membership Functions (TMF) for our problem, because of its simplicity and heavy use by researchers for prediction models (Zadeh, 2002). It is a three-point function, defined by minimum α, maximum β and modal value m i.e. TMF (α, m, β), where ($\alpha \leq m \leq \beta$). This process is known as fuzzification. These membership functions are then processed in fuzzy domain by inference engine based on knowledge base (rule base and data base) supplied by domain experts and finally the process of converting back fuzzy numbers into single numerical values is called defuzzification (Aggarwal *et al.*, 2005b).

5.4.3 Experimental Design

We propose that maintainability of component-based system is a measure of five factors mentioned above. These combined factors can be used to measure the maintainability, as it cannot be measured directly. The proposed fuzzy logic based model considers all five factors as inputs and provides a crisp value of maintainability using the Rule Base. All inputs can be classified into fuzzy sets viz. Low, Medium and High. The output maintainability is classified as Very High, High, Medium, Low and Very Low. All possible combinations (3^5 i.e. 243) of inputs are considered to design the rule base. Each rule corresponds to one of the five outputs based on the expert opinions. Some of the proposed rules are shown as:

- If Interaction Complexity among components is High, Reusability of component is Low, Testability is Low Understandability is Low and Trackability is Low then it is very difficult to maintain the system i.e. Maintainability will be Very Low.
- If Interaction Complexity among components is High, Reusability of component is Low, Testability is Low, Understandability is Low and Trackability is Medium then Maintainability still will be Very Low.

- If Interaction Complexity among components is High, Reusability of component is Low, Testability is Low, Understandability is Medium and Trackability is Medium then Maintainability will be Low.
- If Interaction Complexity among components is High, Reusability of component is Medium, Testability is Medium, Understandability is Medium and Trackability is Medium then Maintainability will be Medium.
- If Interaction Complexity among components is Medium, Reusability of component is Medium, Testability is Medium, Understandability is Medium and Trackability is Medium then Maintainability will be Medium.
- If Interaction Complexity among components is Low, Reusability of component is High, Testability is High, Understandability is Medium and Trackability is Medium then Maintainability will be High.
- If Interaction Complexity among components is Low, Reusability of component is High, Testability is High, Understandability is Medium and Trackability is Medium then Maintainability will be High.
- If Interaction Complexity among components is Low, Reusability of component is High, Testability is High, Understandability is High and Trackability is High then Maintainability will be Very High.

All 243 rules are inserted into the proposed model and a rule base is created. Depending on a particular set of inputs, a rule is fired. Using the rule viewer, output i.e. maintainability is observed for a particular set of inputs using the MatLab Fuzzy tool box. Table 5.1 shows the values of various parameters set for inputs, outputs for fuzzification process.

System	Name='maintainability' Type='mamdani' Version=2.0 NumInputs=5 NumOutputs=1 NumRules=243 AndMethod='min' OrMethod='max'

	ImpMethod='min' AggMethod='max' DefuzzMethod='centroid'
Input1	Name='Reusability' Range=[0 1] NumMFs=3 MF1='Low':'trimf',[0 0.2 0.35] MF2='medium':'trimf',[0.3 0.5 0.68] MF3='high':'trimf',[0.65 0.8 1.0]
Input2	Name='Intercation_Complexity' Range=[0 1] NumMFs=3 MF1='low':'trimf',[0 0.2 0.35] MF2='medium':'trimf',[0.32 0.5 0.68] MF3='high':'trimf',[0.62 0.8 1.0]
Input3	Name='Testability' Range=[0 1] NumMFs=3 MF1='low':'trimf',[0 0.16 0.35] MF2='medium':'trimf',[0.3 0.5 0.68] MF3='high':'trimf',[0.62 0.85 1.0]
Input4	Name='understandability' Range=[0 1] NumMFs=3 MF1='low':'trimf',[0 0.16 0.35] MF2='medium':'trimf',[0.3 0.53 0.68] MF3='high':'trimf',[0.63 0.8 1.0]
Input5	Name='trackability' Range=[0 1] NumMFs=3 MF1='low':'trimf',[0 0.15 0.35] MF2='medium':'trimf',[0.32 0.55 0.66] MF3='high':'trimf',[0.63 0.76 1.0]
Output1	Name='output1' Range=[0 1] NumMFs=5 MF1='Very_Low':'trimf',[0 0.1 0.21] MF2='Low':'trimf',[0.19 0.3 0.42] MF3='Medium':'trimf',[0.38 0.5 0.62] MF4='High':'trimf',[0.59 0.7 0.81] MF5='Very_High':'trimf',[0.78 0.9 1.0]

Table 5.1: Parameter Values for Inputs and Output

Fuzzification of inputs into output, membership functions for reusability, surface view for reusability, interaction complexity and maintainability and rules for defuzzification process are shown in Fig. 5.1, Fig. 5.2, Fig. 5.3 and Fig. 5.4 respectively.

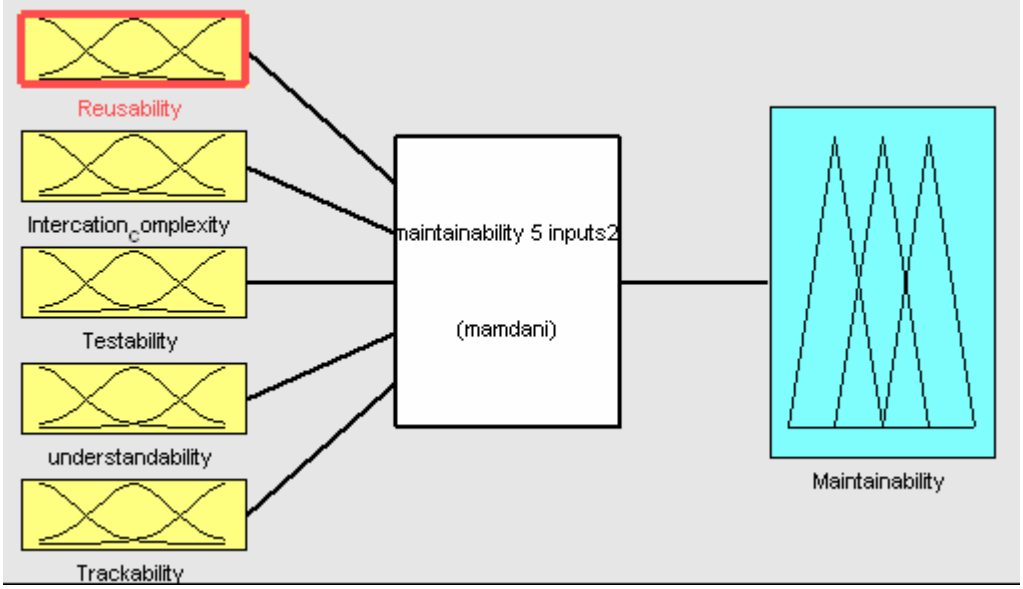


Fig. 5.1: Fuzzification of Inputs into Output (Maintainability)

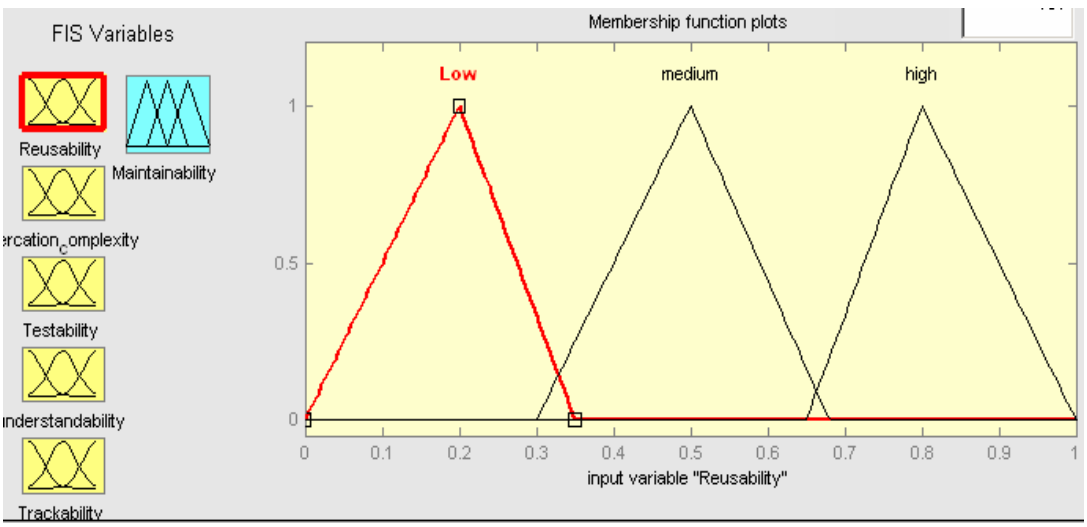


Fig. 5.2: Membership Functions for Reusability

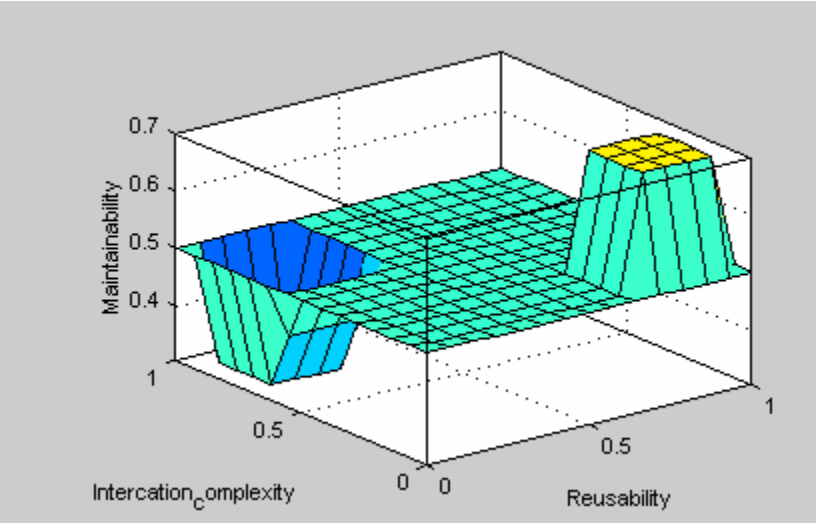


Fig. 5.3: Surface View for Reusability on X-axis, Interaction Complexity on Y-Axis and Maintainability on Z-Axis

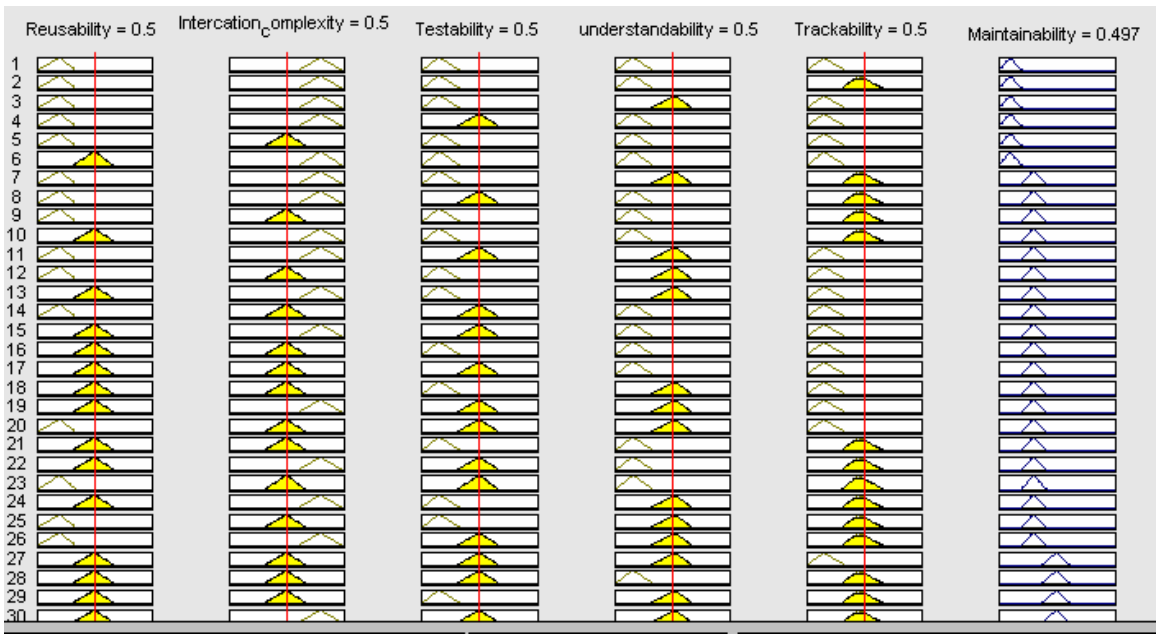


Fig. 5.4: Snapshot of Rule Viewer showing some of the Rules

By using rule viewer, the value of maintainability can be estimated by taking all five values for input factors.

5.4.4 Empirical Evaluation

We considered a class room based project, which is a Billing System for Hotel and is developed in Java. This application consists of three components, namely, Mathematical Calculator, Tax Calculator and Calendar. All these components are already built-in components in JavaBeans and are downloaded from website http://www.jars.com. A good documentation of these components is also available on websites to understand the functionalities of these components. These components are used as black-box components. The students were asked to replace the old version Tax Calculator component with the new upgraded version. Values of all five input factors are measured by using appropriate metrics, for the modifications required in the system. We measured reusability by Neural Network based approach discussed in Chapter 4. Interaction complexity is measured based on the number of dependent components, discussed in Chapter 3. Understandability, Testability and Trackability are measured based on the information available on the website and provided by the developers of these components. After normalizing the values of these factors, defuzzification is applied for these values using the proposed model. The maintainability for these values comes out to be 0.72, which is under High category. It means that the system is highly maintainable and can adopt the changes/modifications in the component .

5.4.5 Validation of the Proposed Model

The proposed methodology is validated by using Analytical Hierarchical Process, discussed in Chapter 6. Two class room projects are considered for the validation purpose. Proposed fuzzy logic based approach is applied to estimate the maintainability for these two projects. For validation, a survey is conducted on software professionals working on maintenance projects in different software organizations. They were asked to give their preferences of maintainability factors discussed earlier on a scale of 1 (never required) to 4 (always required). This data is then analyzed by using AHP. The weight values for all the factors were obtained on a scale of 0 to 1, given in Table 5.2 as:

Interaction Complexity	Reusability	Testability	Understandability	Trackability
0.26	0.18	0.13	0.32	0.11

Table 5.2: Weight Values of Maintainability Factors

The metrics for all the four factors of maintainability were measured for both the projects. These metric values were then multiplied by their corresponding weight values to get the values of overall maintainability for these projects. The values of maintainability, obtained from these two approaches are shown in Table 5.3.

Project	Fuzzy Logic	AHP
P1	0.72	0.66
P2	0.43	0.32

Table 5.3: Maintainability Values by Fuzzy Logic and AHP

From the Table 5.3, it is clear that values obtained from Fuzzy Logic based approach for Maintainability is almost near to that of AHP approach. It indicates that proposed approach may be able to predict the maintainability for CBS and can be used by the application developers.

5.5 CONCLUSION

Maintaining a component-based system may require several different activities than other legacy systems. Due to the use of black-box components in the system, it is difficult to identify whether the maintenance activity is to be performed in the system or in the components (which is very limited due to the non-availability of the source code) or in the integration of components in the system. This chapter discusses various issues and challenges in maintaining component-based systems. It also proposes to improve the maintainability characteristic under ISO 9126 quality model to add trackability sub-characteristic under it. Chapter discusses several approaches in literature for measuring maintainability of object-oriented and component-based systems. It proposes a Fuzzy Logic based model to measure maintainability for CBS. Fuzzy Logic based approach has several advantages over other methods including Neural Network and others. One major advantage is that it may also work without the data. We empirically evaluate the proposed model on a classroom project. The proposed model is able to evaluate the maintainability of the system. The model is validated against two classroom projects by using AHP and results obtained are close in both the approaches. However, it still requires the validation of the proposed model for real-life projects.

Chapter 6

QUALITY MODELS

6.1 INTRODUCTION

Quality is a functional and artistic measurement used, for instance , to specify user satisfaction with a product, or how well the product performs , compared to similar products. In CBSD, the proper search and selection processes of components have become corner-stage for effective and quality-oriented software development. Here developers have to rely on the vendors , from whom he is taking the component to integrate it in the application. This component may no t meet the quality requirements set by the developers and may produce catastrophic results in a typically complex situation. So far, most of the software engineering community has concentrated on the functional aspects of the component, leaving aside the q uality and extra-functional properties. However, this kind of properties deserves special attention, since they are essential in any commercial evaluation process. A poor quality component , selected for an application may cause a great difficulty in managi ng the application from both, functional and behavior point of view. Therefore it is very necessary for CBS developer s to select the best component in terms of functional as well as non -functional by applying appropriate quality evaluation mechanism/model.

Several quality models have been proposed by researchers for software systems which include (McCall and Joseph, 1976; Boehm *et al*., 1978; Grady, 1992; Dromey, 1995; Sedigh *et al.*, 2001b; ISO, 2001) and others. Most of these are generic models and are proposed for general application systems. Out of these models, ISO 9126 is a prominent model which attempts to incorporate the findings of almost all other models . This chapter reviews a number of quality models available and proposes a new quality model for software component. The present work extends ISO 9126 quality model by

The major findings of this Chapter have been published in ACM SIGSOFT Software Engineering Notes, Vol. 33, Issue 6, November 2008, pp: 1-10

tailoring and extending some of its quality characteristics and sub-characteristics. It adds reusability, complexity, trackability and flexibility characteristics which are relevant to components and removes attractiveness, analyzability, which we believe may not be relevant. The proposed work also defines the methodology to evaluate the quality of software component and measure it for a real-life example.

6.2 SOFTWARE QUALITY

The discipline of software quality is a planned and systematic set of activities to ensure that quality is built into the software. It consists of software quality assurance, software quality control, assessment and other aspects. According to the IEEE 610.12 (IEEE, 1990) standard, software quality is a set of attributes of a software system and is defined as:

- The degree to which a system, component, or process meets specified requirements.
- The degree to which a system, component, or process meets customer or user needs or expectations.
- Quality comprises all characteristics and significant features of a product or an activity which relate to the satisfaction of given requirements.

These standards are in existence for a long time and their relevance might be a little too broad. IEEE standard expresses quality in terms of customer expectation. If a customer's expectation is nil, it doesn't mean that a product with nil characteristics is a quality product? So this type of definition may not be appropriate.

Ince (1994) describes the modern view of quality as:

"A high quality product is one which has associated with it a number of quality factors. These could be described in the requirements specification; they could be cultured, in that they are normally associated with the artifact through familiarity of use and through the shared experience of users; or they could be quality factors which the developer regards as important but are not considered by the customer and hence not included in the requirements specification".

Quality is the totality of features and characteristics of a product or a service that bears on its ability to satisfy the given needs, for example, conformance to specifications. It is the degree to which a customer or user perceives that software meets his or her composite expectations (Khosravi and Gueheneuc, 2004).

The evaluation of quality for a software system depends upon the following:

- Quality Model
- Quality characteristics which may further be classified into several sub-characteristics.
- Metrics to measure the attributes of characteristics and sub-characteristics.

In the ISO standard 8402 (ISO, 1994), a software quality model is defined as: "The set of characteristics and the relationships between them which provide the basis for specifying quality requirements and evaluating quality".

Software quality models have been proposed in order to provide many benefits: these can be used as a base to define a commonly agreeable quality framework, which consolidates the different views on quality; they can be tailored to specific contexts; they provided a measurable base to the evaluation of software quality.

6.3 QUALITY MODELS FOR LEGACY SYSTEMS

A quality model establishes a framework to perform some kind of measurement of the specific desirable characteristics that are needed in the final system and perceived by the end user. Here an important assumption of these models is made that internal product characteristics, related to the product development, affect external attributes or quality in use (Losavio *et al.*, 2002). Following section briefly presents some of the widely used and accessed quality models for legacy software applications:

6.3.1 McCall Model

The first quality model was proposed by McCall and Joseph (1976). They presented a Software Quality Factor Framework and classified the quality attributes into three groups shown in Table 6.1.

- Product Operation: refers to the system's ability to be quickly understood, efficiently operated and capable of providing the results required by the user i.e. involving attributes such as correctness, reliability, efficiency, integrity and usability.
- Product Revision: relates to error correction and system adaptation. This aspect is generally considered the costliest part of the software and involves attributes such as maintainability, flexibility and testability.
- Product transition: refers to distributed processing, together with rapidly changing hardware and involves attribute such as portability, reusability and interoperability.

Product Operation Factors	**Product Revision Factors**	**Product Transition Factors**
• Correctness • Reliability • Efficiency • Integrity • Usability	• Maintainability • Flexibility • Testability	• Portability • Reusability • Interoperability

Table 6.1: McCall Quality Model

The major advantage of this model is the relationship created between its quality characteristics; however, the main drawback is that it does not include the functionality aspect of the software product. Also, some of the factors and measurable properties, like traceability and self-documentation among others, are not really definable or even meaningful at an early stage for non-technical stakeholders. It is, therefore, difficult to use this framework to set precise and specific quality requirements. This model is not applicable with respect to the criteria outlined in the IEEE Standard for a software quality metrics methodology for a top-down approach to quality engineering (Cote *et al.,* 2006). It is, therefore, not suited as a foundation for software quality engineering according to the stated premises.

6.3.2 Boehm's Model

The Boehm model (Boehm *et al.*, 1978) is similar to the McCall model in that it represents a hierarchical structure of characteristics, each of which contributes to total quality. Boehm's notion includes users' needs, as McCall's does; however, it also adds the

hardware yield characteristics not encountered in the McCall model (McCall and Joseph, 1978). Boehm's model looks at utility from various dimensions, considering the types of user expected to work with the system once it is delivered. General utility is broken down into portability, utility and maintainability. Utility is further broken down into reliability, efficiency and human engineering. Maintainability is, in turn, broken down into testability, understandability and modifiability. However, Boehm's model does not elaborate the methodology to measure these characteristics.

6.3.3 FURPS Model

FURPS (Grady, 1992) takes into account the five characteristics that make up its name: Functionality, Usability, Reliability, Performance, and Supportability. The model proposed by Robert Grady from Hewlett-Packard Co. decomposes characteristics into two different categories of requirements:

- Functional requirements: Defined by input and expected output.
- Non-functional requirements: Usability, Reliability, Performance, and Supportability.

One disadvantage of the FURPS model is that it does not consider the portability aspect, which may be an important criterion for application development, especially for component-based systems.

6.3.4 Dromey's Model

Dromey (1995) proposed a quality evaluation framework that analyzes the quality of software components through the measurement of tangible quality properties. All these components possess intrinsic properties that can be classified into four categories:

- Correctness: Evaluates component whether some basic principles are violated.
- Internal: Measure how well a component has been deployed according to its intended use.
- Contextual: Deals with the external influences by and on the use of a component.
- Descriptive: Measures the descriptiveness of a component (for example, does it have a meaningful name?).

These properties are used to evaluate the quality of the components. While Dromey's work is interesting from a technically inclined stakeholder's perspective, it is difficult to see how it could be used at the beginning of the lifecycle to determine user quality needs.

6.3.5 ISO 9126 Model

International Organization for Standardization (ISO) proposed a standard, known as ISO 9126 (ISO, 2001), which provides a generic definition of software quality in terms of six main characteristics for software evaluation. These characteristics include: Functionality, Reliability, Usability, Efficiency, Maintainability and Portability. All the characteristics then further contain their corresponding sub-characteristics as shown in Table 6.2.

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

Table 6.2: ISO 9126 Quality Characteristics

One of the advantages of this model is that it identifies the internal and external quality characteristics of a software product. On the other hand, it does not show very clearly how these aspects can be measured (Maryoly *et al.*, 2002).

The quality models described above (McCall and Joseph, 1976; Boehm, 1978; Grady, 1992; Dromey, 1995 and ISO, 2001) contain several factors in common, like maintainability, efficiency, reliability. However, some of the factors like correctness, maturity, and understandability are not so common and are found in one or two models. These quality models are summarized in the Table 6.3.

Quality Attributes	Boehm	McCall	FURPS	Dromey	ISO 9126
Efficiency	☆	☆	☆	☆	☆
Reliability	☆	☆	☆	☆	☆
Maintainability	☆	☆	☆	☆	☆
Portability	☆	☆		☆	☆
Usability		☆	☆	☆	☆
Testability	☆	☆			☆
Functionality			☆	☆	☆
Reusability		☆		☆	
Interoperability		☆			☆
Integrity		☆			☆
Flexibility		☆	☆		
Understandability	☆				☆
Changeability/Customizability	☆				☆
Maturity				☆	☆
Correctness		☆			
Human Factor	☆				

Table 6.3: Comparative Analysis of Quality Factors in various Models

6.4 QUALITY MODELS FOR COMPONENT-BASED SYSTEMS

In component-based systems, it is very difficult to relate system properties to component properties (Crnkovic *et al.*, 2005). For component-based systems, crucial questions in relation to choosing a quality component are the following:

- How likely is it that the component will work properly in target application without or with minor reconfiguration?
- In what ways has the component been tested in a sufficient variety of situations and are the testing methods relevant to intended use?
- Has the component already been used in other applications similar to this, if so, how much and with what result?

- What are the implications of using this component on system performance, reliability, maintainability, portability and other aspects?

To answer these questions, we need to choose those quality attributes and finally a quality model which may be applied on software components and CBS. There is no general consensus on the traditional quality models which can fit for CBS. McCall's ignored Functionality, Boehm's contains a diagram without any suggestion about measuring the quality characteristics, and ISO 9126 does not show very clearly how the attributes can be measured. Thus, there is an absence of any kind of metrics that could help in evaluating quality characteristics objectively, in particular when the underlying software project is component-based.

However, there are several models proposed exclusively for CBS also and are described below briefly. But most of these models are based or derived from ISO9126. Sedigh *et al.* (2001b) categorized the system level metrics into management, requirement and quality. Management includes the cost, time-to-market, software engineering environment and system resource utilization. Requirements include requirements conformance and requirements stability, while quality includes adaptability, complexity of interface and integration test coverage, End-to-End test coverage, Fault profiles, reliability and customer satisfaction. The relationship among metrics is described using the influence diagram.

Bertoa and Vallecillo (2002) define the characteristics and sub-characteristics in the changed context of component-based systems. The work divides sub-characteristics into runtime and lifecycle categories based on their nature. It adds compatibility sub-characteristic under functionality, which indicates whether former versions of the component are compatible with its current version. The work also proposes various attributes associated with the sub-characteristics and finally defines these attributes with the classification for the measurement of such attributes like ratio, presence, integer and time (Bertoa and Vallecillo, 2002). Presence metric identifies whether an attribute is present in a component or not. This metric is used to measure time intervals. It uses an integer type variable to indicate the absolute value, together with a string variable that indicates the units (seconds, months, etc.). Level metric is used to indicate a degree of

effort, ability, etc. It is described by an integer variable that can take any of the following values: 0 (Very Low), 1 (Low), 2 (Medium), 3 (High), 4 (Very High). Lastly, ratio metric is used to describe either fraction or percentages. Although the paper presents a good description on quality characteristics, sub-characteristics and their measurement, it fails to perform any empirical evaluation of the attributes on any application, thus leaving the proposed work as incomplete.

Rawashdeh and Matalkah (2006) also conducted a survey on various quality models available, which includes McCall, Boehm, FURPS, Dromey and ISO 9126. It performed some tailoring on ISO 9126 model by adding compatibility sub-characteristic under functionality and complexity under usability. It omitted stability and analyzability from maintainability and adds manageability to it. It also added new characteristic, named Stakeholders in its proposed model who are the members of the team responsible for developing, maintaining, integrating and/or using COTS systems. The paper performed a good comparison for various quality models; however, all the models considered are traditional models and may not fit for component-based systems, as it is. Moreover, the proposed model does not explain how the attributes belonging to various characteristics and sub-characteristics will be measured to finally evaluate the quality of the system.

6.5 QUALITY ATTRIBUTES FOR COMPONENTS

The main quality characteristics (Functionality, Usability, Efficiency, Reliability, Portability and Maintainability) are available in almost all quality models proposed so far for component-based systems. However, researchers differ while choosing sub-characteristics under these characteristics. Proposed work in this chapter is an extension to the work mentioned above. Following section defines the terms used for various characteristics and sub-characteristics in our proposed quality model for software components:

6.5.1 Functionality

Functionality is a set of attributes that bear on the existence of a set of functions and their specified properties (ISO, 1991). It means that the component should provide

the functions and services as per the requirement when used under the specified condition. Pre-existing components with or minimum changes will allow low cost, faster delivery of end product. The sub-characteristics under functionality are briefly explained here:

- **Suitability:** Suitability expresses how well the component fits into the developer's requirements. As the exact requirement can only be known to system developer, it cannot be measured by component developer during its development.
- **Accuracy**: It evaluates accuracy of the component with correct precision level required by the system developer.
- **Interoperability:** This sub-characteristic indicates whether the format of the data, handled by the target component is compliant with any international or 'de facto' standard or convention.
- **Security**: It refers how the component is able to control the unauthorized access to the services provided to it.
- **Compliance**: This characteristic indicates if a component is conforming to any international standard or certification etc.).

6.5.2 Reliability

In general, reliability is the probability that a system or component will produce failure within a given period of time. In other words, reliability expresses the ability of the component to maintain a specified level of fault tolerance (fault frequency and fault severity), when used under specified conditions. Reusability aspect of the same component with multiple applications will increase the reliability of that component as it may be observed here that this component would have been thoroughly tested before using it in previous applications. Reliability is broken into the following sub-characteristics:

- **Maturity:** In component context, it deals with the number of commercial versions of the component and the time interval between each version.

- **Recoverability**: It measures the ability for a component to recover from an unexpected failure (e.g. through exception handling) and also to recover the lost data along with the original performance.
- **Fault Tolerance:** This sub-characteristic indicates whether the component can maintain a specified level of performance in case of faults.

6.5.3 Usability

It is the ability of a component to be understood, learned, used, configured, and executed, when used under specified conditions. Obviously, here the user of the component is application/system developer rather than end user. Therefore the usability of the component should be less complex and more reusable and developer friendly so that it can be assembled properly in the final system (Bertoa *et al.*, 2006). Sub-characteristics of Usability are defined as under.

- **Understandability:** It is the capability of the component to enable the user (system developer) to understand whether the component is suitable, and how it can be used for particular tasks and conditions of use.
- **Learnability:** This sub-characteristic refers the ability of the component to enable the system developer to learn the component. For example, the user documentation and the help system should be complete; the help should be context sensitive and explain how to achieve common tasks, etc.
- **Operability:** It is the capability of the component to enable the user (system developer) to operate and control it.
- **Attractiveness:** It indicates the capability of the software component to be attractive to the user. Here as stated earlier, the user is system developer, and he/she may be more interested in the programmatic interfaces, API's defining the services provided by the components so that they can be composed and integrated with the target system rather than its attractiveness or GUI interfaces. Therefore, we may omit this sub-characteristic from the proposed model.
- **Compliance**: This characteristic indicates if a component is conforming to any international standard or certification etc. relating to usability. Currently, no

standards have been set up for this sub-characteristic in the component context; therefore it may be omitted from the quality model for the time being.

6.5.4 Efficiency

This characteristic expresses the ability of a component to provide appropriate performance, relative to the amount of resources used. Efficiency is affected by the component technology, mainly through resource usage by the runtime system but also by interaction mechanism. Component can be internally optimized to improve performance without affecting their specifications. Components should be tested on various platforms to check the performance (Borretzen, 2005). The two sub-characteristics of efficiency are defined as follows:

- **Time behavior**: This characteristic indicates the ability to perform a specific task at the correct time, under specified conditions.
- **Resource behavior**: This characteristic indicates the amount of the resources used, under specified conditions.

6.5.5 Maintainability

This characteristic describes the ability of a component to be modified. As the component developers do not have the source code of the component, they can only adapt it, reconfigure it and finally test it before integrating it in the final product. The sub-characteristics under maintainability are:

- **Customizability:** As mentioned earlier, system developer can only adapt, reconfigure, test and finally embed it into the system, as he is not having the source code of the component. Customizability refers to the ability to modify the component through its limited available information, like interfaces and parameters. For example in JavaBeans, components can be customized through its customizable parameters (set methods).
- **Testability:** This sub-characteristic refers whether the component provides some sort of tests or test suites that can be performed to the component to check its functionality inside (or in isolation of) the final system in which the component will be integrated.

- **Stability**: It refers to the component ability to handle the unexpected changes during the maintenance. In other words, it is the degree to which software is composed of discrete components such that a change to one component has the minimal or zero impact on the other components or the system.
- **Analyzability:** It refers to the auto-analysis of component. It identifies the parts of the components, which are to be modified (Alvero *et al.*, 2005). However, a component rarely consists of methods for its auto analysis. Therefore analyzability may not be required and is removed from the model .

6.5.6 Portability

This characteristic is defined as the ability of a component to be transferred from one environment to another with little modification, if required. The component should be easily and quickly portable to new environments if and when necessary, with minimized porting costs and schedules. In component-based development, it is a very important characteristic as a component may be used and reused in various different environments. Therefore the specification of a component should be platform independent. Various sub-characteristics defined under portability are:

- **Replaceability:** This sub-characteristic indicates whether the component is backward compatible with its previous versions. This means that the new component can substitute the previous ones without any major efforts.
- **Adaptability**: It refers whether the component can be adapted to different specified platforms.
- **Installability:** It is the capability for a component to be installed easily on different platforms.

6.6 NEW CHARACTERISTICS ADDED TO PROPOSED QUALITY MODEL

Apart from above mentioned characteristics, we propose following sub -characteristics to be added in our proposed quality model:

6.6.1 Complexity

Because the source code of the component is not available to the application developer, the complexity of a component in CBD is limited to interface methods, pre and post conditions, properties and interactions available to the developer. Measuring the complexity at the initial stage is helpful during analyzing, testing, and maintaining the system. This measurement could direct the process of improvement and reengineering work. A complexity measure could also be used as a predictor of the effort that is needed to maintain the system. In component-based systems, functionalities are not performed within one component. Components communicate and share information in order to provide system functionalities. The system developers adapt the component, reconfigure it and then finally integrate it in the final system without going into the internals of the component. We propose that complexity should be added under usability characteristic.

6.6.2 Trackability

While reconfiguring a component for changed/improved functionality, maintainers must perform a full cycle of product evaluation, integration and testing. Even if the new release of a component claims to be functionally backward compatible with an older version, there will undoubtedly be differences such as resource usage, performance, or target system requirements (Sharma *et al.*, 2008a). Therefore, it is very necessary to keep track of all the parameters of the older version so that these details can be compared with the enhanced or changed parameters. Thus, maintenance activities may be extended to include provisions for keeping a proper track of various system properties during the maintenance activities, which may include tracking system performance or resource utilization, before and after any maintenance activity, like replacement of an old component with a new version with some added/modified functionality. This may also include the possible security violations due to some maintenance activities. Tracking not only will validate any improvement efforts, but also once an information process is presumed to be stable, tracking may provide insight into maintaining statistical control. This, in effect, will ease the overall maintenance process. Therefore trackability aspect should be added under the maintainability characteristic (Vigedar and Kark, 2006).

6.6.3 Reusability

Reusability is the intrinsic property of CBS. Reuse is the process of adapting a generalized component to various contexts of use. It improves productivity and maintainability and therefore the overall quality of the end product. A reusable component must be generic, that is, it has appropriate features that enable the re-user to create specific instances of the components to satisfy application specific requirements. We propose to add it under functionality characteristic.

6.6.4 Scalability

Scalability expresses the ability of the component to support major data volumes. It may also be measured as the number of users or transactions it can scale to without sensible decrease in response time. It is related with the performance and so should be included under efficiency characteristic.

6.6.5 Flexibility

It refers to the ease of change/modification of a component under maintenance activity.

6.7 NEW PROPOSED QUALITY MODEL

After tailoring the quality model for CBS, new proposed quality model for software components is as shown in Table 6.4. The sub-characteristics shown in bold are newly added to the ISO 9126 model.

Functionality	**Reliability**	**Efficiency**	**Usability**	**Maintainability**	**Portability**
Suitability	Maturity	Time Behavior	Understandability	Stability	Adaptability
Accuracy	Fault Tolerance	Resource Behavior	Learnability	Testability	Installability
Interoperability	Recoverability	**Scalability**	Operability	Changeability	Replaceability
Security			**Complexity**	**Trackability**	
Compliance				**Flexibility**	
Reusability					

Table 6.4: New Proposed Quality Model for Software Components

6.7.1 Evaluation of Proposed Model

Not all quality characteristics are of equal importance to the software product. Therefore, developers must realize that to design a successful system does not mean that the system must conform to all the software quality attributes defined, but rather the most important attributes need to be identified, specified and prioritized. Moreover, the priorities of these characteristics and sub-characteristics will vary from one application domain to another. Like for throwaway applications e.g. a system developed to manage Y2K problem, maintainability may be of no use. Similarly, if the application is intended to work only for a dedicated platform, portability aspect is not required at all. However, the financial applications will require more security, efficiency, reliability, fault tolerance and availability. Similarly, for an e-Commerce system, the important characteristics would be reliability, performance, availability, security, and maintainabilit y (maintenance on these applications occurs continuously) (Voas and Agresti, 2004). Therefore, we should consider only the just enough and related quality characteristics and sub-characteristics while developing a software system for a particular domain. I t will eventually reduce the overall efforts, time and cost for the development. Empirically, we can assign weight values to these characteristics and sub-characteristics based on their importance in these domains. These weight values may vary from one dom ain to another.

6.7.2 Weight Assignment Technique for Quality – Case Study

To establish these facts in real practice, we conducted a survey on software professionals working on e-commerce projects by using component-based technologies in different multi-national companies. These companies include Steria (I) Ltd., HCL Technologies, R Systems International Ltd., McAfee Software (India) Pvt. Ltd and RMSI India Ltd. These professionals have varied experience (5-12 years) ranging from senior software developers to the project managers. All the professionals involved have completed at least three projects on these technologies with varied responsibilities. Survey form (sample attached in appendix) consists of all the quality characteristics and sub-characteristics proposed in our model. Professionals were requested to give their preferences on these characteristics and sub-characteristics ranging from *Never Required* - 1 to *Always Required* - 4, while keeping in mind a particular application domain.

Response received from these professionals is analyzed through Analytical Hierarchy Process (AHP) concept, explained briefly as:

6.7.3 Analytical Hierarchy Process (AHP)

The AHP Model was designed by T. L. Saaty (Saaty, 1994) as a decision making aid. It is a quantitative technique that facilitates structuring a complex multi-attribute problem. It provides an objective methodology for deciding among a set of solution strategies for solving that problem. Users of the AHP decompose their decision problem into a hierarchy of more easily comprehended sub-problems, each of which can be analyzed independently. The elements of the hierarchy can relate to any aspect of the decision problem—tangible or intangible, carefully measured or roughly estimated, well- or poorly-understood—anything at all that applies to the decision at hand. Its application has been reported in numerous fields, such as transportation planning, portfolio selection, corporate planning and marketing. It involves:

- Development of relative importance among the attributes using experts' opinion or through exhaustive paired comparison analysis.
- Developing through an algorithm a weightage for each of the attributes.
- Performing similar analysis for the alternative solution strategies for each of the attributes.
- Developing a single overall score for each of the alternate solution strategies.

6.7.4 Evaluation of the Proposed Model

The survey is conducted on 36 professionals and analyzed in three phases by taking first 10 then 25 and finally total 36 persons. After each phase, we analyzed the response by using AHP and we found a maximum of 3% variation in the results, which indicates towards the sufficiency of sample data and shows that there is no much difference in the result if we increase the number of respondents. MS-Excel is used to process the general data. The weight values for each main characteristic in the range from 0 to 1 are shown in Table 6.5. The sum of all weight values is 1.

Functionality	Reliability	Usability	Efficiency	Maintainability	Portability
0.205	0.184	0.202	0.160	0.139	0.110

Table 6.5: Weight Values for Quality Characteristics

Table shows that in e-Commerce domain, functionality and usability are two most important characteristics, while workers gave least preference to portability. However, as a characteristic, this lowest value cannot be ignored and has to be considered while evaluating the overall quality for CBS.

Further, weights of sub-characteristics under these characteristics are decided by AHP analysis. The weight value of a characteristic is distributed among its sub-characteristics. These weight values are used to calculate the measured value of a quality characteristic from the measured values of its sub-characteristics. The sum of the weight values of all the sub-characteristics under a characteristic is equal to the weight value of that parent characteristic. Similarly, the sum of all sub-characteristics, irrespective of parent characteristics, is again equal to 1. The weight values of the sub-characteristics are shown in Table 6.6.

It may be noted from this table that the weight value for functionality is the highest among all, but its sub-characteristics have very less weight values as compared to other sub-characteristics. Similarly, though efficiency and portability have very less weights, but their constituents are assigned quite large weights as compared to others. The reason is obvious that here number of sub-characteristics are not same for all characteristics. Functionality divides its weight value among its seven sub-characteristics, while portability and efficiency divide among three. Therefore while choosing the sub-characteristics, the weight of their parent characteristics should also be considered.

Characteristic	Sub-characteristics	Weight Values	Sum	Grand Total
Functionality	Suitability	0.037	0.205	
	Accuracy	0.043		
	Interoperability	0.028		
	Security	0.039		
	Compliance	0.026		
	Reusability	0.032		
Reliability	Maturity	0.067	0.184	
	Fault Tolerance	0.060		
	Recoverability	0.057		
Efficiency	Time Behavior	0.056	0.16	
	Resource Behavior	0.052		
	Scalability	0.052		
Usability	Understandability	0.057	0.202	
	Learnability	0.048		
	Operability	0.049		
	Complexity	0.048		
Maintainability	Stability	0.022	0.139	
	Testability	0.032		
	Changeability	0.033		
	Trackability	0.026		
	Flexibility	0.026		
Portability	Adaptability	0.043	0.11	
	Installability	0.037		
	Replaceability	0.030		
				1.00

Table 6.6: Weight Values for Quality Sub-characteristics

The weight values obtained through the process mentioned above may help developers to select those quality characteristics and sub-characteristics, which are important and relevant as per their quality requirement in that domain. Through this

weighting technique, developers can be restricted not to over design the system in favor of a particular characteristic. It will control the development time and financ ial tradeoff also. But if we choose only selected characteristics and sub-characteristics, the sum of these will not be 1, as we are excluding the weight values of not-so-important characteristics/sub-characteristics. In this case, we can normalize the wei ght values of selected characteristics/sub-characteristics relatively to make the sum equal to 1. For example, if only suitability, accuracy, security and reusability have to be selected under the functionality characteristic, which have the weight values 0.033, 0.038, 0.035 and 0.028. Other attributes like interoperability, compliance and compatibility are not considered to be important (though these have been assigned certain weight values). We need to normalize these weight values to the selected sub-characteristics so that the sum of all these selected sub-characteristics will be equal to 0.205 (weight value for functionality shown in Table 6.5). The following formula can be used to normalize the weight value of sub-characteristic say SC_i, which is under characteristics C:

$$\textit{Normalized Weight Value of } SC = \frac{\textit{Weight Value of } C}{\sum \textit{Weight Values of Selected sub characteristics}} * \textit{Weight Value of } SC \qquad (6.1)$$

The normalized weight values in this case will be 0.05, 0.058, 0.054 and 0.043. Similarly, we can get the normalized weight values of all the sub-characteristics chosen for a particular application for a domain.

6.7.5 Experimental Evaluation of Quality as a Single Variable

Quality is the composition of all its selected characteristics and sub-characteristics. To measure it as a single variable, one must include the contributi on of each constituent and sub-constituent. The objective of developer must be to achieve all selected quality characteristics and sub-characteristics in a best possible way. However, in some cases, it may be possible that if we increase one aspect, other will automatically be decreased. For example, if we try to increase the reusability or portability, complexity will be increased. Similarly, to implement more security aspects will reduce the performance. Therefore, efforts need to be made to establish an accepted balance among

all these. Equation 6.1 shows the formula used to evaluate the quality of the system as a whole (Voas and Agresti, 2004):

$$Q = w_f F + w_r R + w_u U + w_e E + w_m M + w_p P \qquad (6.2)$$

where w_f, w_r, w_u, w_e w_m and w_p are the weight values for the quality characteristics - functionality (F) , reliability (R), usability (U), efficiency (E), maintainability (M) and portability (P) respectively. Further, the characteristics are divided into sub - characteristics, the values of F, R, U, E, M and P can be measured by using their constituents' like-

$$F = w_s S + w_a A + w_i I + w_{se} Se + w_c C + w_{co} Co + w_{re} Re \qquad (6.3)$$

$$R = w_{ma} Ma + w_{ft} Ft + w_{rec} Rec \qquad (6.4)$$

where S: Suitability, A: Accuracy, I: Interoperability, Se: Security, C: Compliance, Co: Compatibility, Re: Reusability, Ma: Maturity, Ft: Fault Tolerance and Rec: Recoverability. Also, the weight value symbols are used as a multiplier along with the sub-characteristics (like w_s is the weight value of S). Other characteristics can also be expressed in a similar way.

Now, to evaluate the individual sub-characteristics, we need to use the related metric. But here units of all the metrics may not necessarily be same. Like, some may be measured in numbers; others may be in ratio or just presence. Therefore, we need to normalize these values in the range of 0 to 1, so that all can be fit under a unique scale. Within the normalized range, the highest value for a metric is 1 and is the maximum achievable level.

6.7.6 Experimental Evaluation of Quality on a Case Study

To evaluate the quality as a whole, we conducted an experiment on a class room based project, which was to be developed in Java and related technologies. Two COTS components were selected and their quality were evaluated and compared to choose the better component for the application. After a preliminary consideration, the following characteristics and sub-characteristics, shown in Table 6.7, were specified for the evaluation:

Functionality	Reliability	Efficiency	Usability	Maintainability
Suitability	Maturity	Scalability	Understandability	Changeability
Accuracy		Resource Behavior	Learnability	Testability
Security				Trackability
Reusability				

Table 6.7: Selected quality attributes for evaluation

Here portability and its sub-characteristics are not selected, as the application is intended only for a fixed platform. Based on these selected characteristics and sub - characteristics, the normalized weight values are obtained as given in the Table 6.8 and 6.9.

Functionality	Reliability	Usability	Efficiency	Maintainability
0.23	0.207	0.227	0.18	0.156

Table 6.8: Normalized Weight Values for Selected Quality Characteristics

Characteristic	Sub-characteristics	Weight Values	Sum	Grand Total
Functionality	Suitability	0.057	0.230	
	Accuracy	0.066		
	Security	0.060		
	Reusability	0.047		
Reliability	Maturity	0.207	0.207	
Efficiency	Resource Behavior	0.18	0.180	
Usability	Understandability	0.123	0.227	
	Learnability	0.104		
Maintainability	Testability	0.055	0.156	
	Changeability	0.056		
	Trackability	0.045		
Total				**1.00**

Table 6.9: Normalized Weight Values for Selected Quality Sub-characteristics

Table 6.10 shows the set of metrics collected to measure these characteristics and sub-characteristics. Out of these selected 11 sub-characteristics, 4 (marked with *) need to be normalized on 0 to 1 scale.

Parameter	Metric
Suitability	1-(No. of operations not suitable/Total number of operations provided)
Accuracy	Number of operations having required accuracy/Total number of operations
Security	No. of access controllability provided/Total no. of access controllability required
Reusability	Number of customizable properties/total number of properties
Maturity*	No. of versions released so far for the same component
Resource Behavior	1- (%CPU usage for the execution of the component/100)
Understandability*	Documentation, Help System, Training provided
Learnability	No. of observable properties/Total number of properties
Changeability	Number of customizable properties/total number of properties
Testability*	Sufficient number of test cases provided
Trackability*	Functional and Behavioral tracking system provided for easy maintenance

Table 6.10: Set of Metrics to be used for evaluation

For Maturity, we assume that a component having its first version has maturity 0, second released version 0.25, third released versions 0.5, forth version 0.75 and five or more versions means the mature enough component and it may have maturity value 1.0. Understandability can be measured on the basis of documentation; help system and training provided to the users for the target component and 0.33 each can be assigned for these three constituents respectively. Testability may be measured on the basis of number of test cases provided by the component vendor/developer along with the component. We propose to assign value 0, if no test cases are provided at all. Value 0.5 may be assigned, if test cases are provided but not sufficient to test the component thoroughly and finally value 1, if adequate number of test cases is provided. Lastly, trackability may be measured on the basis of tracking mechanism provided by the component developer for

maintenance activities. This may include the provision for keeping track of both functional and non-functional aspects like functionality, performance, security violations, portability affected due to the maintenance activities. Again, 0, 0.5 and 1 can be assigned for no tracking provided at all, only functional or behavioral tracking provided, and both functional and behavioral tracking provided, respectively.

Now, all the sub-characteristics selected for both the components are normalized on the scale of 0 to 1. The values obtained for these metrics for first component are shown in Table 6.11. Result shows that the quality of this component is 0.583 on a scale of from 0 to 1.

Characteristics	**Sub-characteristic**	**Metric (M)**	**Weight Value (w)**	**$M_i * w_i$**	**Quality Measure of Characteristics**
Functionality	Suitability	1-1/7 = 0.85	0.057	0.048	0.184
	Accuracy	6/6=1	0.066	0.066	
	Security	¾=0.75	0.060	0.045	
	Reusability	7/13=0.54	0.047	0.025	
Reliability	Maturity	0.25	0.207	0.052	0.052
Efficiency	Resource Behavior	0.89	0.180	0.160	0.160
Usability	Understandability	0.66	0.123	0.081	0.129
	Learnability	6/13=0.46	0.104	0.048	
Maintainability	Changeability	7/13=0.54	0.056	0.030	0.058
	Testability	0.5	0.055	0.028	
	Trackability	0	0.045	0	
Total					**0.583**

Table 6.11: Final Value of Quality

Similar approach is applied to get the value of quality for s econd component, which is 0.437. These two values indicate that first component is better than the second one and can be used in the application.

6.8 CONCLUSION

Proposed methodology may be used to estimate the quality of any component before using it in the final system. It will help in selecting the high quality component for the system. Also, it may be used to estimate the efforts required to achieve a required value of any characteristic. Like in the above experimentation, the score of the functionality is 0.184 out of 0.205. To achieve a minimum accepted level of functionality for example say 0.19, we need to increase the value of some of its constituent parameters, without affecting others. We can achieve it by putting extra efforts in security by providing one more controllability aspect, so that the metric value for security now is 1.0 and final value of functionality will become more than the minimum required value. However, it may be possible that if we increase security aspect, it will decrease oth er aspects like complexity or performance. Therefore, we need to have a balance between these values so that the component can be used in the final system. Further, to achieve a particular level of any quality aspect, developers must employ certain techniq ues, methodologies, tools and processes, each of which has associated costs and benefits. These costs and benefits need be considered while implementing the requirement analysis.

Chapter 7

SUMMARY AND SUGGESTIONS FOR FUTURE WORK

7.1 INTRODUCTION

The primary objective of this work has been to design metrics and study other aspects related to the quality of software components and component-based systems. Salient findings of this study are briefly recapitulated point wise and some pointers to future research in this direction are being summarized in this chapter.

7.2 REVIEW OF RESULTS OF THIS STUDY

We have designed an Interface Complexity Metric for software components and presented in Chapter 2. This metric is based on the complexities involved in interface methods and properties of these interfaces in components. This metric is implemented on a number of Java Bean components. Validation of this proposed metric is conducted by using other aspects including performance, customization and readability. A correlation analysis is performed among proposed complexity metric and these three metrics. Results show that complex components take much time to execute than simple components and are very difficult to maintain. Also, complex components are not easily understandable. These facts may be obvious, though, but our work has brought quantitative affirmation of all these.

In Chapter 3, we discussed several dependency representation techniques among components. We proposed a link-list based technique for component dependency representation, which may be an efficient and faster technique in comparison to others for complex systems. Also, it helps to extract several useful measures like interaction density, interaction complexity, isolated components, most critical components and others, which may be useful for estimating reusability, maintainability and overall development efforts.

Chapter 4 proposes an Artificial Neural Network based approach to estimate reusability of software components. This approach considers factors namely, customizability, interface complexity, portability and understandability as input to estimate reusability as an output. Several real-life components were used to get the data for training and testing purpose. Results show that network is able to predict the reusability with an RMSE 0.1648. The proposed approach may be used by application developers to select the highly reusable components for their application. System developed with better reusable components will result in high maintainability.

Chapter 5 discusses various activities and challenges in maintaining component-based systems. It proposes to modify the maintainability characteristic by adding a new sub-characteristic, trackability, under it. Trackability will keep track of all non-functional maintenance activities like, security, performance, conformance to standards and others. It will result in better understanding the maintenance task. The work also proposes a fuzzy logic based approach to estimate maintainability for CBS. It considers factors like interaction complexity, reusability, testability, understandability, trackability as inputs while maintainability is considered as output. In total 243 fuzzy sets have been designed and are represented by membership functions (Low, Medium and High). Output is represented by Very Low, Low, Medium, High and Very High. Defuzzification process produces the value of maintainability corresponding to input sets. This proposed model is then validated on two student class room based projects. Results show that proposed model can be used to predict the maintainability of CBS, which will help in estimating development efforts and quality for the application.

Chapter 6 performs a detailed study on various existing quality models for legacy and component-based systems. It proposes a new quality model by tailoring some of the existing properties available in these models. It also proposes some new sub-characteristics, namely complexity, trackability, reusability, flexibility in the model. Further, priorities are set for all these characteristics and sub-characteristics based on the requirement of these in a particular domain. Survey and AHP based methodologies are used to assign the weight values. This model is then evaluated on a classroom based project to compare and select the better quality component for the application. Proposed

methodology produces the quality of component as a single measurable variable and can be used by developers to choose better quality component for their end product.

7.3 SUGGESTIONS FOR FUTURE WORK AND LIMITATIONS

Since the size of the component dataset is small, results obtained for our proposed metrics (interface complexity and reusability) may be of limited capability. This is because of limited number of trial components available on web sites. Other components are commercial and have financial implications. However, results obtained from the present study are quite instructive and give the hope that industry will find it suitable for implementation in real-life applications. We are generating the information continuously and hope that based on the larger size of data, the results can be further refined.

Different cross-validation methods need to be explored to a greater and wider extent for better acceptability of the proposed metrics and quality model.

In the proposed work, we considered only the Java Bean components, as resources for these components are easily available as compared to others. However, other component technologies like, COM, DCOM, .NET etc. may also be used for evaluation and validation of proposed metrics. Proposed quality model is evaluated and validated on a class room based project. Efforts are being made to implement it on real-life industry based applications.

Reusability and maintainability metrics are estimated by using Artificial Neural Network and Fuzzy Logic-based approaches. However, other emerging approaches in software engineering like Neuro-Fuzzy may also be used to estimate these aspects, because incorporating the hybrid approach may have greater benefits. Therefore, the future work in this direction need be carried out to further explore various emerging technologies. We are looking into these aspects also.

REFERENCES

1. Aggarwal, K. K., Singh, Y., Chandra, P., Puri, M., 2005. Measurement of Software Maintainability Using a Fuzzy Model, Journal of Computer Sciences, Vol. 1, Issue 4, pp: 538-542.

2 Aggarwal, K.K., Singh, Y., Chandra, P., Puri, M., 2005. Sensitivity Analysis of Fuzzy and Neural Network Models, ACM SIGSOFT Software Engineering Notes, Vol. 30, Issue 4, pp: 1-4.

3 Aggarwal, K. K., Singh, Y., Kaur, A., Malhotra, R., 2005. Software Reuse Metrics for Object-Oriented Systems, Proceedings of the Third ACIS Int'l Conference on Software Engineering Research, Management and Applications, pp: 48–55.

4 Aggarwal, K. K., Singh, Y., Kaur, A., Malhotra, R., 2006. Application of Artificial Neural Network for Predicting Maintainability using Object-Oriented Metrics, Transactions on Engineering, Computing and Technology, Vol. 15, pp: 285-289.

5 http://en.wikipedia.org/wiki/Analytic_Hierarchy_Process

6 Ahn, Y., Suh, J., Kim, S., Kim, H., 2003. The Software Maintenance Project Effort Estimation Model Based on Function Points, Journal of Software Maintenance: Research and Practice, Vol. 15, Issue 2, pp: 71-85.

7. Alvero, A., Santana, E., Almeida, R., Meira, S., 2005. Quality Attributes for a Component Quality Model, Proceeding of 10th International Workshop on Component-Oriented Programming, Glasgow, Scotland (ECOOP'05), Glasgow, pp: 1-9.

8. Ardimento, P., Bianchi, A., Visaggio, G., 2004. Maintenance-oriented Selection of Software Components, Proceedings of 8th European Conference on Software

Maintenance and Reengineering, pp: 115–124.

9. Arsanjani, A., Zedan, H., Alpigini, J., 2002. Externalizing Component Manners to Achieve Greater Maintainability through a Highly Re-configurable Architectural Style, Proceedings of International Conference on Software Maintenance, pp: 628–637.

10. Bandi, R. K., Vaishnavi, V. K., Turk, D. E., 2003. Predicting maintenance performance using object-oriented design complexity metrics, IEEE Transactions on Software Engineering, Vol. 29, Issue 1, pp: 77–87.

11 Bandini, S., Paoli, F. D., Manzoni, S., Mereghetti, P., 2002. A support system to COTS-based software development for business services, Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, Vol. 27, pp: 307–314.

12 Banker, R. D., Kauffman, R. J., Wright, C. R. and Zweig, D., 1991. Automating Output Size and Reusability Metrics in an Object-Based Computer Aided Software Engineering (Case) Environment, Information Systems Working Papers Series, available at: http://www.ssrn.com/abstract=1289052.

13 Barnard, J., 1998. A New Reusability Metric for Object-Oriented Software, Software Quality Control, Vol. 7, Issue 1, pp: 35–50.

14. Bansiya, J., Davis, C., 2002. A Hierarchical Model for Object-Oriented Design Quality Assessment, IEEE Transactions on Software Engineering, Vol. 28, Issue 1, pp: 4-17.

15. Basili, V.R., 1980. Qualitative Software Complexity Models: A Summary in Tutorial on Models and Methods for Software Management and Engineering, IEEE Computer Society Press, Los Alamitos, California.

16. Basili, V.R., Rombach, H. D., 1988. Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment, Technical Report CS-TR-2158, Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742.

17 Beizer, B., 1990. Software Testing Techniques, Van Nostrand Reinhold, New York.

18. Bertoa, M., Vallecillo, A., 2002. Quality Attributes for COTS Components, I+D Computacion, Vol. 1, Issue 2, pp: 128-144, available at http://www.alarcos.inf-cr.uclm.es/qaoose2002/docs/QAOOSE-Ber-Val.pdf.

19. Bertoa, M., Troya, J. M., 2003. A Survey on the Quality Information Provided by Software Component Vendors, Proceedings of 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2003), pp: 1-6.

20. Bertoa, M., Troya, J. M., Vallecillo, A., 2006. Measuring the Usability of Software Components, Journal of Systems and Software, Vol. 79, Issue 3, pp: 427-439.

21. Boehm, B. W., Brown, J. R., Kaspar, J. R., Lipow, M. L. & MacCleod, G. , 1978.

Characteristics of Software Quality. New York: American Elsevier.

22. Boehm, B. W., Brown, J. R., Yang, Y., 2004. A Software Product Line Life Cycle Cost Estimation Model, IEEE Proceedings of Empirical Software Engineering (ISESE'04), pp: 156-164.

23. Borretzen, J. A., 2005. The Impact of Component-Based Development on Software Quality Attributes, available at http://www.idi.ntnu.no/emner/dt8100/Essay2005/Boerretzen.pdf

24. Boxall, M. A. S., Araban, S., 2004. Interface Metrics for Reusability Analysis of Components, Proceedings of. Australian Software Engineering Conference (ASWEC'2004), Melbourne, Australia, pp: 40-46.

25. Brereton, P., Budgen, D., 2000. Component-Based Systems: A Classification of Issues, IEEE Computer, Vol. 33, Issue 11, pp: 54-62.

26. Briand, L., Daly, J. W., Wust, J. K., 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems, IEEE Transactions on Software Engineering,

Vol. 25, Issue 1, pp: 99-121.

27. Cai, X., Lyu, M.R., and Wong, K., 2005. A Generic Environment for COTS Testing and Quality Prediction, Testing Commercial-off-the-shelf Components and Systems, Sami Beydeda and Volker Gruhn (eds.), Springer-Verlag, Berlin, pp: 315–347.

28. Caldiera, G., Basili, V. R., 1991. Identifying and Qualifying Reusable Software Components, IEEE Computer, Vol. 24, Issue 2, pp: 61-70.

29. Chidamber, S., Kemerer, C., 1994. A Metrics Suite for Object-oriented Design, IEEE Transactions on Software Engineering, Vol. 20, Issue 6, pp: 476-493.

30. Cho, E. S., Kim, M. S., Kim, S. D., 2001. Component Metrics to Measure Component Quality, Proceedings of 8th Asia-Pacific Software Engineering Conference, Macau, pp: 419-426.

31. Clements, P. C., Northrop, L. M., 1996. Software Architecture: An Executive Overview (CMU/SEI-96-TR-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

32. Cote, M., Suryn, W. Georgiadou, E., 2006. Software Quality Model Requirements for Software Quality Engineering, Proceedings of Software Quality Management and INSPIRE Conference, pp: 31-50.

33. Crnkovic, I., Larsson, M., and Preiss, O., 2005. Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes, Chapter in Architecting Dependable Systems III, LNCS 3549, pp: 257-278.

34. D' Souza, D. F., Wills, A. C., 1999. Objects, Components and Frameworks with UML: The Catalysis Approach. Addison Wesley, Reading, MA, 1999.

35. Devanbu, P. Karstu, S., Melo, W., Thomas, W., 1996. Analytical and Empirical Evaluation of Software Reuse Metrics, Proceedings of 18th International Conference on

Software Engineering, Berlin, Germany, pp: 189–199.

36. Dromey, R. G., 1995. A Model For Software Product Quality, IEEE Transactions on Software Engineering, Vol. 21, Issue 2, pp: 146-162.

37. Dumke, R., Schmietendorf, A., 2000. Possibilities of the Description and Evaluation of Software Components, Metrics News, Vol. 5, Issue 1, pp: 13-26.

38. Edmond, V., 2007. Halstead Complexity Measures, available at http://www.sei.cmu.edu/str/ descriptions/halstead_body.html.

39. http://www.elegantjbeans.com

40. Fioravanti, F., Nesi, P., 2001. Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems, IEEE Transactions on Software Engineering, Vol. 27, Issue 12, pp: 1062–1084.

41. Fonash, P., 1993. Characteristics of Reusable Software Code Components, Ph.D. Dissertation, George Mason University, Fairfax, Virginia.

42. Frakes, W., Terry, C., 1996. Software Reuse: Metrics and Models, ACM Computing Surveys, Vol. 28, Issue 2, pp: 415–435.

43. Freedman, R. S., 1991. Testability of Software Components, IEEE Transactions of Software Engineering, Vol. 17, Issue 6, pp: 553-564

44. Gill, N. S., 2003. Reusability Issues in Component-based Development, ACM SIGSOFT Software Engineering Notes, Vol. 28, Issue 4, pp: 1-5.

45. Gill, N. S., Grover, P. S., 2003. Component-Based Measurement: Few Useful Guidelines, ACM SIGSOFT Software Engineering Notes, Vol. 28, Issue 6, pp: 1-4.

46. Gill, N. S., Grover, P. S., 2004. Few Important Considerations for Deriving Interface Complexity Metric for Component-Based Systems, ACM SIGSOFT Software

Engineering Notes, Vol. 29 Issue 2, pp: 1-6.

47. Gill, N. S., 2006. Importance of Software Component Characterization for Better Software Reusability, ACM SIGSOFT Software Engineering Notes, Vol. 31, Issue 1, pp: 1-3.

48. Gill, N. S., Balkishan, 2008. Dependency and Interaction Oriented Complexity Metrics of Component-Based Systems, ACM SIGSOFT Software Engineering Notes Vol. 33 Issue 2, pp: 1-5.

49. Geoff, D., 1994. A Model for Software Product Quality, IEEE Transactions on Software Engineering, Vol. 21, Issue 2, pp: 146-162.

50. Ghezzi, C., Jazayeri, M., Mandrioli, D., 1991. Fundamentals Of Software Engineering, Prentice-Hall, New Jersey, USA.

51. Goulao, M., Abreu, F. B., 2004. Independent Validation of a Component Metrics Suite, 8th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE '04), Oslo, Norway.

52. Grady, R. B., 1992. Practical Software Metrics For Project Management And Process Improvement, Prentice Hall.

53. Grover, P. S., Kumar, R., Sharma, A., 2007. Few Useful Considerations for Maintaining Software Components and Component-Based Systems. ACM SIGSOFT Software Engineering Notes, Vol. 32, Issue 4, pp: 1-5.

54. Gui, G., Paul, D. S., 2007. Ranking Reusability Of Software Components Using Coupling Metrics, Journal of Systems and Software, Vol. 80, Issue 9, pp: 1450-1459.

55. Guo, J., 2002. Using Category Theory to Model Software Component Dependencies, Proceedings of the 9th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS, 02), pp: 185-192.

56. Halstead, 1977. Elements of Software Science, New York, Elsevier North -Holland.

57. Haykin, S., 2003. Neural Networks, A Comprehensive Foundation, Prenti ce Hall India.

58 Hudson, D. L., Cohen, M. E., 2003. Neural Networks & Artificial Intelligence for Biomedical Engineering, Prentice Hall of India.

59. IEEE Standard Glossary of Software Engineering Terminology , 1990. IEEE Std 610.12-1990, The Institute of Electrical and Electronics Engineers , Inc.

60. IEEE Standard for Software Maintenance , 1998. IEEE Std 1219-1998. The Institute of Electrical and Electronics Engineers , Inc.

61. Ince, D., 1994. ISO 9001 and Software Quality Assurance , McGraw-Hill, New York.

62. ISO 8402 Quality Management and Quality Assurance - Vocabulary, 1994. International Organization for Standardization, Geneva, 2nd Edition .

63. International Standard, ISO/IEC 9126, (2001). Institute of Electrical and Electronics Engineering, Part 1, 2, 3: Quality Model .

64. Jacobson, I., Booch, G., Rumbaugh, J., 2003. The Unified Software Development Process, Addison Wesley Inc., USA.

65. http://www.jars.com

66. http://www.java.sun.com/developer/onlineTraining/Beans

67. Judith, A. C., Audrey, E. T., 1998. A Management Guide to Software Maintenance in COTS-Based Systems, a Technical Report of Mitre, Center for Air Force C2 Systems: Bedford, MA, pp: 1-35.

68. Jung, H., Kim, S., Chung, C., 2004. Measuring Software Product Quality: A Survey of ISO/IEC 9126, IEEE Software, Vol. 21, Issue 5, pp: 88–92.

69. Kafura, D., Henry, S., 1981. Software Quality Metrics, Based on Interconnectivity, Journal of Systems and Software. Vol. 2, pp: 121-131.

70. Kajko-Mattsson, M., Canfora, G., Chorean, D., van Deursen, A., Ihme, T., Lehmna, M., Reiger, R., Engel, T., Wernke, J., 2006. A Model of Maintainability – Suggestion for Future Research, Proceedings of International Multi-Conference in Computer Science & Computer Engineering (SERP'06), pp: 436-441.

71. Kamiya, T., Kusumoto, S., Inoue, K., Mohri, Y., 1999. Empirical Evaluation of Reuse Sensitiveness of Complexity Metrics, Information and Software Tech nology, Vol. 41, pp: 297-305.

72. Karunanithi, S., Bieman, J. M., 1993. Candidate Reuse Metrics for Object Oriented and Ada Software, Proceedings of IEEE International Software Metrics Symposium, available at http://www.cs.colostate.edu/~bieman/Pubs/santh iBiemanMetrics93.pdf

73. Kececi, N., Abran, A., 2001. Analyzing, Measuring And Assessing Software Quality Within a Logical-Based Graphical Model, available at http://www.gelog.etsmtl.ca/publications/pdf/602.pdf.

74. Khairuddin, H., Elizabeth, K., 1996. A Software Maintainability Attributes Model, Malaysian Journal of Computer Science, Vol. 9, Issue 2, pp: 92-97

75. Keller, A., Blumenthal, U., Kar, G., 2000. Classification and Computation of Dependencies for Distributed Management, Proceedings of the 5th International Conference on Computers and Communications (ISCC 2000), pp: 78 -84.

76. Kharb, L., Singh, R., 2008. Complexity Metrics for Component-Oriented Software Systems, ACM SIGSOFT Software Engineering Notes, Vol. 33, Issue 2, pp: 1-3.

77. Khosravi, K., Gueheneuc, Y.G., 2004. A Quality Model for Design Patterns, Online at:http://www.yanngael.gueheneuc.net/Work/Tutoring/Documents/041021+Kashayar +Khosravi+ Technical+Report.doc.pdf.

78. Kitchenham. B. and Kari. K. 1993. Inter item Correlations among Function Points, Proceedings of the IEEE Computer Society International Software Metrics Symposium, Baltimore, MD, pp: 11-15.

79. Kon, F., Campbell, R. H., 2000. Dependence Management in Component-based Distributed Systems, IEEE Concurrency, Vol. 8, Issue 1, pp: 26-36.

80. Land, R., 2002. Measurements of Software Maintainability, Proceedings of ARTES Graduate Student Conference, ARTES, pp: 1-7.

81. Larsson, M., 2004. Predicting Quality Attributes In Component-Based Software Systems, Ph.D. dissertation, Malardalen University.

82. Larsson, M., 2007. Applying Configuration Management Techniques to Component-Based System, MRTC Report, IT Licentiate thesis, 2000-07, Uppsala University.

83. Li, B., 1998. Software Reuse, available at http://sern.ucalgary.ca/courses /seng/693/ W98/ lib/reuse.htm

84. Li, B., 2003. Managing Dependencies in Component-Based Systems Based on Matrix Model, Proceedings of NetObject.Days, Erfurt, Germany, pp: 22-25.

85. Li, H., 1993. Object-Oriented Metrics that Predict Maintainability, Journal of Systems and Software, Vol. 23 Issue 2, pp: 111-122.

86. Lientz, B. P., Swanson, E. B., 2000. Software Maintenance Management, Addison-Wesley Reading, MA.

87. Lisa, C., Delugach, H. S., 2001. Dependency Analysis Using Conceptual Graphs, In Proceedings of the 9th International Conference on Conceptual Structures, ICCS 92001, pp: 117-130.

88. Losavio, F., Chirinos, L., Perez, M. A., 2002. Quality Models to Design Software Architecture, Journal of Object Technology, Vol. 1, Issue 4, pp: 165-178

89. MacDonell, S. G., Gray, A. R., Calvert, J. M., 1999. FLSOME: Fuzzy Logic for Software Metric Practitioners and Researchers, In the Proceedings of the 6 th International Conference on Neural Information Processing ICONIP'99, Perth, pp: 308 -313.

90. Mahmood, S., Lai, R., Kim, Y. S., Kim, J. H., Cheon, P. S., Suk, O. H., (2005), A Survey of Component-Based System Quality Assurance and Assessment, ActaPress Journal of Information and Software Technology, Vol. 47, Issue 10, pp: 693-707.

91. Mahmood, S., Lai, R., 2005. Measuring the Complexity of a UML Component Specification, Proceedings of Fifth International Conference on Quality Software (QSIC 2005), pp: 150-157.

92. Mari, M., Eila, N., 2003. The Impact of Maintainability on Component -based Software Systems, in Proceeding of the 29th EUROMICRO Conference (IEEE), pp: 25-31.

93. Mark, R. V., Dean, J., 2000. Maintaining a COTS-Based Systems, Proceedings of the NATO Information Systems Technology Panel Symposium on Commercial Off -the-Shelf Products in Defense Applications, Brussels, Belgium, pp: 1-6.

94. Mark, R. V., Anatol, W. K., 2006. Maintaining COTS-Based Systems: Start with the Design, in Fifth International Conference on Commercial -off-the-Shelf (COTS)-Based Software Systems, pp: 8-13.

95. Maryoly, O., Perez, M. A., Rojas, T., 2002. A Systemic Quality Model for Evaluating Software Products, available at http://www.lisi.usb.ve/publicaciones.

96. http://www.mathworks.com/access/help desk/help/toolbox/nnet/index.html?/access/ helpdesk/help/toolbox/nnet/purelin.html

97. Mayrhauser, A., Anderson, C., Mraz, R., 1995. Using a Neural Network to Predict Test Case Effectiveness, Proceedings of IEEE Aerospace Applications Conference, Snowmass, CO, pp:77-91.

98. McCabe, T., 1976. A Software Complexity Measure, IEEE Transactions on Software Engineering, Vol. 2, Issue 4, pp: 308-320.

99. McCall, J. A., Joseph, P. C., 1976. A Framework For The Measurement of Software Quality. ACM SIGMETRICS Performance Evaluation Review, Vol. 7, Issue 3-4, pp: 133 – 139.

100. Microsoft Corporation. Definition of the term component; http://www.msdn.microsoft.com/repository/OIM/resdkdefinitionofthetermcomponent.asp

101. Mili, H., Mili, F., Mili, A., 1995. Reusing Software: Issues and Research Directions, IEEE Transaction on Software Engineering, Vol. 21, Issue 6, pp: 528-561.

102. Mukherjee, A., Deshpande, J. M., 1995. Neural Network Based Expert Systems for Structural Design, Computers and Structures, Vol. 54, Issue 3, pp 367-375.

103. Musilek, P., Pedrycz, W., Succi, G., Reformat, M., 2000. Software Cost Estimation with Fuzzy Models, ACM SIGAPP Applied Computing Review, Vol. 8, pp: 24 -29.

104. Nael, S., 2006. Complexity Metrics AS Predictors of Maintainability and Integrability of Software components, Journal of Arts and Sciences, Issue 5, pp: 39-50.

105. Nagib, C. B., Callaos, De, 1994. Designing with a Systematic Total Quality, Educational Technology, Vol. 34, pp: 29-36.

106. Narsimhan, V. L., Hendradjaya, B., 2005. Theoretical Considerations for Software Component Metrics, Enformatika Transactions on Engineering, Computing and Technology, Vol. 10, pp: 169-174.

107. Narasimhan, V. L. and Hendradjaya, B., 2004. A New Suite of Metrics for the Integration of Software Components, 1st International Workshop on Object Systems and Software Architectures (WOSSA'2004), S. Australia, Australia, pp: 34 -39.

108. Ortega, M., Perez, M.A., Rojas, T., 2002. A Systemic Quality Model For Evaluating

Software Products, Software Quality Control, Vol. 11, Issue 3, pp: 219 – 242.

109. Paul, A., 2002. CBD Survey: The State of the Practice, a white paper by Cutter Consortium, available at http://www.cutter.com/research/2002/edge020305.html.

110. Pernilla, E., 2002. Dealing with the Complexity of CBSE – Fundamental Environmental Needs, Chapter 19: Industrial Experience with Dassault System Component Model, J. Estublier, J. M. Favre, R. Sanlavilla, pp: 86-92.

111. Poulin, J., Caruso, J., Hancock, D., 1993. The Business Case for Software Reuse, IBM Systems Journal, Vol. 32, Issue 40, pp: 567-594.

112. Preiss, O., Weqmann, A., Wong, J., 2001. On Quality Attribute Based Software Engineering, Proceeding of 27th EuroMicro Conference, pp: 114-121.

113. Pressman, R. S., 2005. Software Engineering: A Practitioner's Approach, 6th Edition, McGraw Hill Book Co.

114. Rawashdeh, A., Matalkah, B., 2006. A New Software Quality Model for Evaluating COTS Components, Journal of Computer Science, Vol. 2, Issue 4, pp: 373-381.

115. Rotaru, O. P., Dobre, M., Petrescu, M., 2005. Reusability Metrics for Software Components. Proceedings of the 3rd ACS / IEEE International Conference on Computer Systems and Applications (AICCSA-05), Cairo, Egypt, pp: 24-29.

116. Ryder, J., 1998. Fuzzy Modeling of Software Effort Prediction, Proceedings of IEEE Information Technology Conference, Syracuse, New York, pp: 53-56.

117. Saaty T. L., 1994. Fundamentals of Decision Making and Priority Theory with the Analytic Hierarchy Process, Vol. 6, Pittsburgh.

118. Sailu, M. O., Ahmed, M., and AlGhamdi, J., 2004. Towards Adaptive Softcomputing Based Software Effort Prediction, Fuzzy Information, Processing NAFIPS' 04, pp: 16-21.

119. Saleh, M. A., 2004. Measuring the Complexity of Component-Based System Architecture, Proceedings of International Conference on Information and Communication Technologies: From Theory to Applications (IEEE), pp: 593-594.

120. Sedigh-Ali, S., Gafoor, A., and Paul, R.A., 2001. Metrics-Guided Quality Management for Component-Based Software Systems, Proceedings of 25th Annual International Computer Software and Applications Conference (COMPSAC '01) , pp. 303-308.

121. Sedigh-Ali, S., Ghafoor, A., Paul, R. A., 2001. Software Engineering Metrics for COTS-Based Systems, IEEE Computer, Vol. 34, Issue 5, pp: 44-50.

122. Sedigh-Ali, S., Gafoor, A., and Paul, R. A., 2003. Metrics and Models for Cost and Quality of Component-Based Software, Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp: 149-156.

123. Selby, R.W., 1989. Quantitative studies of Software Reuse, Software Reusability, Vol. 2, applications and experience, Addison Wesley, pp: 213-233.

124. Sharma, A., Grover, P. S., Kumar, R., 2006. Investigation of Reusability, Complexity and Customizability Metrics for Component-Based Systems. ICFAI Journal of Information Technology, Vol. 1, Issue 2, pp: 6-11.

125. Sharma, A., Kumar, R., Grover, P. S., 2007. Complexity Measures for Software Components, WSEAS Transactions on Computers, Vol. 6, Issue 7, pp: 1005-1012.

126. Sharma, A., Kumar, R., Grover, P. S., 2008. Empirical Evaluation of Complexity for Software Components, International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Vol. 18, Issue 5, pp: 519-530.

127. Sharma, A., Kumar, R., Grover, P. S., 2008. Estimation of Quality for Software Components - an Empirical Approach, ACM SIGSOFT Software Engineering Notes, Vol. 33, Issue 5, pp: 1-10.

128. Sharma, A., Kumar, R., Grover, P. S., 2009. Reusability Assessment for Software Components – a Neural Network Based Approach, Accepted for publication in International IEEE Conference (IACT' 09) to be held at Thapar University, Patiala from 26-28 March, 2009.

129. Shukla, R, Mishra, A. K., 2008. Estimating Software Maintenance Effort - A Neural Network Approach, Proceedings of the 1st conference on India Software Engineering Conference, Hyderabad, India, pp: 107-112

130. Sindre, G., Conradi, R., Karlsson, E. A., 1995. The REBOOT Approach to Software Reuse, Journal of Systems and Software, Vol. 30, Issue 3, pp: 201-212.

131. Sivanandam, S. N., Sumathi, S., Deepa, S. N., 2007. Introduction to fuzzy logic using MATLAB, Springer.

132. Singh, Y., Kaur, A., Sangwan, O. P., 2004. Neural Model for Software Maintainability, Proceedings of International Conference on ICT in Education and Development (AISECT 2004), pp: 1-11.

133. Sparling, M., 2000. Lessons Learned Through Six Years of Component-Based Development, Communications of the ACM Journal, Vol. 43, Issue 10, pp. 47-53.

134. Stafford, J. A., Alexandar, L. W., Caporuscio, M., 2003. The Application of Dependence Analysis to Software Architecture Descriptions, Lecture Notes in Computer Science, Vol. 2804, pp: 52-62.

135. http://www.sun.java.com

136. Szyperski, C., 1999. Component Software - Beyond Object-Oriented Programming, 2nd Edition, Addison-Wesley.

137. Tawfiq, S. M., Abd, M. M., Green, S., 2007. A Software Cost Estimation Model Based on Quality Characteristics, Workshop on Measuring Requirements for Project and

Product Success, Palma de Mallorca, Spain, Available at: www-swe.informatik.uni-heidelberg.de/home/events/MeRePDocs/paper2.pdf.

138. Tomer, A., Goldin, L., Kuflik, K., Kimchi, E., Stephen, R., Schach, 2004. Evaluating Software Reuse Alternatives: A Model and Its Application to an Industrial Case Study, IEEE Transactions on Software Engineering, Vol. 30, Issue 9, pp: 601-612.

139. Tracz, W., 1990. The 3 Cons of Software Reuse, Proceedings of the Third Annual Workshop on Software Reuse, Syracuse, NY.

140. Tracz, W., 1997. Developing Reusable Java components, Proceedings of the symposium on Software Reusability, Boston, Massachusetts, United States, pp: 100 – 103.

141. Vernazza, T., Granatella, G., Succi, G., Benedicenti, L., Mintchev, M., 2000. Defining Metrics for Software Components, Proceedings of. World Multi-conference on Systematics, Cybernetics and Informatics, Vol. 11, pp: 16-23.

142. Vieira, M., Dias, M., Richardson, D. J., 2001. Describing Dependencies in Component Access Points, Proceedings of the 4th Workshop on CBSE, 23rd International Conference on Software Eng. (ICSE 2001), Toronto, Canada, pp: 115-118.

143. Vieira, M., Richardson, D. J., 2002. Analyzing Dependencies in Large Component-Based Systems, Proceedings of the 17th IEEE International Conference on Auto-mated Software Engineering, Edinburgh, UK, pp: 241-246.

144. Vieira, M., Richardson, D. J., 2002. The Role of Dependencies in Component-Based Systems Testing and Evolution, Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), Orlando, USA, pp: 62-68.

145. Vigedar, M., Kark, A. W., 2006. Maintaining COTS-Based Systems: Start with the Design, Proceedings of the Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems, pp: 1-11.

146. Voas, J., 1998. Maintaining Component-Based Systems, IEEE Software, Vol. 15, Issue 4, pp: 22-27.

147. Voas, J., Agresti, W., 2004. Software Quality from a Behavioral Perspective, IEEE Computer Society, IT Professional, Vol. 6, Issue 4, pp: 46-50.

148. Wang, A. J. A., 2002. Reuse Metrics and Assessment in Component-based Development, Proceedings of 6th IASTED International Conference on Software Engineering and Applications (IASTED, 2002), pp. 583-588.

149. Washizaki, H., Hirokazu, Y., Yoshiaki, F., 2003. A Metrics Suite for Measuring Reusability of Software Components, Proceedings of the 9th International Sy mposium on Software Metric, pp: 211-223.

150. Weyuker, E. J., 1988. Evaluating Software Complexity Measures, IEEE Transactions on Software Engineering, Vol. 14, Issue 9, pp: 1357-1365.

151. Widrow, B., Rumelhart, D. E., and Lehr, M. A., 1994. Neural Networks: Applications in Industry, Business, And Science. Communications of the ACM, Vol. 37, pp: 93-105.

152. Woodman, M., Benediktsson, O., Lefever, B., Stallinger, F., 2001. Issues of CBD product quality and process quality, Proceedings of the 4th Interna tional Workshop of Component-Based Software Engineering at 23rd International Conference on Software Engineering, pp:56-68.

153. Won, K., 2005. On Issues with Component-Based Software Reuse, Journal of Object Technology, Vol. 4, Issue 7, pp: 45-50.

154. Wu, Y., Offutt, J., 2003. Maintaining Evolving Component-Based Software with UML, Seventh European Conference on Software Maintenance and Reengineering (CSMR'03), Benevento, Italy, pp: 133-142.

155. Yau, S. S., Dong, N., 2000. Integration in Component-Based Software Development

Using Design Patterns, Proceedings of 24th International Computer Software and Applications Conference, pp: 369-374.

156. Yi, C., Nahrstedt, K., 2001. QoS-Aware Dependency Management for Component-Based Systems, Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing, San Francisco, USA, pp. 127-138.

157. Zadeh, L. A., 2002. From Computing with numbers to computing with words-from manipulation of measurements to manipulation of perceptions, International Journal of Applied Mathematics and Computer Science, Vol.12, Issue 3, pp: 307-324.

158. Zuse, H., 1991. Software Complexity Measures and Methods, Walter de Gruyter, Berlin – New York.

159. http://www.oreilly.com