Identification and Selection of Components using Component Coupling and Interface Metrics

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science and Engineering

By:
Navneet Kaur
(801132018)

Under the supervision of:
Ms. Ashima Singh
Assistant Professor, CSED

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

July 2013

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Identification and Selection of Components using Component Coupling and Interface Metrics"**, in the partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Ms. Ashima Singh* and refers other researcher's work which are duly listed in the reference section.

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

Navneet Kaur.

(Navneet Kaur)

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

Ashima Singh 15/7/13

(Ms. Ashima Singh)
Assistant Professor,
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

i

Acknowledgement

The key elements concentration, dedication, hard work and application are not the only essential factors for achieving the desired goals but also guidance, assistance and co-operation of people is necessary.

First and foremost, I would like to express my deep and sincere gratitude to my supervisor Ms. Ashima Singh, Assistant Professor, Computer Science and Engineering Department, Thapar University, Patiala. But no volume of words is enough to express my gratitude towards my guide. Her wide knowledge and logical way of thinking have been of great value for me. Her understanding and personal guidance have provided a strong basis for my thesis. Without her mentoring this thesis would never have been realized.

I owe my sincere thanks to Dr. Maninder Singh, Professor & Head of Computer Science and Engineering Department, Thapar University, Patiala, for motivation and providing facilities.

I wish to express my sincere thanks to all the staff members and all my friends especially Ritu, Rupinder, Jyoti, Rajni and Bandhana, who always supported and encouraged me.

I was very fortunate to have an unconditional support from my family. I thank my family, who gave me courage to get my education, supported me in all achievements throughout my life. Last but not the least; I would like to thank God for giving me inner peace and strength.

Navneet Kaur

(Navneet Kaur)

Abstract

Modern software industry is now looking forward to Component Based Software Engineering (CBSE) for maximizing the software reusability. So CBSE is becoming the modern trend for software development. CBSE is focused on assembling the existing components to build a software, with the potential benefits of reduced development cost, time and improved quality. It differentiates itself from the conventional software development approach by concentrating highly on component integration to build a software rather than developing the software from scratch each time. The quality of the Component Based Software (CBS) depends upon the complexity of composed components. Thus evaluation of component complexity becomes a critical activity during component selection in Component Based Software Development (CBSD). So many researchers have proposed various complexity metrics for measuring component complexity during components selection for CBS. But many of the existing complexity metrics are not suitable and sufficient for measuring component complexity in the unavailability of source code or internal details of components.

In this thesis, two complexity metrics, one based on measuring component Interface Complexity (IC_{BB}) and another based on measuring component Coupling Complexity (CC_{BB}), have been proposed which do not depend on source code or internal details of component. These metrics have been empirically evaluated. Further, correlation study has been conducted for these metrics with the quality characteristics and other aspects. The metric IC_{BB} has been correlated with quality characteristics named portability and reusability, and it has been found that high complexity of the component interface leads to less portability and reusability of component. The metric CC_{BB} has been correlated with the effort for replacement or modification and testing effort. The positive relation between them shows that as the component coupling complexity increases effort for component replacement or modification, and testing effort also increases.

Table of Content

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Over a few decades, software systems are becoming comparatively larger and more complex than before. Use of the traditional software development approach to this new situation may result in problems like failure to meet deadline, budget and quality requirements. It was realized that by developing software products each time from scratch, it would never be possible to overcome these problems. Thus the concept of software re-usability was developed which provides the idea of Component Based Software Development (CBSD). Now CBSD is becoming a popular and effective approach for software development, which uses reusable components as building blocks for constructing software systems. CBSD provides advantages like reduced development time, cost , effort and increased quality along with many others [1]. But CBSD provides one of the central problems in measuring complexity of component and Component Based Software (CBS) system, as measuring and controlling the software complexity is an important aspect of every software development paradigm. The complexity of CBS system depends upon the complexity of the used components. So it is very necessary to select the appropriate less complex components for the CBS system. Thus, as the number of components available on the market increases, it becomes more important to devise metrics to quantify the various characteristics of components, upon the basis of which a suitable component may be selected for CBS system. Many researchers have proposed various types of complexity metrics based on different aspects. But many of the existing complexity metrics are not suitable and sufficient for measuring component complexity, in unavailability of source code or internal details of components.

In this thesis, two component complexity metrics have been proposed, named Interface Complexity metric for Black Box component (IC_{BB}) and Coupling Complexity metric for Black Box component (CC_{BB}), for assessing the component complexity in terms of component's Interface Complexity and Coupling Complexity respectively. These metrics have been empirically evaluated. Further IC_{BB} metric has been correlated with the quality

characteristics named portability and reusability, and CC_{BB} metric has been correlated with the aspects like effort for replacement or modification, and testing effort.

1.2 What is Component Based Software Engineering?

Component Based Software Engineering (CBSE) is a modern approach for software system development. CBSE focuses on the design and construction of software systems using reusable software components. This principle embodies the "buy, don't build" philosophy which shifts the emphasis from programming software to composing software systems [2][6].

The main focus of CBSE is on:

- Reusing existing software components to build a software system.
- Reducing the development time and cost.
- Increasing the productivity and quality.

But CBSE provides one of the central problems in measuring and controlling the software complexity.

1.3 Software Complexity

Measuring and controlling the software complexity is the main objective of each software development paradigm. The first problem encountered when attempting to understand software complexity is to define what it means for a software to be complex. Software complexity has been studied for over many years and lots of measures have been proposed to capture different aspects of software complexity however, there is no consensus about what software complexity actually is. There are many definitions of software complexity given by different researchers, some of them have been given below:

- 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 " [41].
- Zuse [4] defined software complexity as the difficulty in maintaining, changing and understanding software. It deals with the psychological complexity of programs.
- Basili [19][4] defined software complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task.

The interacting system may be a machine or human being. When a computer system acts as an interacting system, complexity is defined in terms of execution time and storage required for performing the computation. And if the interacting system is a human being (programmer), then complexity is defined by the difficulty of performing tasks such as coding, testing, debugging or modifying the software.

- Bill Curtis [21] has reported two types of software complexity, these are as below:

  Psychological Complexity: It affects the performance of programmers trying to comprehend or modify a class/module.

  Algorithmic Complexity: It is also called computational complexity and it characterizes the run-time performance of an algorithm.

Thus there is no standard definition exists for software complexity in literature. Because software complexity cannot be defined by a single definition because it is a multidimensional attribute of software and different researchers/users have different view on software complexity [19]. The knowledge about software complexity is useful in many ways. It is an indicator of development, testing and maintenance efforts, defect rate and reliability. Complex software or module is difficult to develop, test, debug and maintain. So the study of software complexity is an important aspect during each software development paradigm. But for measuring and controlling the software complexity effectively, software complexity metrics are required to measure it.

1.4 Software Complexity Metrics

Metrics are defined as "Quantifiable measures that could be used to measure characteristics of a software system or the software development process"**.** Software complexity metrics are used to measure the software quality to check whether it satisfies the requirements. Software complexity metrics are essential to plan, predict, monitor, control and evaluate the products. The main goals of software complexity metrics are to reduce costs, improve quality and reduce testing effort.

In order to measure and control the software complexity, researchers have proposed a wide range of software complexity metrics based on different aspects, but in case of CBSD many of the existing software complexity metrics are not suitable and sufficient for measuring the component complexity, due to unavailability of source code or internal

details of components in most of the times. In our research work, we have proposed two component complexity metrics for measuring the component complexity in terms of component's interface complexity and coupling complexity.

1.5 Need of Study

Measuring and controlling software complexity is an important objective during each software development paradigm. Software complexity measure can be used as a predictor of the effort that is needed to develop, test or maintain the system. This measurement could direct the process of improvement and reengineering work.

In CBSD approach, the complexity of CBS system depends upon the complexity of used components. Thus the complexity of component should be determined before using it, by using complexity metrics. The complexity of component affects the many other aspects like reusability, portability, testability and ease of replacement or modification of component. The selection of less complex components having more portability, reusability, testability and ease of replacement or modification, will help in providing a less complex CBS system with good quality. And it will also help in reducing the integration effort and testing effort for the resulting system. But for measuring the component complexity there is a need of suitable metrics. But many of existing complexity metrics are not suitable for measuring component complexity, as they depend upon the source code or internal details of component, which may not be available in case of black box components. So there is a need of appropriate complexity metrics for measuring component complexity for the selection of appropriate component for CBS system. In this thesis, two component complexity metrics have been proposed, which do not use source code or internal details of component for measuring component complexity during component selection for CBS system.

1.6 Objectives of the Study

- The detailed study of Component Based Software Engineering (CBSE), various terms and concepts of CBSE, CBSE process and various process models for CBSD.

- The study of various software complexity metrics, which may be applied for measuring component complexity .
- Proposal of component complexity metrics appropriate for selecting potential candidate components for reusability.
- Empirical evaluation and validation of the proposed metrics for component selection.

1.7 Outline of the Thesis

This thesis has six chapters whose outlines are provided below:

Chapter one describes the motivation for this thesis. It provides the overview of the thesis, brief introduction of CBSE, description of software complexity and its importance, description of software complexity metrics and their importance and objectives of study. It highlights the need of suitable metrics for measuring component complexity and their importance in CBSD.

Chapter two is a study of Component Based Software Engineering, which provides an explanation of the CBSE approach, its purposes, its terms and concepts, the process used by CBSE, the process models used for CBSD, some guidelines for component developers and users, and various benefits of CBSE.

Chapter three is a survey on component complexity metrics, which describes the need of appropriate complexity metrics for components. In this chapter various complexity metrics, which are relevant for measuring component complexity for component selection, have been discussed with their limitations and a conclusion has been provided.

Chapter four provides the problem definition which is to be solved. It also describes the various challenges for solving the problem and scope of study.

Chapter five describes the proposed component complexity metric used for measuring component interface complexity. This chapter provides the explanation of the proposed metric, theoretical evaluation of proposed metric, empirical evaluation and validation of the proposed metric with respect to quality characteristics portability and reusability.

Chapter six describes the proposed component complexity metric used for measuring component coupling complexity. This chapter provides the explanation of the proposed metric, theoretical evaluation of proposed metric, empirical evaluation and validation of

the proposed metric with respect to software aspects like effort for replacement or modification, and testing effort.

Then the conclusion and future scope of the thesis work has been provided.

Chapter 2

Component Based Software Engineering

2.1 Introduction

Software is the heart of many industrial systems in today. But over a few decades, software systems are becoming comparatively larger and more complex than before. Use of the traditional software development approach to this new situation may result in problems like failure to meet deadline, budget and quality requirements. It was realized that by developing software products each time from the scratch, it would never be possible to overcome these problems. Thus the concept of software re-usability was developed which provides the idea of Component Based Software Engineering (CBSE). Now CBSE has become a modern approach for software development.

CBSE is a process that focuses on the design and construction of computer-based systems using reusable software components. This principle embodies the "buy, don't build" philosophy which shifts the emphasis from programming software to composing software systems [2][6].

The main purpose of CBSE is to develop the large software systems from existing software components, thus reducing the development time and cost, and increasing the productivity and quality. These advantages are mainly provided by the reuse of already developed software components [1].

If one is familiar with Object Oriented Programming (OOP), then it can be useful to think of CBSE in a similar way. In OOP, objects are reusable entities that could be connected together into programs. These objects are often contained in vast libraries of reusable code. Similarly the development of component-based systems starts with a collection of existing components. The components are integrated into the system with some proprietary code that glues the components together. This code is also known as glue code. However, OOP is not sufficient for CBSE. Because OOP does not express the "uses" relationship, which is needed for CBSE. Object Orientation express "has a" and "is a" relationships. Components express the context within which they will work by declaring different system resources required for the component to work properly. But

this type of concept is not typically supported by Object Orientation. A component does not have to be an object, it can be a function or an executable program that is not treated as an object. But Orientation technology can successfully be used for development of components.

2.2 Component Based Software Engineering

Component Based Software Engineering is a software engineering paradigm in which software systems are developed by integrating existing components. Thus the main focus of CBSE is at realizing long waited software reuse by changing both software architecture and software process. This approach has emerged from the failure of object oriented development to support effective reuse. Single object classes are too detailed and specific. But components are more abstract than object classes and can be considered to be standalone service providers. The CBSE process is quite different from that of the traditional waterfall approach, due to extensive use of components [3]. In CBSE, the component selection and evaluation are special life cycle phases and much effort is required in the selection of components, testing and verification phases. This chapter presents a study of CBSE. The aim of this study is to help in providing a better understanding of different CBSE terms and concepts [5][7][8].

2.2.1 Component

A software component is a system element offering a predefined service or event, and able to communicate with other components [9]. The component can be used from the outside of it via the interfaces. To component users, a component is a self-contained unit that can be used for a specific purpose. The internal implementation of a component is usually hidden from its users. Components are not only reused within organizations to which the components' developers belong, but are also distributed in the form of an object code via the Internet and reused in other environments. The component is written with reuse in mind so that it can be deployed, with little or no modification, in other applications also [7]. But users who want to reuse components often cannot obtain source codes of the components except for object codes. There are many definitions of software components given by different researchers, some of them have been given below:

- Clement Szyperski defined software component as a unit of composition with contractually specified interfaces and explicit context dependencies, which can be deployed independently and is subject to composition by third party [8][10].
- Councill and Heinmann defined software component as a software element that conforms to a component model and can be independently deployed and composed without modifications according to a composition standard [8][5].
- Wallnau collected different definitions and compiled them down to four definitions that are representative of those that are emerging in the software industry and that contain different concepts of components [3]. Author defined :
    1) A component as a replaceable part of a system that fulfills a clear function.
    2) A run-time component.
    3) A component as a unit of composition with contractually specified interfaces.
    4) A business component as a software implementation of an "autonomous" business concept.

The components may be developed in house, but there is also the possibility to buy software components from component vendors, so called Commercial - Off - The-Shelf (COTS) components [12]. The use of COTS software components is increasing in today's development of new systems. Using COTS components can be one way of reducing development time and be competitive by getting products to the market fast and inexpensively [11]. Development with COTS components has many advantages like functionality is instantly accessible, components may be less costly, and components may have been developed by experts in the area. Developers that want to use existing COTS components face the problem of determining the exact functionality and quality of these components [13]. They want to have some assurances that the component functionality corresponds to the expected functionality. In addition, developers want to have some assurances of component's quality. Developers must have good understanding of COTS component in order to integrate them properly with the system under development. However, COTS components are typically treated as black boxes because only a brief component description is available and the source code for the component is not available. Once the component is accepted for integration, these conflicts, or mismatches,

must be repaired through component adaptation. Only when these mismatches are removed, it is possible to integrate the component into the system.

2.2.2 Reusability

Reusability is the likelihood of a segment of source code that can be used again to add new functionalities with slight or no modification [44]. The implementation time can be reduced by using the reusable modules. In the case of component based development the applications can be built from existing components by assembling and replacing interoperable parts. Thus a single component can be reused in many applications that will result in providing a faster development of applications with reduced cost and high quality.

When a component is developed it is tested with respect to specific application domain. But they may fail in the new environment. So there are so many cases where reuse may lead to the software failure. This may occur because of the lack of experience for reuse, lack of documentation, tools and methodology for reuse along with several others.

Mainly, there are two types of component-based reuse: with no change to an existing component and with change [47]. In the case of reuse without change, simply a component is selected from a software component database and used into new software being developed. But in some cases there is need to implement the change in component. The main reason is due to the difference in functionality. But reuse a component with change is also a very difficult task because it is difficult to identify those parts of the components that require changes. After implementing the changes there is a need to test the modified components thoroughly before they are plugged into the system.

Another characterization of software reuse is the way it is implemented. First is known as white-box reuse in which developers usually have the access to the code that can be modified to include the new demands of the application [46]. Thus the developer can modify the component to fit in the target system, thus this type of reuse maximize the reuse opportunities. But this approach has one limitation also. Because for code modification there is a need of high level of familiarity with the implementation details. Another type of reuse is known as black-box reuse. In this approach the component is used as it is and developer does not have any access to the source code [46][52]. So with

the black box component reuse, it is difficult to determine the reusability of the component. The developer can only rely on the specifications. In this category component must be flexible enough for better reusability.

2.3 Component Based Software Engineering Process

CBSE process deviates from the conventional software development process in a way that it is integration centric as opposed to development. In this approach much effort is devoted in selecting, qualifying and testing of components for Component Based Development. Thus the main steps for CBSE process are: 1) Finding the components which may be used for system development, 2) Qualifying the components that fit the requirements of system, 3) Adapt the selected components so that they suit the existing component model or requirement Specification, 4) Compose and deploy the components using a framework for components and 5) Replace earlier versions with later versions of components for maintenance purposes.

As shown in the figure 2.1, CBSE process encompasses two parallel engineering activities [6][14]. These activities are:

- Domain Engineering
- Component Based Development

2.3.1 Domain Engineering

Domain Engineering focuses on identifying, constructing, cataloguing, and disseminating a set of software components having applicability to existing and future software in a particular application domain. An application domain may be defined as a product family which is the set of applications with similar functionality or intended functionality. The objective of domain engineering is to establish a mechanism to enable the software engineers for sharing these components to reuse them during work on existing and future systems.

As defined by Paul Clements, domain engineering is about finding commonalities among systems to identify components that can be applied to many systems and to identify program families that are positioned to take fullest advantage of those components.

Some examples of application domains are as follow:

- Air traffic control systems
- Defense systems
- Financial market systems

Domain engineering begins by defining the domain to be investigated. This is achieved by analysis of the existing applications and consultation with experts of the type of application you are going to develop.

Then identification of operations and relationships, which recur across the domain, is performed in order to realize a domain model and therefore being candidates for reuse. This domain model assists the software engineers for identifying and categorizing components, which will be subsequently implemented.

Structural Modeling is a particular approach to domain engineering, which is a pattern based approach work under the assumption that every application domain has repeating patterns. These patterns may be in anything having reuse potential like function, data or behavior. This approach has similarity with the pattern based approach in OOP, in which a particular style of coding is reapplied in different contexts.

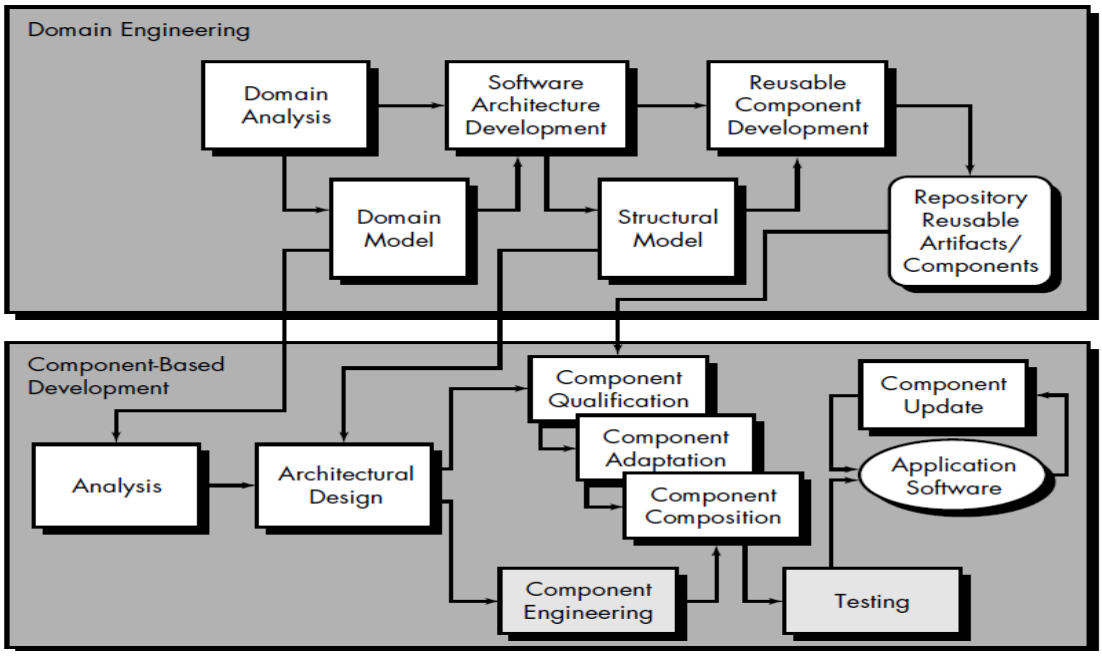

Figure 2.1: Component Based Software Engineering Process [6]

2.3.2 Component Based Development

Component Based Development (CBD) is a CBSE activity that occurs in parallel with domain engineering. Once the architecture has been established, it must be populated by components. When reusable components are available for potential integration into the architecture, they must be qualified and adapted. But when new components are required, they must be engineered. The resultant components are composed (integrated) into the architecture template and tested thoroughly. Thus CBD encompasses the two processes:

a) Assembling software systems from reusable software components
b) Component Engineering

These two processes have been described below:

a) Assembling software systems from reusable software components

Unfortunately, the existence of reusable components does not ensure that these components can be integrated easily or effectively into the architecture chosen for a new application. So a sequence of component-based development activities is applied when a component is proposed for use. These activities are:

- Component Qualification
- Component Adaptation (also known as wrapping)
- Component Composition

The brief description of these activities has been given below:

- **Component Qualification**

  Component qualification examines reusable components. This activity ensures that a candidate component will provide the required functionality and it is suitable for properly fitting into the architectural style specified for the system. The three important characteristics looked at are performance, reliability and usability. The interface description is used to provide useful information about the operation and use of a software component. But this does not always provide the whole picture of whether a component will fit the requirements and the architectural style. This is a process of discovery by the Software Engineer.

- **Component Adaptation**

  The individual components are written to meet different requirements, each one making

certain assumptions about the context in which it is deployed. Thus adaptation is required because very rarely components will integrate immediately within the system. The goal of adaptation is to ensure the minimization of conflicts among components. The different approaches for adaptation (also known as wrapping) have been proposed depending upon the accessibility of the internal structure of a component. These approaches are:

- **White box wrapping:** It is applicable only, when a software team has full access to the internal design and code for a component. In this case the implementation of the component is directly modified in order to resolve any incompatibilities. But it is extremely unlikely to be used in the case of COTS because of unavailability of source code and internal details.
- **Grey box wrapping:** It is applied, when the component library provides a component extension language or API that enables conflicts to be removed or masked.
- **Black box wrapping:** It is applicable in the case, where access to source code is not available. It requires the introduction of pre- and post processing at the component interface to remove or mask conflicts.

The software engineer should determine whether the effort required for component adaptation adequately is justified or whether a custom component should be engineered instead, in order to remove the encountered conflicts. After the component adaptation it must be checked for compatibility for integration and tested for any unexpected behavior emerged due to the modifications made.

- **Component Composition**

The component composition activity assembles the qualified, adapted, and engineered components to populate the architecture established for an application. This is accomplished by establishing an infrastructure for binding the components into an operational system. This infrastructure is usually a library of specialized components itself. A model is provided by an infrastructure for the coordination of components and specific services that enable components coordination with one another and perform common tasks.

b) Component Engineering

The CBSE process encourages the use of existing software components. But sometimes the components must be engineered which means new software components must be developed and integrated with existing COTS and in-house components. These components should be engineered for reuse as these new components become members of the in-house library of reusable components.

2.4 Component Based Software Development Process Models

Several models have been developed for assuring the efficient software development using Component Based Software Development approach. A brief discussion of some of the models is given below:

2.4.1 Somerville's Model

Somerville has provided a sequential approach for CBSD [16][17], as shown in figure 2.2. There are mainly six phases which include:

- **Outline System Requirements:** The user requirements are developed in outline rather than in detail as specific requirements limit the number of components that might be used.
- **Identify Candidate Components:** This phase is concerned with identifying as many components as possible for reuse by using a complete outlined set of requirements.
- **Negotiate Requirements:** In this phase requirements are refined and modified in order to comply with components.
- **Architectural Design:** This phase is concerned with developing an architectural design.
- Requirements can be negotiated for incorporating architectural compatible components. After system architecture is designed, steps 2 and 3 may be repeated.
- **Integration:** Finally the chosen components are integrated to get the software system.

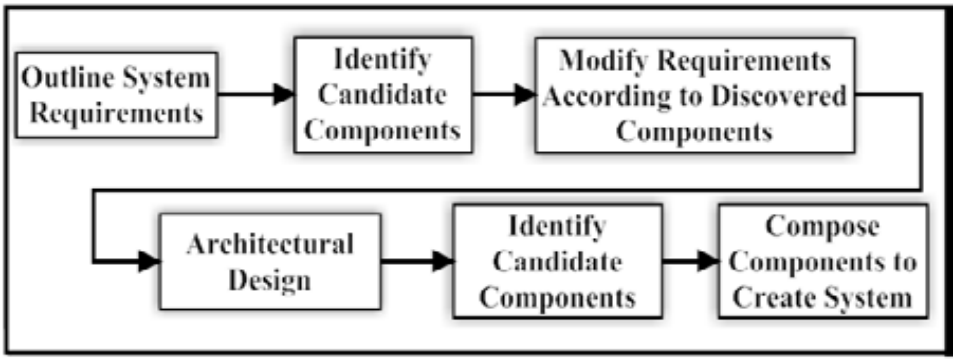

Figure 2.2: The CBSD process model proposed by Somerville [17]

2.4.2 The Y- Model

The Y model consists the basic activities of software development process such as system analysis, design, implementation, testing, deployment and maintenance as such [16][17]. But some additional activities have been added to support the component based software development as shown in figure 2.3.

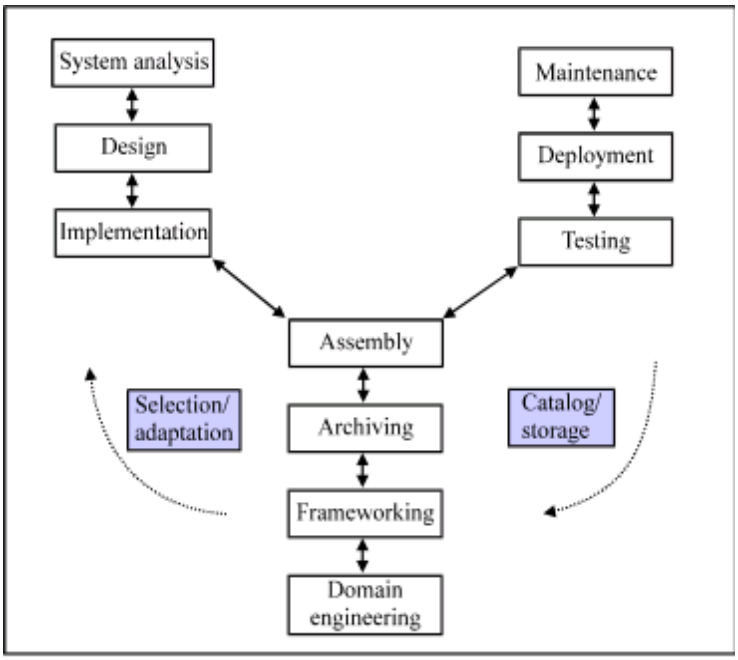

Figure 2.3: The Y Model [16]

The additional activities added to support CBSD are:

- **Domain Engineering:** Domain engineering is concerned with identification of potentially reusable components for a particular domain.

- **Frameworking:** A framework is a skeleton or a template used for producing software in an application domain. It captures the semantic relationships between the components of a particular domain. The objective of the frameworking phase is to reuse the already developed software components and classifying them further to form new frameworks.
- **Assembling:** Assembling is concerned with composing the existing reusable software components for developing the software application.
- **Archiving:** Archiving is concerned with archiving the components developed for a particular software application for future use in related applications. This involves activities such as component cataloguing and storage.

2.4.3 The V- Model

The V model is an adaptation of the traditional waterfall model for building a system from pre-existing components [15][16][20]. As shown in figure 2.4, it defines a sequential process consisting of following phases:

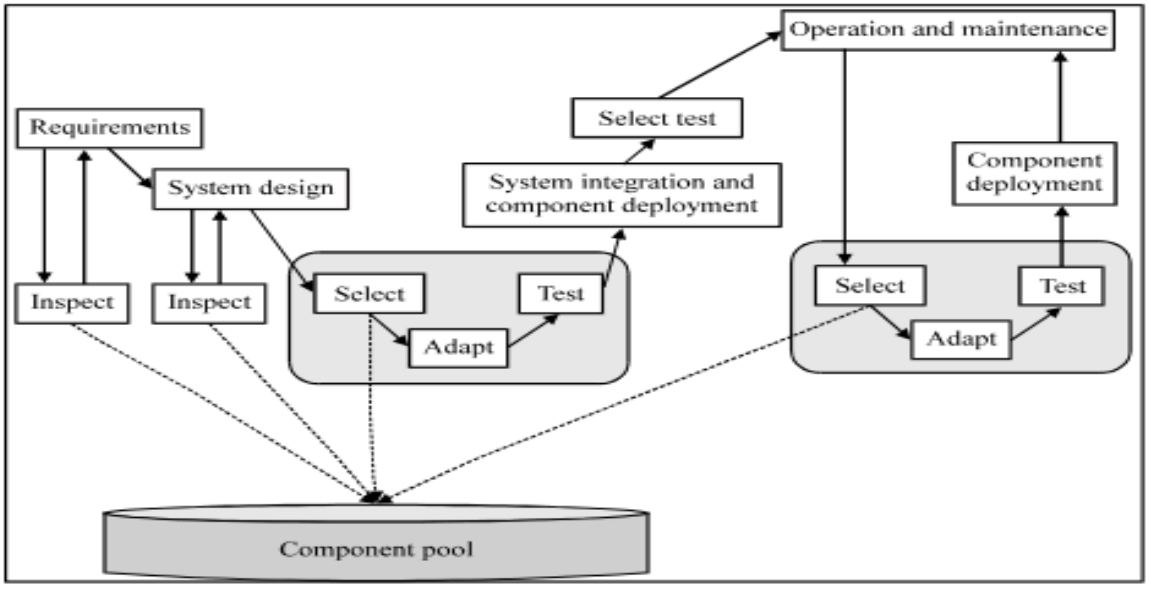

Figure 2.4: The V Model [20]

- **Requirements Analysis and Specifications:** This phase is concerned with analyzing and specifying requirements keeping in mind the available components.

If possible, requirements can be negotiated for making the use of available components.

- **System and Software Design:** Component pool is again available in this phase. The components selected during first phase may be rejected if they do not fit into the overall design of software. Alternatives may be selected from the component pool.
- **Implementation and Unit Testing:** An ideal case for application development is to build it from the direct integration or connection of component interfaces. But in practice, some glue code or component wrappers need to be written in order to bridge component interface mismatches. Sometimes new functions need to be written for filling gaps in system requirements and component capabilities. Then these modules are tested separately.
- **System Integration:** The integration process is concerned with integration of standard infrastructure components which build a component framework and application components.
- **System Verification and Validation:** Standard tests and verification techniques are used here.
- **Operation Support and Maintenance:** In order to support the changing requirements, new components may have to be added or existing components may have to be replaced/modified in the operational system. The maintenance process consists of deploying the new or modified components in the system, changes in glue code or replacing the troubling components.

2.5 Guidelines for Component Developers

In order to generate and provide more reusable components, the following points must be considered by the developers [3][18]:

- **Independency or Less Dependencies:** The component must be generated as an independent component if possible, or the component must be generated in such a way that it has less dependencies on other components of system. It will help in reducing the integration effort and increasing the maintainability of resulting system.

- **Documentation:** The importance of documentation in reuse is critical. A potential user needs accurate information about a component in order to align the component with a requirement. So the developer should provide appropriate and unambiguous documentation, including all the features of the component. Do not restrict the documentation to functionality, document all other properties as well like performance, resource consumption, limitations, robustness, etc.
- **Test Suites:** Appropriate test-suites should be provided with the component so that the component users can test component for the environment in which the component is to be used. It is extremely important to test an imported component in the environment in which it will operate.
- **Generalization:** Components need to be carefully generalized to enable reuse in a variety of contexts. However, it will take more effort for solving a general problem rather than a specific one.
- **Certification of Quality:** Reuse requires some blind faith on the part of the component user that the component being used will be as suitable and reliable as documented. For a reuse initiative to succeed, the component should be certified for its quality.
- **Source Code:** Source code should be provided if possible, it might help the application developer to understand the semantics of component.

2.6 Guidelines for Component Users

There are some guidelines for the component users which must be considered by them while making a decision for buying the component from the component vendors [3]. These guidelines are:

- A thorough evaluation of the component suppliers. Are they suitable as a supplier? Do they have good quality products and support? Check their economy so they don't easily bankrupt.
- Put a lot of effort into the legal agreement with the supplier. This may save component user if the supplier goes out of business or if they refuse to support them.
- Create good and long term relations with the supplier for better cooperation.

- Assign key persons for supervising the component market. They shall keep track of new components and trends.
- Adjust the development process to a component based process.
- If possible, try to get access to the source code.
- Test the components thoroughly in the environment in which they are being used.

2.7 Benefits of CBSE

The main objectives of CBSE are to reduce the development cost and time for building large and complex software systems, and increase the quality of the software. One important paradigm shift implied here is to build software systems by using existing software components rather than generating them from scratch each time. This requires thinking in terms of system families rather than single systems. Some benefits of CBSE have been discussed below [5][9]:

- Reusability is the main benefit provided by CBSE. Because once the component has been developed, it can be used in another applications also. But domain engineering should be kept in mind.
- Product delivery schedules are reduced because shorter development cycles would save time as to developing.
- The development costs are reduced since existing components are used to develop the systems.
- The reliability is increased since the components have previously been tested in various contexts.
- The efficiency and flexibility is improved due to the fact that components can easier be added or replaced.
- The maintainability of enterprise software systems is improved by allowing new, higher-quality components to replace old ones.
- The quality of software systems is improved as the application-domain experts develop components, then software engineers who specialize in component based software engineering assemble those components into enterprise software systems.

- Lets developers focus on their business requirements and core competencies, rather than re-solving the same technical problems over and over.
- Higher level models make complex systems easier to understand: component based development is the best technique for managing complexity of systems as they increase in size and scope.

Chapter 3

A Survey on Component Complexity Metrics

3.1 Introduction

Measuring and controlling the software complexity is an important aspect during each software development paradigm. Because the complexity of software affects many other attributes like testability and maintainability etc [19]. So the researchers proposed many software complexity metrics, but most of the metrics are based on the source code of the software. But nowadays Component Based Software Development (CBSD) is becoming the trend for software development. This approach is based on constructing the software system by integrating the prefabricated software components. The quality of resulting system depends upon the complexity of the composed components. So evaluation of component complexity is a critical activity in the component selection process for CBSD. So for the evaluation of component complexity suitable metrics must be used.

The metrics may play an important role in quality assurance specially in the acquisition of components and in deciding whether they should be used or not. But metrics should provide a basis for deciding whether reuse is sensible or not. In this chapter the various complexity metrics relevant for measuring the component complexity have been discussed, with their limitations and a solution has been proposed.

3.2 Study of Various Existing Complexity Metrics

Software complexity cannot be removed completely but it can be controlled. But, for controlling the software complexity effectively, software complexity metrics are required to measure it. So many researchers have proposed various metrics for evaluating and predicting software complexity. This section describes various metrics which may be applied for measuring component complexity.

3.2.1 Traditional Software Complexity Metrics

Traditional software complexity metrics have been designed and applied for measuring the software complexity of structured systems since 1976. Among these metrics

developers often found that Lines of Code, McCabe's Cyclomatic complexity metric, Halstead's complexity metric and Henry's & Kafura's fan-in, fan-out are most commonly used metrics [4]. The brief description of these metrics has been given below:

Metric 1: Lines of Code (LOC)

LOC metric is based on the size of methods. This metric gives measure of physical lines, statements, and/or comments. It is used for size estimation. High value of this metric shows more complexity [22] [23].

Metric 2: McCabe's Cyclomatic Complexity Metric

McCabe's Cyclomatic complexity metric is based on the program graph and is defined as [25]:

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

Where e, n and p represent the number of edges, number of nodes in the graph and no. of connected nodes respectively. This metric gives the measure of independent algorithmic test paths. More independent paths mean more testing effort.

Metric 3: Halstead's Complexity Metric

Halstead's complexity metric attempts to estimate the programming effort [22][24]. It measures complexity by summarizing the number of operators and operands contained in a program. The measurable and countable properties are:

- n1 = number of unique or distinct operators appearing in the implementation
- n2 = number of unique or distinct operands appearing in the implementation
- N1 = total usage of all of the operators appearing in the implementation
- N2 = total usage of all of the operands appearing in the implementation

Then the vocabulary, n of the program is defined as:

$$n = n1 + n2$$

The implementation length, N of the program is defined as:

$$N = N1 + N2$$

From the length and vocabulary, the volume, V of the program is defined as:

$$V = N \log_2(n)$$

The difficulty, D of the program is defined as:

$$D = (n1 * N2)/(2 * n2)$$

And effort, E is defined as:

$$E = D * V$$

Metric 4: Henry's and Kafura's Metric

Henry and Kafura also proposed the complexity metric based on the number of local information flows entering (fan-in) and exiting (fan-out) in each procedure [25]. This metric is given as:

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

Where length is any measure of length such as Lines of Code or alternatively McCabe's Cyclomatic complexity is sometimes substituted.

All these four metrics can be applied for measuring the component complexity. But these are based on the availability of the source code of component which may not be available in the case of black box components. Thus traditional software complexity metrics are not sufficient for measuring component complexity.

3.2.2 Object Oriented Software Complexity Metrics

The most impressive findings related to Object – Oriented metrics were the one proposed by Chidamber and Kemerer. They have proposed six complexity metrics [26] [27]. These complexity metrics are:

Weighted Methods per Class (WMC)

Weighted Methods per Class metric is intended to count the combined complexity of local methods in a given class. Thus WMC is defined as the sum of complexity of a class's local methods.

Consider a class C, with methods $M_1 ... M_n$ that are defined in the class and let $C_1 ... C_n$ be the complexity of the methods. Then WMC is given as below:

$$WMC = \sum_{i=1}^{n} C_i$$

If we consider the complexity of the method as unity, then WMC is equal to the Number of Methods in a class (NOM). The greater value of this metric shows more complexity, increase in testing effort and decrease in understandability.

Depth of Inheritance (DIT)

Depth of Inheritance metric is for class. The depth of a class within the inheritance tree

is the maximum number of steps from the class node to the root of the tree. It is measured by the number of ancestor classes. High value of DIT shows high design complexity. But it represents the greater potential for reuse of inherited methods.

Response for Class (RFC)

The RFC metric gives the count of all methods that can be invoked in response to a message to an object of the class or by some method in the class. The high value of RFC shows high complexity of the class. If a larger number of methods can be invoked in response to a message, the testing of the class becomes complicated since it requires a greater level of understanding on the part of the tester.

Coupling Between Objects (CBO)

For a given class, this metric measures the number of other classes to which the class is coupled. Excessive coupling prevents reuse. The more independent a class is, easier it is reused in another application. High value of this metric shows the poor design, difficulty in understanding, decrease in reuse and increase in maintenance effort.

Lack of Cohesion Method (LCOM)

The cohesion of a class is characterized by how closely the local methods are related to the local instance variables in the class. LCOM gives the number of disjoint sets of local methods. A highly cohesive module should stand-alone and high cohesion indicates good class subdivision. It implies simplicity and high reusability.

Consider a class C_1 with n methods M_1, M_2 ..., M_n. Let $\{I_j\}$ = set of instance variables used by method M_i. There are n such sets $\{I_1\},\{I_2\}...\ \{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \Phi\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \Phi\}$. If all n sets $\{I_1\},\{I_2\}...\ \{I_n\}$ are Φ then let $P = \Phi$. LCOM of a class can be defined as below:

$LCOM = |P| - |Q|, \text{ if } |P| > |Q|$

$LCOM = 0 \text{ otherwise}$

The high value of LCOM indicates that the methods in the class are not really related to each other and vice versa.

Number of Children (NOC)

Number of Children metric is based on a node (class) of inheritance tree. This metric gives the number of immediate successors of the class. High value of NOC shows more reuse, poor design and increase in testing effort.

Although CK metric suite has been accepted widely but empirical validations of these metrics in real world software development settings are limited. Various flaws and inconsistencies have been observed in this suite of six class-based metrics.WMC metric is based on the source code and other metrics will also require internal details of component to compute complexity which may not be available in case of black box components. Thus this metric suite is not sufficient for measuring the component complexity.

Mishra [28] proposed a metric for determining the class complexity at method level by considering internal structure of the methods. Fothi et al. [29] designed a class complexity metric which is based on the complexity of control structures, data and relationship between data and control structures. But these metrics also depend upon the availability of source code. Thus these metrics are also not suitable for measuring black box component complexity.

3.2.3 Component Complexity Metrics

Many researchers have proposed various metrics for determining component complexity by considering different aspects, some of the existing component complexity metrics have been discussed below:

a) Tullio Vernazza et al. extended the CK metrics [30]. Authors proposed new metrics corresponding to each CK metric. The definitions of these metrics have been given below:

Weighted Classes per Component (WCC)

WCC is an extension for NOM, it measures weighted classes per component and number of classes. A component may consist of a group of classes and the complexity of the various classes influence the complexity of the resulting component. More complex classes means more difficulty in understanding and maintenance, and consequently the component will be complex and difficult to maintain. The WCC metric has been defined as:

$$WCC = \sum_{i=1}^{m} NOM\ (C_i)$$

Where NOM (C_i) represents the complexity of i^{th} class and m represents the number

of classes in the in the component.

Extensions for DIT: Maximum of the DIT and Mean of Unrelated Trees

The high value of DIT is used to identify classes that are hard to maintain. The effort in maintaining a group of classes can therefore be indicated by the values of DIT. But the extended metric considers the highest value of DIT, called MAXDIT, and the mean of DIT of unrelated trees (MUT). The definition of MAXDIT is:

$$MAXDIT = \max_{Ci \in K} \{DIT(Ci)\}$$

Where K is a component consists of m classes and DIT (C_i) is the DIT value of i^{th} class which belongs to the component K. The mean of DIT of unrelated trees is the mean of the values of MAXDIT for each independent class hierarchies. Formally, assume that K is partitioned in r unrelated components $U_i \ldots U_r$, with:

$$\bigcup_{i=1}^{r} U_i = K$$

U_i and U_j never sharing any super class when i is different from j. Thus MUT has been defined as below:

$$MUT = \frac{\sum_{i=1}^{r} MAXDIT(U_i)}{r}$$

Number of Children for a Component (NOCC)

Number of Children for a Component is an extension for NOC, it gives the number of children of all the classes in the component. NOCC has been defined as below:

$$NOCC = \sum_{i=1}^{m} NC(C_i)$$

Where NC (C_i) gives the number of children for class C_i and m represents the number of classes in the component. It appears that NOCC is an indicator of reuse inside the component.

External CBO (EXTCBO)

External CBO is an extension for CBO. It measures the level of coupling for a component. It gives the number of external classes coupled to the component. EXTCBO metric has been defined as:

$$EXTCBO = \sum_{i=1}^{m} e_i$$

Where e_i is the number of external classes coupled to the class C_i and m represents the number of classes in component.

Response Set for a Component (RFCOM)

Response Set for a Component metric is an extension for RFC. It gives the number of all the methods in the member classes and the methods called by those classes. RFCOM is the sum of the values of RFC for all the classes in the set. It has been defined as:

$$RFCOM = \sum_{i=1}^{m} RFC(C_i)$$

Where RFC (C_i) is the value of Response for Class C_i and m represents the number of classes in component. High value of this metric shows more component complexity.

Component Cohesion

Authors have discarded LCOM because it is not possible to verify it with the BMB properties. Instead they proposed for measuring Component Cohesion (CC), which gives the number of internal classes to which a class is coupled normalized with the number of the possible coupling relationship among the classes: m (m-1). The CC has been defined as:

$$CC = \sum_{i=1}^{m} \sum_{j=1,\, j \neq i}^{m} h(C_i, C_j) / m.(m-1)$$

Where $h(C_i, C_j) = \begin{cases} 1 & \text{if } C_i \text{ and } C_j \text{ are coupled} \\ 0 & \text{otherwise} \end{cases}$

These metrics are validated against the theoretical properties by Briand et.al. But there is no empirical validation conducted for these metrics against any industry project, thus leaving the work incomplete.

b) Salman proposed several metrics for measuring a component based system's complexity by mainly focusing on its structural complexity [36]. Author considered components, connectors, interfaces, and composition trees as main attributes that

determine structural complexity of a component based system. The different metrics proposed for these attributes are:

Component Metrics: These metrics characterize the components in a system, these metrics are: Total Number of Components (TNC) in a system, Average Number of Methods per Component (ANMC) and Total Number of Implemented Components (TNIC).

Connector Metrics: These metrics characterize the connectors in a system, these metrics are: Total Number of Links (TNL) – total number of links in the design of a system, Average Number of Links between Components (ANLC) and Average Number of Links per Interface (ANLI).

Interface Metrics: These metrics characterize interfaces in the system, these metrics are: Average Number of Links per Interface (ANLI) and Average Number of Interfaces per Component (ANIC).

Composition Tree Metrics: These metrics characterize the composition tree, these metrics are: Depth of the Composition Tree (DCT) and Width of the Composition Tree (WCT).

This metric set is validated using a set of properties proposed by Weyuker [37]. However, the proposed metrics are very basic in nature. These are based on just the total number and do not consider the complexity of individual interfaces.

c) Bertoa et al. proposed the metrics for software components to access their usability [31]. Usability is a quality criterion in ISO 9126 quality model, which covers five sub – characteristics namely, Understandability – it is the capability of the component to enable the user to understand whether the component is suitable, and how it can be used for particular tasks and conditions of use, Learnability – it is the capability of the software component to enable the user to learn the application, Operability – it is the capability of the software component to enable the user to operate and control it, Attractiveness – it is the capability of the software component to be attractive to the user, Usability Compliance – it is the capability of the software component to adhere to standards, conventions, style guides or regulations relating to usability. Out of these, only first three are considered relevant for determining software component usability. Several measurable concepts and attributes related

with these three aspects of quality are explored. But most of the proposed work is subjective and very difficult to measure on a real time application. So these metrics are not appropriate for measuring component complexity and usability.

d) Cho et al. proposed a complexity metric for software components [32]. This component complexity metric takes into account four types of complexity metrics. These metrics are:

Component Plain Complexity: It gives the sum of component elements (classes, abstract classes and interfaces), added complexity of all classes, and added complexity of all the methods of the classes.

Component Static Complexity: It gives the measure of the complexity of internal structure of a component by considering the different types of relationships in a component.

Component Dynamic Complexity: It gives the measure of complexity caused by message passing occurring internally in a component.

Component Cyclomatic Complexity: This metric measures the complexity of methods of a class by using McCabe's complexity metric.

The first three complexity metrics are used at design stage. But the cyclomatic complexity metric is applied after implementation. But all these metrics are based on the analysis of source code of component. So these metrics cannot be used for measuring the black box component complexity, as most of the times source code of these components is not available.

e) Gill and Grover proposed a complexity metric called Component Interface Complexity Metric (CICM) [33]. CICM assumes interface signatures, constraints on the interfaces and interface configurations of a software component, for determining the complexity. Thus CICM has been defined as:

$$CICM(S) = aC_s + bC_c + cC_g$$

Where C_s is the complexity contributed by interface signatures, C_c is the complexity contributed by interface constraints, and C_g is the complexity contributed by interface configurations of the software component. And a, b, and c are the respective coefficients for C_s, C_c and C_g, and are dependent on the nature of software

component and the nature of its interfaces. However, work still lacks of any empirical evaluation and validation of the proposed metric.

f) Ardimento et al. provided the study of impact of characteristics of individual components on the quality of the component based systems in which they are integrated [53]. Authors considered the following characteristics and provided the relevant metrics for them:

- Adequateness of the component with respect to the target system, which depends upon the functional coverage and compliance of component. The metrics proposed for this characteristics are:

  Functional Coverage: This metric measures the percentage of the functionality provided by the component to the component based system. It has been defined as:

  $$Fun\_Cov_i = \frac{Fun_i}{TotFunCBS}$$

  Where Fun_i and TotFunCBS represents the functionalities provided by the i^{th} component to the system and the total functionality provided by the system itself respectively, and Fun_Cov_i represents the functional coverage of the i^{th} component.

  Compliance: Compliance gives the effective usage of the component in the system. The compliance of the component is good when the functionality made available by the component to the system is fully utilized. It is calculated as follows:

  $$Compli_i = \frac{Fun_i}{TotalFunCompli}$$

  Where Fun_i represents the functionality that is actually provided by the component and TotalFunCompli is functionality that the i^{th} component made available to the system.

- Training Time, which is the measure of time required to train the people to use the component. Training Time will be less for easily reusable components. Normalized value of Training Time for the i^{th} component is calculated as:

$$NTT_i = \frac{TT_i}{\sum_1^n TT_i}$$

- Familiarity, which shows the level to which software developers understand the component through their previous experiences. An ordinal scale is used to measure it, with low, medium and high levels.
- Support, which express the support provided by the vendor in form of documents, discussion groups and forums etc. This metric is also measured on ordinal scale with low, medium, and high levels.

But these metrics, proposed for the considered component characteristics, are subjective in nature and some of them have not been empirically validated which leaves the work incomplete.

g) Rotaru et al. proposed the metrics to measure component reusability using the compose-ability and adaptability attributes of the software components [34]. Authors defined metrics for measuring these attributes based on the interface complexity of a component. Component compose-ability is defined in terms of parameters and return types of its interface methods. A component having least no parameters and no return values of the interface methods has high degree of compose-ability. The component compose-ability is defined as reverse of multiplicity of a component where multiplicity of a component is defined as the sum of the multiplicities of its interface methods. Interface method multiplicity is defined as the sum of the multiplicities of its return type and signature parameters. The return type's multiplicity is one if the interface method returns a value, otherwise it is zero. Similarly parameter's multiplicity is one if the parameter is of simple data type, otherwise parameter's multiplicity is r (r>1) if its data type is a complex one (e.g. pointer type). A component's adaptability is defined as the degree to which the component can adapt to functional as well as non-functional changes. The level of component adaptability is also affected by ease with which the environment can accommodate a component. So component adaptability is the sum of component's adaptability as an independent entity and the adaptability of the environment.

h) Boxal and Araban considered software component's interfaces for measuring their degree of reusability [35]. Authors assumed that understandability of a component

affects level of component reusability. The component's interface properties can be used for determining the understandability of a component. They also proposed some metrics on the basis of size of an interface, argument count, argument repetition scale etc. These metrics give better understanding of the properties of a component's interfaces and may help in measuring the component reusability.

3.3 Conclusion

Component Based Software Development (CBSD) is becoming the trend for software development, which is based on constructing the software system by integrating the prefabricated software components. But the quality of the resulting system depends upon the complexity of the composed components. So evaluation of component complexity is a critical activity in the component selection process. One necessary step is to define a set of metrics that offer useful and simple results for the component selection process. Thus in this chapter, various existing complexity metrics relevant for measuring the component complexity have been discussed in section 3.2.

But, as discussed above in section 3.2, most of the existing metrics are based on the availability of source code or the internal details of component which may not be available in case of black box components. Some of the existing metrics are subjective in nature and some metrics does not have any empirical evaluation and validation. So many of the existing metrics are not suitable and sufficient for measuring the component complexity.

Thus for the selection of less complex components for CBS system, there is a need of complexity metrics that can measure the component complexity without using source code or going into internal details of components.

Chapter 4

Problem Statement

4.1 Problem Definition

Measuring and controlling the software complexity is an important objective during each software development paradigm as it affects all other software quality attributes like reusability, reliability, testability, maintainability etc [19]. But CBSD provides one of the central problems in measuring and controlling the complexity of CBS system, developed by assembling the components. The complexity of CBS system depends upon the complexity of used components. Thus evaluation of component complexity is a critical activity during the component selection for CBS system. Researchers have proposed a wide range of complexity metrics for measuring component complexity. But many of the existing complexity metrics are based on the source code or internal details of component which may not be available in case of black box components, it leads to the difficulty in identification and selection of an appropriate less complex component for CBS system. So there is an acute need of complexity metrics that can measure the component complexity without using source code or internal details of components, for appropriate component identification and selection.

4.2 Challenges for Study

There are few challenges for the study. These are:

- Unavailability of source code or internal details of component.
- Incomplete and ambiguous component specification documentation.
- Knowledge about the various CBSE terms and concepts.
- Empirical evaluation of metrics and validation with respect to quality characteristics like reusability, portability, testability etc.

4.3 Scope of Study

The scope of the proposed complexity metrics is in software development organizations interested in reusing existing software assets. The development effort is wasted if the

organization is not able to reuse the existing components. By using the proposed complexity metrics, software developer can assess the complexity, reusability, portability, testability and ease of replacement or modification for components which are to be used in CBS system. It will help in providing the good quality CBS system. On the basis of these complexity metrics software developer can predict the following aspects for a component for appropriate component selection:

- Integration Effort
- Testing Effort
- Effort for Replacement or Modification

Chapter 5

The Proposed Interface Complexity Metric

5.1 Introduction

As discussed in Chapter 3, measuring component complexity is an important aspect during Component Based Software Development, but many of the existing metrics cannot be applied directly for measuring component complexity in unavailability of source code or internal details of component. In this chapter a metric has been proposed which measures the black box component complexity in terms of its interface complexity, and it is based on component interface specification. To explain and empirically evaluate the proposed complexity metric a case study has been discussed. This metric has been correlated with the quality characteristics like portability and reusability by using other metrics named Self-Completeness of Component's Parameter (SCC_p) and Self-Completeness of Component's Return Value (SCC_r).

5.2 The Proposed Interface Complexity Metric for Black Box Components

Interfaces are the access points of a component, through which a component can request a service declared in an interface of the service providing component. Interface complexity affects the component reusability. Less interface complexity helps in reducing the integration effort and results in more reusability. Mathematically, interface complexity is defined as sum of complexity of the interface methods. The complexity of a interface method depends on its nature. The nature of the interface method can be determined on the basis of number and data types of parameters and return type [9]. Rotaru et al. considered the interface methods complexity to determine the compose-ability of the component [34]. The components interfaced by methods having no return value and no parameter will have biggest compose-ability degree because it does not have any external dependency. The interface methods having no parameter value but having return value will have less compose-ability degree and the interface methods having parameters as well as return values will cause in lowest compose-ability degree.

We extended the approaches described by Rotaru et al. [34], Gill and Grover [33], and Boxall and Araban [35], while proposing a new Interface Method Complexity (IMC) metric for components. We propose that the complexity of interface method depends upon the return value's type, number and data-types of parameters, and the number of parameter incompatibilities which arise when the parameters are passed between the components. Thus a metric named IMC, for measuring interface method Complexity, has been proposed. The IMC metric has been defined as below:

$$IMC = W_r + PC(M) + Number\ of\ parameter\ incompatibilities \quad \ldots(\text{i})$$

Where W_r represents the weight assigned to return type and PC(M) is the Parameter Complexity for a Method, which is the complexity caused by parameters of a method.

$$PC(M) = \sum_{i=1}^{n} W_p(P_i) \quad \ldots(\text{ii})$$

Where $W_p(P_i)$ is the weight assigned to the i^{th} parameter of the method on the basis of its data type and n represents the number of parameters in a method.

Thus by using the mathematical definition of Interface Complexity, a metric named Interface Complexity metric for Black Box components (IC_{BB}) has been proposed for determining the interface complexity. This metric has been defined as below:

$$IC_{BB} = \sum_{i=1}^{m} IMC_i \quad \ldots(\text{iii})$$

Where IMC_i is the interface method complexity of the i^{th} method in interface and m represents the number of methods in component interface.

The interface methods can be divided in the following categories:

- Interface methods having no return value and no parameters.
- Interface methods having return value but no parameter.
- Interface methods having no return value but having parameters.
- Interface methods having return value as well as parameters.

The complexity of the interface methods can be measured on the basis of number and data types of parameters, return type and number of parameter incompatibilities. On the basis of data types of parameters and return value, and by considering the number of parameter incompatibilities for a method, some weight values will be assigned to the

Interface method, and their sum will show its complexity.

The data types can be divided in the following categories:

- Very simple includes integer, float, double, boolean etc.
- Simple includes structure data types.
- Medium includes class type and object type.
- Complex includes pointer and built in data types.
- Very complex includes user defined data types.

The methods having no return value and no parameter have been considered as simple methods and their weight value has been assumed 0.025 [43]. All other interface methods are assigned weight values on the basis of data types of parameters and return value type, and the number of parameter incompatibilities are also considered for determining the overall complexity of the interface method. The following table 5.1 represents the weight values assigned to different categories of data types for parameters and return values.

Table 5.1: The assigned weight values to the different categories of data types

Parameter Type , Return Value Type →	Very Simple	Simple	Medium	Complex	Very Complex
Assigned Weight ↓	0.10	0.20	0.30	0.40	0.50

A factor has been included in the existing approaches, that affects the complexity of interface methods. This factor is parameter incompatibility. When the components are integrated with each other, then one component may pass the parameter to the another component's method but sometimes the data type of the passing parameter may be different from the data type of parameter declared in the method to which the parameter is passed. Then there will be parameter incompatibility problem.

For example, suppose return value of one component's method is passed to the another component's method as a parameter to perform its task, but if their data types are different then there will be parameter incompatibility problem. So the return value must

be converted in the required form before passing as a parameter to second component's method (some integration code can be used to handle the parameter incompatibility). More number of parameter incompatibilities result in more difficulty for using that component interface method. It will reduce the understandability of method's behavior but it may increase integration complexity to connect the component with other components to provide accurate functionality. Thus it will be more difficult to use the component.

5.3 Theoretical Evaluation of Proposed Metric using Weyuker's Properties

Weyuker [37][38] proposed an axiomatic framework in the form of several properties for evaluating complexity aspects of software systems. These properties are:

Property1: There exist P and Q, for which |P| ≠ |Q|, where P and Q are programs or components.

Property 2: Let C be a non-negative number, then there are finitely many programs/components P for which |P|= C.

Property 3: There exist distinct components/programs P and Q for which |P|= |Q|.

Property 4: There are functionally equivalent programs/components P and Q such that |P| ≠|Q|.

Property 5: For any program/component bodies P and Q, we have |P|<=|P;Q| and |Q|<=|P;Q|.

Property 6: For program/component bodies P, Q and R such that |P|=|Q| and |P;R| ≠ |Q;R|.

Property 7: There are program/component bodies P and Q such that Q is formed by permuting the order of statements of P and |P| ≠ |Q|.

Property 8: If P is renaming of Q, then |P|=|Q|.

Property 9: There exist program/component bodies P and Q such that |P|+|Q|< |P;Q|.

These properties are evaluated for the proposed interface complexity metric, $IC_{BB,}$ as described below:

- There may exist two different components having different interface complexities, thus satisfying the first property.

- The interface complexity cannot be negative because each component will have at least one method with some functionality. Thus its complexity will always have some positive value. It confirms the second property.
- There may exist two different components with different functionality, but proposed complexity metric may have same value for them, as their methods may have same interface structure and same number of parameter incompatibilities but providing different functionality. It confirms third property.
- There may be two components having same functionality, but they may have different values for proposed complexity metric, as these components may be designed by using different concepts of programming and technologies. It validates fourth property.
- If a component is assembled with another component to get an integrated component for enhanced functionality, the complexity of these two individual components will be lesser than the complexity of the integrated component, as the functionality added by the another component will increase some complexity, which satisfies the fifth property.
- Two components may have same interface complexity as they may have same number of methods with same number of parameters having same data types, same number of return values of same data types and same number of parameter incompatibilities. However, they may be developed by using different programming methodologies and therefore when integrating with another component or in the system, both may have different integration code and implementation thus resulting in different complexities of the system in both the cases. It confirms sixth property.
- If the order of interface methods or the order of their parameters is changed, it will not change the complexity caused by interface methods, thus the proposed metric does not satisfy seventh property.
- It is obvious that renaming of a component, its methods, parameters and return values will not affect the complexity of that component or method. It confirms the eighth property.
- When any two components are assembled or integrated, there may be a need to write some more methods or code related with the integration in addition to the

existing methods. This will increase the complexity of the assembled component. It validates the last property.

5.4 Empirical Evaluation of Proposed Interface Complexity Metric using A Case Study

In order to empirically evaluate the proposed metric, we have considered a case study of Student Information (SI) system from which the students of different departments can receive information about their marks details, fee details and course details. This system has been developed by integrating the components. This system has been represented in the form of a class diagram and component diagram as shown below in figure 5.1 and figure 5.2 respectively. In figure 5.1 the class name shows the component name and the various methods provided by the component are represented by the class. And figure 5.2 represents the various components of SI system, the interfaces realized by different components and dependencies between the components. We considered only the business methods in our case study. For simplicity we have considered only one parameter in the methods.

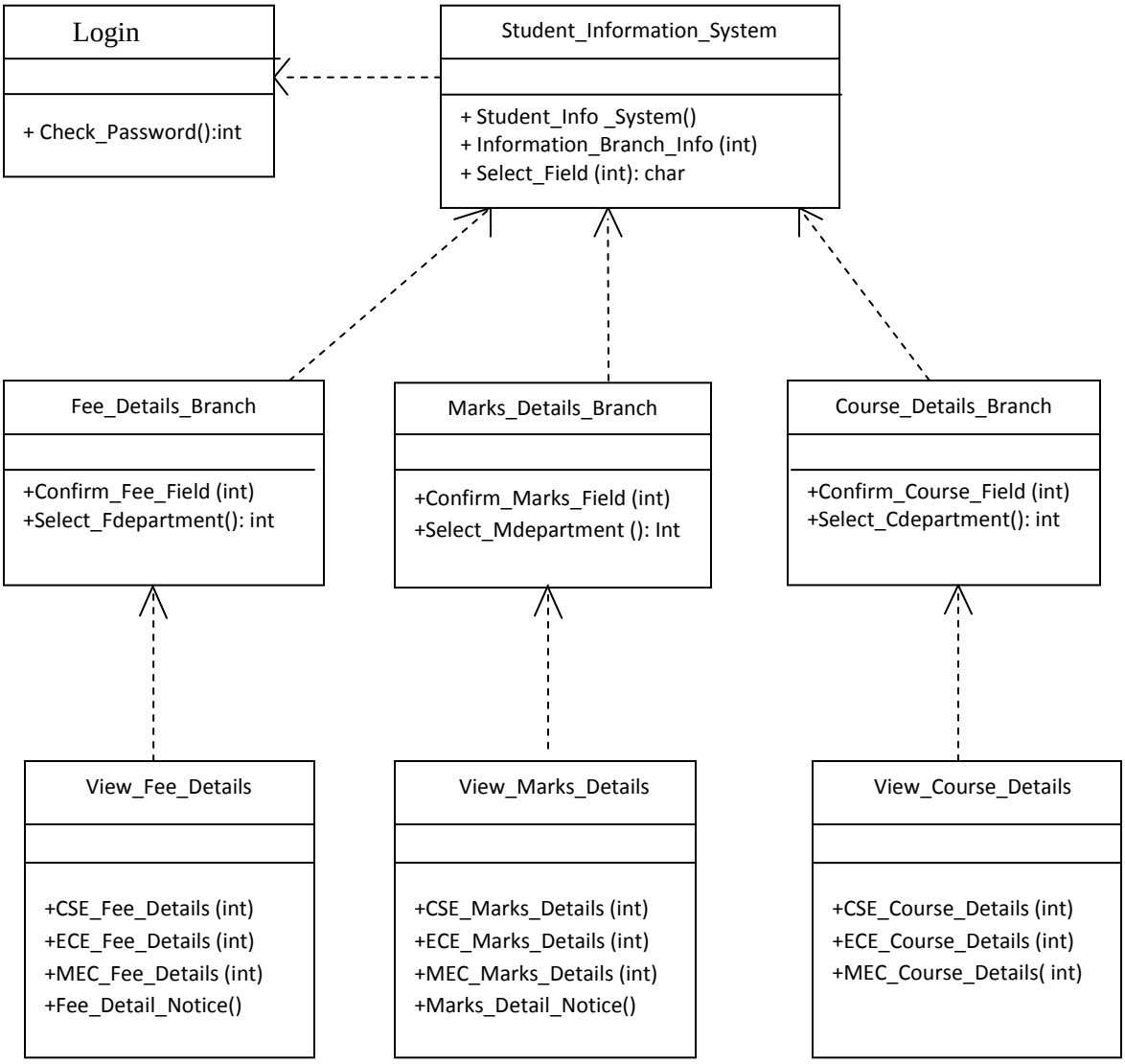

Figure 5.1: Class diagram of Student Information system

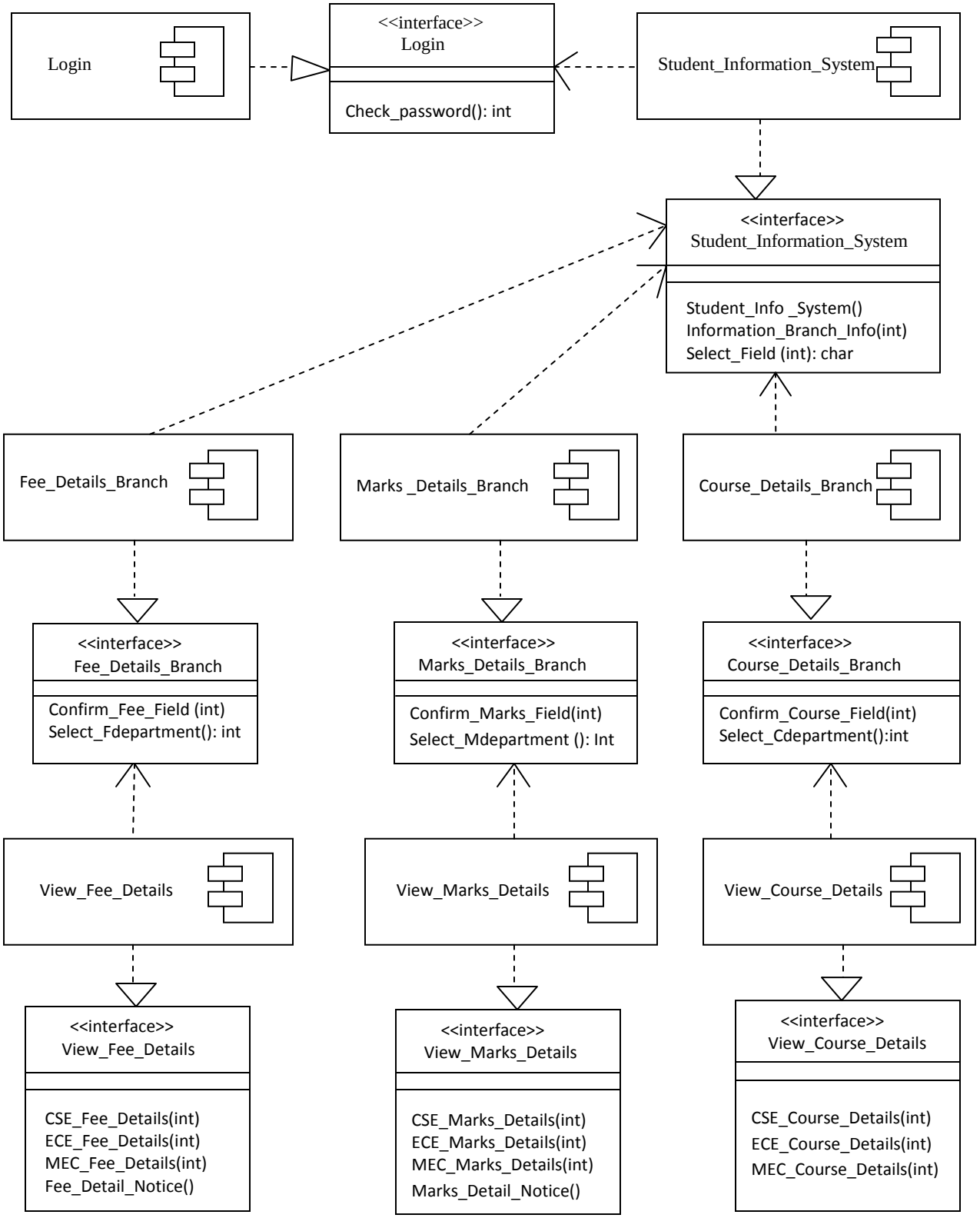

Figure 5.2: Component diagram of Student Information system

Description of the working of Student Information System

The Student Information (SI) system is composed of eight components and their working has been described as below:

- **Login Component:** This component checks the password entered by the user, in order to authenticate the user. If the password is correct then it will return value 1 otherwise it will return value 0. The return value will be passed to the second and third method of component named Student_Information_System.
- **Student_Information_System Component:** This component provides the information about the system working, information about the various information branches. The third method of this component provides the options to the user to select the branch from which the user wants to get information. The second and third method will perform their actions only if the password is correct. This component passes F, M and C, if user want to get information about fee details, marks details or course details respectively, for the confirmation of the selected branch. But the components Fee_Details_Branch, Marks_Details_Branch and Course_Details_Branch accept the integer value for the confirmation of the selected branch, as shown in their first method declaration. Thus when this component is used three parameter incompatibilities are caused which will make it difficult to use the method. It will also reduce the compose-ability.
- **Fee_Details_Branch Component:** The first method of this component confirms the selected branch from which the user wants to get information. If user has selected fee details branch to get information then it will return 1 otherwise 0. Second method of this component displays a list of departments from which the user can select any department for which the user want to check fee_details. This component will pass 1, 2 and 3 for CSE, ECE, and MEC departments respectively, to the View_Fee_Details component. When this component is used it will create one parameter incompatibility because component Student_Information_System passes the char parameter but the Fee_Details_Branch component takes the integer parameter to confirm the selected branch. Similarly in the components Marks_Details_Branch and Course_Details_Branch, the first method confirms the selected branch for the selection of marks details branch and course details branch

respectively. The second method of the components Marks_Details_Branch and Course_Details_Branch, provides the list of departments from which the user can select any department for which the user want to check the marks details or course details respectively. Each one of these components causes one parameter incompatibility as the Student_Information_System component passes the char value but the first method of these components uses the integer value, for the confirmation of selected branch.

- **View_Fee_Details Component:** 1, 2 and 3 values are passed to this component when the user wants to view fee details of CSE, ECE and MEC departments respectively. The Fee_Details_Branch component passes the integer value, for the selected department, to the View_Fee_Details component and this component also uses the integer value to confirm the selected department. Because the passing parameters and received parameters have same data types, so this component does not create any parameter incompatibility problem. Similarly in the case of View_Marks_Details and View_Course_Details components 1, 2 and 3 values are passed by the Marks_Details_Branch and Course_Details_Branch components for selecting CSE, ECE and MEC department respectively, for getting the marks details and course details information. As the passed parameters and received parameters have the same data types, so no parameter incompatibility problem is caused.

To obtain the values of proposed complexity metric IC_{BB}, this metric is applied on the each component of SI system. Different weight values are assigned to the methods, represented by the component interface, on the basis of its return type and parameters data types, and the number of parameter incompatibilities are also added as they cause the complexity in the use of component method. The IC_{BB} values, calculated using equations (i), (ii) and (iii), for each component in the SI system have been given in the form of table 5.2 as shown below:

Table 5.2: Value of IC_{BB} metric for each component in SI system

Component Name	IC_{BB}
Login	0.10
Student_Information_System	3.325
Fee_Details_Branch	1.20

Marks_Details_Branch	1.20
Course_Details_Branch	1.20
View_Fee_Details	0.325
View_Marks_Details	0.325
View_Course_Details	0.30

In order to validate the proposed metric, the other metrics named Self-Completeness of Component's Parameter (SCC_p) and Self-Completeness of Component's Return Value (SCC_r) defined by Washizaki et al. [39][52] have been used. SCC_p and SCC_r metrics are used in determining the external dependency and portability of Components. In business methods, there is a possibility that parameters and return values depend on the rest of the software which originally use the component.

Definition of SCC_p: Self-Completeness of Component's Parameter

SCC_p is the percentage of business methods with any parameters in all business methods implemented within a component c. The business methods without return value/ parameter will lead to self completeness of a component and thus lead to high portability of the component. A high value of this metric indicates a high dependency of the component on the exterior and low portability. The SCC_p is defined as below:

$$SCCp(c) = \begin{cases} \frac{B_p(c)}{B(c)} & (B(c) > 0) \\ 1 & (\text{otherwise}) \end{cases}$$

Where

$B_p(c)$: Number of business methods with parameters in c.

B(c): Total number of business methods in c.

Definition of SCC_r: Self-Completeness of Component's Return Value

SCC_r represents the percentage of business methods which return no value in all business methods implemented within a component c. It is a degree of the component's self-completeness and independence. The high value of SCC_r shows the high component portability.

$$SCC_r(c) = \begin{cases} \frac{B_v(c)}{B(c)}, B(c) > 0 \\ 1, otherwise \end{cases}$$

Where

$B_v(c)$: The business methods without return value in c.

B(c): The total business methods in c.

The following table 5.3 shows the values of SCC_p and SCC_r metrics for each component in SI System.

Table 5.3: The values of SCC_p and SCC_r metrics for each component in SI System

Component Name	SCC_p	SCC_r
Login	0	0
Student_Information_System	0.667	0.667
Fee_Details_Branch	0.5	0.5
Marks_Details_Branch	0.5	0.5
Course_Details_Branch	0.5	0.5
View_Fee_Details	0.75	1
View_Marks_Details	0.75	1
View_Course_Details	1	1

Karl Pearson's Correlation Coefficient is calculated between IC_{BB} and SCC_p, and between IC_{BB} and SCC_r in order to validate and correlate the proposed metric with quality characteristics portability and reusability.

5.5 Validation of the Proposed Interface Complexity Metric

A correlation analysis has been carried out for the proposed interface complexity metric IC_{BB} with the metrics SCC_p and SCC_r by using the Karl Pearson Coefficient of Correlation. The formula for calculating the Karl Pearson Correlation Coefficient is as below:

$$r = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{(\sum X^2 - \frac{(\sum X)^2}{N})(\sum Y^2 - \frac{(\sum Y)^2}{N})}}$$

The value of this coefficient shows the relation between two variables. The positive value of this coefficient shows the positive relation between two variables, means increase in one variable increases the value of another variable and vice versa. The negative value of this coefficient shows the negative relation between two variables, means increase in one variable decreases the value of another variable and vice versa.

The following table 5.4 shows the calculations for Karl Pearson Correlation Coefficient between IC_{BB} and SCC_p.

Table 5.4: Calculations for the Karl Pearson Correlation Coefficient between IC_{BB} and SCC_p

Component Name	X= IC_{BB}	Y= SCC_p	X^2	Y^2	XY
Login	0.10	0	0.01	0	0
Student_Information_System	3.325	0.667	11.056	0.445	2.218
Fee_Details_Branch	1.20	0.5	1.44	0.25	0.60
Marks_Details_Branch	1.20	0.5	1.44	0.25	0.60
Course_Details_Branch	1.20	0.5	1.44	0.25	0.60
View_Fee_Details	0.325	0.75	0.106	0.563	0.244
View_Marks_Details	0.325	0.75	0.106	0.563	0.244
View_Course_Details	0.30	1	0.09	1	0.30
	$\sum X = 7.975$	$\sum Y = 4.667$	$\sum X^2 = 15.688$	$\sum Y^2 = 3.321$	$\sum XY = 4.806$

The following table 5.5 shows the calculations for Karl Pearson Correlation Coefficient between IC_{BB} and SCC_r.

Table 5.5: Calculations for the Karl Pearson Correlation Coefficient between IC_{BB} and SCC_r

Component Name	X= IC_{BB}	Y= SCC_r	X^2	Y^2	XY
Login	0.10	0	0.01	0	0

Student_Information_System	3.325	0.667	11.056	0.445	2.218
Fee_Details_Branch	1.20	0.5	1.44	0.25	0.60
Marks_Details_Branch	1.20	0.5	1.44	0.25	0.60
Course_Details_Branch	1.20	0.5	1.44	0.25	0.60
View_Fee_Details	0.325	1	0.106	1	0.325
View_Marks_Details	0.325	1	0.106	1	0.325
View_Course_Details	0.30	1	0.09	1	0.30
	$\sum X = 7.975$	$\sum Y = 5.167$	$\sum X^2 = 15.688$	$\sum Y^2 = 4.195$	$\sum XY = 4.968$

By putting all the values in the formula for Karl Pearson Correlation Coefficient, we get the value of Karl Pearson Correlation Coefficient. The following table 5.6 shows the values of Karl Pearson Correlation Coefficient between IC_{BB} and $SCC_{p,}$ and between IC_{BB} and SCC_r.

Table 5.6: Correlation Coefficients between IC_{BB} and considered metrics

Characteristics	Correlation Coefficient
IC_{BB} Vs SCC_p	+ 0.072
IC_{BB} Vs SCC_r	- 0.071

As shown in above table 5.6, the correlation coefficient value between IC_{BB} and SCC_P is (+0.072) which shows the positive relation between IC_{BB} and SCC_p. It means if the value of IC_{BB} metric increases, the value of SCC_p metric also increases which shows the more possibility of external dependencies and less portability. The correlation coefficient value between IC_{BB} and SCC_r is (-0.071), which shows the negative relation between them. So as the value of IC_{BB} metric increases the value of SCC_r metric decreases that shows less portability. Thus we can say that the components having methods which have more parameters and return values will have high interface complexity and more possibility of having external dependencies, and less portability which will further reduce its reusability. The high value of IC_{BB} metric for any component shows more interface complexity, which causes decrease in portability and reusability.

Chapter 6

The Proposed Coupling Complexity Metric

6.1 Introduction

Coupling is a major factor affecting component complexity. Coupling refers to the degree of interdependence between software system components. It has been correlated to important software quality attributes such as maintainability, traceability, and robustness [41]. A coupling metric can provide the information required by developers, testers and maintainers to understand the system, identify the critical components, evaluate the impact of change in one component on the other components and even to support the future evolution of the CBS system when adding, removing and modifying some components. A large and complex CBS system should be evaluated early at the specification phase, to avoid faults, poor interaction among components and failure of one component which could lead to a total system failure. In this chapter a coupling complexity metric has been proposed, which is based on component specification. It provides the coupling complexity of a component and this metric has been correlated with factors like effort for replacement or modification and testing effort. Thus this metric can help in selecting a component, having more testability and ease of replacement or modification than other components, during the component selection phase of CBSD.

6.2 Component Coupling and Its Types

The component coupling may be defined as its degree of interaction with other components to support a specific functionality or configuration [4][42]. In CBS systems components communicate and share information in order to provide system functionalities. Components are composed on regular basis for the purpose of offering more abstract services in a system. This composition creates interactions that cause dependencies (coupling) among components. Thus the component coupling shows the dependencies between the components. In which one component passes some information to another component, i.e, one component depends upon another component. Stevens, Myers and Constantine introduced the concept of coupling in procedural

programming, in their seminal work [41][42]. Six levels of coupling based on the Myers classification were then defined. The formal definitions of these coupling classifications have been defined in the form of binary relations on a pair of system components, x and y. These coupling classifications have been represented below in order from worst to best:

- There exists a content coupling relation R_5: (x , y) ∈ R_5 if x refers to the internals of y, i.e., it branches into, changes data or alters a statement in y.
- There exists a common coupling relation R_4: (x, y) ∈ R_4 if same global variables are referred by x and y.
- There exists a control coupling relation R_3: (x, y) ∈ R_3 if x passes a parameter to y that controls its behavior.
- There exists a stamp coupling relation R_2: (x, y) ∈ R_2 if x passes a record type variable as a parameter to y and y uses only a subset of that record.
- There exists a data coupling relation R_1: (x, y) ∈ R_1 if x and y communicate by parameters, which are either a single data item or a homogeneous set of data items incorporating no control element.
- There exists a no coupling relation R_0: (x, y) ∈ R_0 if x and y have no communication, i.e., are totally independent.

This ordered coupling classification has obtained general acceptance and has formed the basis for several software metrics. The following table 6.1 represents the above discussed coupling definitions with little modifications by Fenton and Melton [41] and they assigned the coupling levels to the different coupling types upon the basis of order of Myer's classification.

Table 6.1: Modified Definitions for Myer's Classification by Fenton and Melton [41]

Coupling Type	Coupling Level	Definition
Content	5	Content coupling exists between two components if one component refers to the internals of another component, i.e., one component changes data or alters a statement in another component.

Common	4	Common coupling exists, when two or more components have read and write access to the same global data.
Control	3	Two components are control coupled if one component can directly affect the execution of another component by passing the control parameter or flag.
Stamp	2	Two components are stamp coupled if one component passes a record type variable as a parameter to another component.
Data	1	Two components are said to be data coupled if these components communicate by passing parameters, each of which is either a single data item or a homogenous structure that does not includes a control element.
No Coupling	0	There is no coupling between two components if they have no communication, i.e., they are totally independent.

Different types of coupling have different types of effects. As shown in table 6.1, there are six coupling levels having range from 0-5. The level 0 shows no coupling complexity and level 5 shows highest coupling complexity. The metric, proposed in this chapter does not consider content coupling and common coupling. Because in case of content coupling, assembly languages typically supported it, but not high-level languages. This form of coupling is unacceptable because it forces you to understand the semantics, implemented in component, which may not be possible in the case of black box component because of unavailability of source code or internal details [50][51]. On the other hand in case of common coupling, the modules share the global variables. When global variables are used directly, it is relatively straightforward to find all instances of these global variables and check their effect. But with pointer variables, a global variable may have several aliases. This makes it impractical to track the possible interactions among different modules, and increases the risk of undesirable effects [48][49][50]. Thus it is very difficult to find and understand the use of the common referenced variables without source code [49]. Thus the proposed metric takes into account the coupling types

in which some kind of information is passed between components and which can be easily determined from the component specification , without going into internal details.

6.3 The Proposed Coupling Complexity Metric for Black Box Components

In order to determine the coupling complexity, it is necessary to find the various factors that affect the coupling complexity. We have proposed a metric, named Coupling Complexity metric for Black Box component (CC_{BB}), which has been correlated with the aspects like effort for replacement or modification of component, and testing effort. The CC_{BB} metric uses the approach in which different weight values are assigned to factors affecting the component coupling complexity. We have considered the following factors affecting component coupling complexity:

- Type of Coupling
- Counts of different types of parameters
- Number of components connected to the considered component

The explanation of these factors is as:

- **Type of Coupling:** The types of parameters which are passed between the components, i.e., type of coupling affects the coupling complexity. As shown in table 6.1, when control parameter is passed in case of control coupling, it has more complexity level than in case of stamp and data coupling. Similarly when a record type variable is passed between components as in case of stamp coupling, then it has more complexity level than data coupling complexity level but lower than the control coupling complexity level. So we have assigned some numeric weight values to the considered coupling types on the basis of their complexity, as shown below in table 6.2.

Table 6.2: Numeric weight values assigned to different coupling types

Coupling Type	Weight Value
Control Coupling	0.30
Stamp Coupling	0.20
Data Coupling	0.10

No Coupling	0

- **Counts of Different Types of Parameters**

  The counts of different types of parameters which are passed between the considered component and components connected to it, also affect the coupling complexity. For example, coupling complexity of a component receiving one control parameter and two record type variables will be more than the coupling complexity of a component receiving one record type variable and one data parameter. More number of complex parameters passed between components means more coupling complexity and it will increase the integration effort.

- **Number of Connected Components**

  The number of components connected to the considered component also affects its coupling complexity. More number of connected components will result in more effort for understanding the behavior of all the components in order to couple them. It will also increase the integration and testing effort. More number of connected components will reduce the ease of modification or replacement, i.e., reduce ease the software evolution. For example, component X and Y are connected to 2 and 4 components respectively, any modification in X component may require modifications in 2 components, but if any modification is made in Y component then it may require modifications in 4 components connected to it which will be more difficult.

Thus by considering all these concepts, Coupling Complexity metric for Black Box component (CC_{BB}), has been defined as:

$$CC_{BB} = C_n * (CPC + SPC + DPC) \quad \ldots(i)$$

Where C_n is the number of components connected to the considered component, CPC is the Control Parameter Complexity caused by passing or receiving the control parameters, SPC is the Stamp Parameter Complexity caused by passing or receiving the record type variables and DPC is the Data Parameter Complexity caused by passing or receiving the data parameters by the considered component. These complexities have been defined as below:

$$CPC = W_{CC} * N_C \quad \ldots(ii)$$

$$SPC = W_{SC} * N_r \quad \ldots(iii)$$

$$DPC = W_{dc} * N_d \quad \ldots(iv)$$

Where N_c , N_r and N_d represent the number of control parameters, number of record type variables and number of data parameters respectively, which are being passed or accepted by the considered component. W_{cc}, W_{sc} and W_{dc} represent the weight values assigned to control coupling, stamp coupling and data coupling respectively, as shown in table 6.2.

If the coupling complexity of the two components is same, then the component connected to less number of components will be considered better, but if the number of connected components are also same then the number of worst type of parameters (accepted or passed) will be compared for both the components in order to select the better component.

6.4 Theoretical Evaluation of Proposed Metric using Weyuker's Properties

Weyuker [37] proposed an axiomatic framework in the form of several properties for evaluating complexity aspects of software systems. These properties are:

Property1: There exist P and Q, for which |P| ≠ |Q|, where P and Q are programs/ components.

Property 2: Let C be a non-negative number, then there are finitely many programs/ components P for which |P|= C.

Property 3: There exist distinct components/programs P and Q for which |P|= |Q|.

Property 4: There are functionally equivalent programs/components P and Q such that |P| ≠|Q|.

Property 5: For any program/component bodies P and Q, we have |P|<=|P;Q| and |Q|<=|P;Q|.

Property 6: For program/component bodies P, Q and R such that |P|=|Q| and |P;R|≠ |Q;R|.

Property 7: There are program/component bodies P and Q such that Q is formed by permuting the order of statements of P and $|P| \neq |Q|$.

Property 8: If p is renaming of Q, then $|P|=|Q|$.

Property 9: There exist program/component bodies P and Q such that $|P|+|Q|< |P;Q|$.

These properties are evaluated for the proposed coupling complexity metric CC_{BB} as described below:

- There may exist two different components having different coupling complexities, thus satisfying the first property.
- The coupling complexity cannot be negative because if component is coupled to any component then it will pass or receive at least one parameter of any type which will result in some positive value for the proposed metric. Thus coupling complexity will never have negative value. It confirms the second property.
- There may exist two different components with different functionality, but the proposed complexity metric may have same value for them, as they may be connected to same number of components, passing or receiving the same number of different types of parameters. It confirms third property.
- There may be two components having same functionality, but the value of the proposed metric may be different for them, as these components may be designed by using different concepts of programming and technologies. So one component may not need to be connected with same number of components and passing or receiving same number of parameters to provide its functionality, as in another component's case. It validates fourth property.
- If a component is assembled with another component to get an integrated component for enhanced functionality, the coupling complexity of these two individual components may be less than the coupling complexity of the integrated component, as the functionality added by the another component may increase some complexity. It satisfies the fifth property.
- Two components may have same coupling complexity as they may have same number of connected components and same number of different types of parameters being passed or received. However, when integrating with another component or in the system, both may result in passing or receiving different number of parameters,

or different types of parameters in order to provide the required functionality, thus resulting in different coupling complexities in both the cases. It confirms sixth property.

- If the order of interface methods is changed, it will not change the number of components need to be connected and number of different types of parameters need to be received or passed, in order to provide the required functionality. Thus the coupling complexity will remain same. So the proposed metric does not satisfy seventh property.
- It is obvious that renaming of a component or parameters will not affect the complexity caused by that component or parameters. It confirms eighth property.
- When any two components are assembled or integrated, it may result in new more dependencies and some new methods or integration code may have to be written in order to provide required functionality. Thus it will increase the complexity of the assembled component. It validates the last property.

6.5 Empirical Evaluation of Proposed Coupling Complexity Metric

For empirically evaluating the proposed metric, we have considered a coupling model graph consisting of five components, which models the coupling between components. The nodes C_1, C_2, C_3, C_4 and C_5 correspond to the components as shown in figure 6.1, and there may be more than one arc between two nodes. Each arc from a node x to a node y represents coupling between components x and y. Each arc is labeled by a pair (I,j), where i represents the coupling relation R_i which is numeric value assigned to coupling type, and j is the number of times the given type of coupling occurs between x and y ,i.e, number of parameters passed from x to y [38]. But for simplicity, we have used the alphabets C, S and D to represent the control coupling, stamp coupling and data coupling respectively between components, instead of using numeric values for coupling relation R_i. The direction of arc represents the direction of flow of parameters from one component to another.

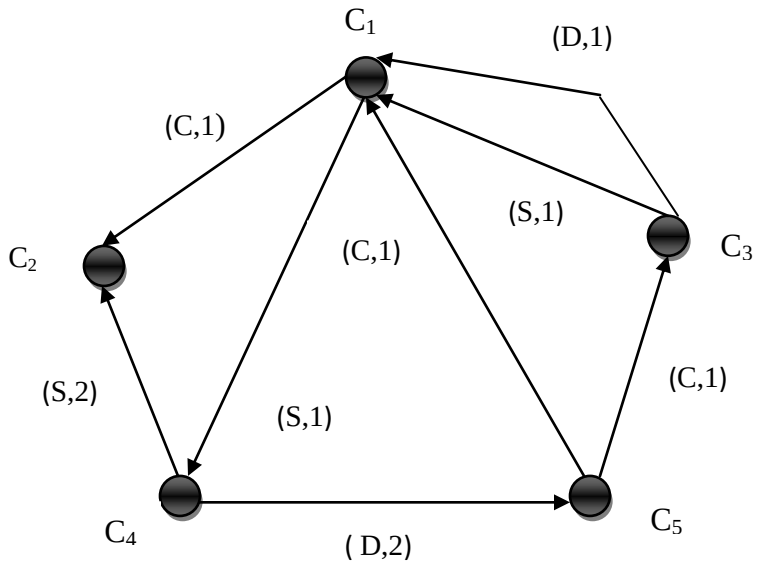

Figure 6.1: Coupling Model Graph [38]

To obtain the values of proposed coupling complexity metric, the sequence of equations (ii), (iii), (iv) and (i) is applied, on each component of the considered coupling model graph. The following table 6.3 represents the value of CC_{BB} metric for each component in considered coupling model graph. Where C_n represents the number of connected components to the considered component.

Table 6.3: CC_{BB} value of each component in the considered coupling model graph

Seq no.	$CPC=N_c * Wcc$	$SPC=N_r * Wsc$	$DPC=N_d * W_{dc}$	C_n	$CC_{BB} = C_n * (CPC+ SPC+ DPC)$
C_1	2 * 0.30	2 * 0.20	1 * 0.10	4	4.4
C_2	1 * 0.30	2 * 0.20	0	2	1.4
C_3	1 * 0.30	1 * 0.20	1* 0.10	2	1.2
C_4	0	3 * 0.20	2 * 0.10	3	2.4
C_5	2 * 0.30	0	2 * 0.10	3	2.4

In order to validate the proposed metric, it has been correlated with the aspects like effort for replacement or modification, and testing effort by using the factors: Number of

Connected Components and Number of Interactions respectively. Where the number of connected components is the number of components coupled to the considered component and it affects the ease of replacement and modification, as discussed in section 6.3. The number of interactions is the count of all types of incoming and outgoing interactions for a considered component (total number of passed and received parameters by the considered component).

- **Evaluating the effort for replacement or modifications by using the factor number of connected components**

The component should have ease of replacement and modification in order to support the changed requirements. But the component coupled to many components may have less ease of replacement or modifications. For example, as shown in figure 6.1, if component C_2 is replaced or modified, it may result in the requirement for the modifications in the two components connected to it. But if the component C_1 is modified or replaced then it may result in the requirement for the modifications in the four components connected to it, which will require more effort. Thus the component connected to many components will require more effort for replacing or modifying it, as it may cause the modifications in the components connected to it, in order to provide the required functionality. Thus it may reduce the ease of replacement or modification of component, which will reduce the ease of software evolution. The number of connected components has been represented by C_n in our case, as shown in table 6.3.

- **Evaluating the testing effort by using the factor number of interactions**

Testability may be defined as the ease with which the component can be tested. The use of the components that are independent or have less number of incoming and outgoing interactions may result in simple and less number of test cases generation as the less number of input and output combinations will have to be considered, which will help in reducing the testing effort. In other words the component having more number of incoming and outgoing interactions will require more testing effort, which shows less testability. The following table 6.4 shows the number of interactions for each component (which is equal to the sum of all types of parameters passed and received by that component). The number of interactions has been represented by I_n.

Table 6.4: The number of interactions for each component in coupling model graph

Component Name	I_n
C_1	5
C_2	3
C_3	3
C_4	5
C_5	4

Karl Pearson's Correlation Coefficient has been used for correlating the proposed coupling complexity metric CC_{BB} with the considered aspects like effort for replacement or modifications and testing effort.

6.6 Validation of the Proposed Coupling Complexity Metric

A correlation analysis by using Karl Person Correlation Coefficient, has been carried out for coupling complexity metric CC_{BB} with the aspects like effort for replacement or modification, and testing effort by using the factors Number of Connected Components (C_n) and Number of Interactions (I_n) respectively. The formula for calculating the Karl Pearson Correlation Coefficient is as below:

$$r = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{(\sum X^2 - \frac{(\sum X)^2}{N})(\sum Y^2 - \frac{(\sum Y)^2}{N})}}$$

The following tables 6.5 and 6.6 shows the calculations for Karl Pearson Correlation Coefficient between CC_{BB} and Effort for Replacement or Modification, and between CC_{BB} and Testing Effort respectively.

Table 6.5: Calculations for the Karl Pearson Correlation Coefficient between CC_{BB} and Effort for Replacement or Modification

Seq no.	X= CC_{BB}	Y= C_n	X^2	Y^2	XY
C_1	4.4	4	19.36	16	17.6
C_2	1.4	2	1.96	4	2.8
C_3	1.2	2	1.44	4	2.4

C_4	2.4	3	5.76	9	7.2
C_5	2.4	3	5.76	9	7.2
	$\sum X = 11.80$	$\sum Y = 14$	$\sum X^2 = 34.28$	$\sum Y^2 = 42$	$\sum XY = 37.2$

Table 6.6: Calculations for the Karl Pearson Correlation Coefficient between CC_{BB} and Testing Effort

Seq no.	$X = CC_{BB}$	$Y = I_n$	X^2	Y^2	XY
C_1	4.4	5	19.36	25	22
C_2	1.4	3	1.96	9	4.2
C_3	1.2	3	1.44	9	3.6
C_4	2.4	5	5.76	25	12
C_5	2.4	4	5.76	16	9.6
	$\sum X = 11.80$	$\sum Y = 20$	$\sum X^2 = 34.28$	$\sum Y^2 = 84$	$\sum XY = 51.4$

By putting all the values in the formula for Karl Pearson Correlation Coefficient, we get the value of Karl Pearson Correlation Coefficient. The following table 6.7 shows the values of Karl Pearson Correlation Coefficient between CC_{BB} and Effort for Replacement or Modification and between CC_{BB} and Testing Effort.

Table 6.7: Correlation coefficients between CC_{BB} and considered aspects

Characteristics	Correlation Coefficient
CC_{BB} Vs Effort for Replacement or Modification	+ 0.98
CC_{BB} Vs Testing Effort	+ 0.83

As shown in table 6.7, the value of correlation coefficient between CC_{BB} and effort for replacement or modification is (+0.98), which shows the positive relation between them, means as the value of CC_{BB} increases, it will increase the effort for replacement or modification of the component in the system, which shows less ease of replacement or modification of component in system and it will decrease the ease of the software evolution. The value of correlation coefficient between CC_{BB} and testing effort is (+0.83) which also shows the positive relation between them, which means increase in value of CC_{BB} shows the increase in testing effort, which shows decrease in the testability of component. Thus the component having less CC_{BB} value will have less complexity, more

ease of replacement and modification, more testability and it will be considered better than other components which have high CC_{BB} value.

6.7 The Assumptions

There should be average number of five interactions per component in CBS system, otherwise the design complexity of component would be highly complex and it will be very complex to maintain the component [4]. So it is suggested that number of parameters passed between the considered component and the components connected to it should not be greater than five, otherwise it will greatly affect the integration effort and testing effort. By using this concept, some points have been concluded.

Table 6.8: Best Case and Worst Case Coupling Complexity Factors

Factors → / Count ↓	C_n	N_c	N_r	N_d	CC_{BB}	Comment
	1	0	0	1	0.10	Best Case
	5	5	0	0	7.5	Worst Case

The Conclusions:

1. In the best case of coupling complexity, the considered component will be connected to only one component and only one data parameter will be passed from one component to another. The coupling complexity value for this case will be 0.10.
2. In the worst case of coupling complexity (by using the concept of five parameters) the considered component will be connected to five components and five parameters of control type will be passed, means one parameter between each pair of considered component with the components connected to it. The coupling complexity value of this case will be 7.5. This is the most unwanted case and the component should not be control coupled to more than two components.
3. But most of the times the developer will have to deal with the coupling complexity having range between the best case and worst case coupling complexity. Some

points have been suggested regarding the coupling complexity having range between best case and worst case coupling complexity. These are:

3.1 The number of data type parameters passed between the considered component and the components connected to it may have range between 0-5.

3.2 But if five parameters are being passed between the components, the record type variables should not be more than 3 otherwise it will create difficulty in integration and testing.

3.3 But in case of control parameters, there should not be more than 2 control parameters which are being passed between the considered component and components connected to it. Because they cause in more coupling complexity and it will require more effort to couple the components and to test their behavior.

3.4 The component passing or receiving less complex type of parameters will be considered better than others.

Conclusion and Future Scope

Measuring and controlling software complexity is an important aspect during each software development paradigm. For this purpose the researchers have proposed a wide range of complexity metrics. This thesis work assumes that complexity of CBS system can be reduced by using less complex components. So less complex components should be selected for CBS system development. But in case of CBSD, it is difficult to measure and control the complexity of software which is being developed, as most of the existing complexity metrics are based on source code or internal details of component. So these metrics are not suitable and sufficient for measuring the component complexity during the component selection process, as most of the time source code and internal details of components are not available. So there a need of complexity metrics that can measure the component complexity without using source code or internal details of components.

In this thesis, two component complexity metrics have been proposed, one for measuring component interface complexity named IC_{BB} which has been correlated with quality characteristics portability and reusability, and another for measuring coupling complexity of a component named CC_{BB} which has been correlated with software aspects named effort for replacement or modification and testing effort. Thus by using IC_{BB} complexity metric during the process of component selection for CBS system, a less complex component having more portability and reusability than other components can be selected. And by using CC_{BB} complexity metric the testability of the component and ease of replacement or modification of the component for software evolution can be predicted. Thus by selecting less complex components, a less complex CBS system having good quality can be developed.

In future these metrics can be correlated with other quality characteristics to determine their behavior with respect to those quality characteristics, which may improve the selection process for potential candidate component selection. In future the tools can be generated for automating the determination of the component complexity based on these metrics. Many other ways can be found for including the more factors for calculating the precise values of the proposed metrics.

References

1. Ivica Crnkovic and Magnus Larsson, "Component-Based Software Engineering-New Paradigm of Software Development," http://scholar.google.co.in/scholar?q=ComponentBased+Software+Engineering+%E2%80%93+New+Paradigm+of+Software+Development , 2001.
2. Prapanna Parthasarathy, "Component Integration Metrics and Their Evaluation," Masters Theses & Specialist Projects, Western Kentucky University, 2007.
3. Ivica Crnkovic, Magnus Larsson, and Frank Luders, "The Different Aspects of Component Based Software Engineering," http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.202.9388
4. Usha Kumari and Shuchita Upadhyaya, "An Interface Complexity Measure for Component-based Software Systems," International Journal of Computer Applications, vol.36, no.1, pp. 46-52, December 2011.
5. Iqbaldeep Kaur, Parvinder S. Sandhu, Hardeep Singh, and Vandana Saini, "Analytical Study of Component Based Software Engineering," World Academy of Science, Engineering and Technology, vol.3, no.26, pp. 437-442, February 2009.
6. Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th ed., New York: McGrawHill, 2001.
7. Arvinder Kaur and Kulvinder Singh Mann, "Component Based Software Engineering," International Journal of Computer Applications, vol.2, no.1, pp. 105-108, May 2010.
8. Ian Sommerville, "Software Engineering," 7th edition, 2004, Available at: http://ifs.host.cs.st-andrews.ac.uk/Books/SE7/Presentations/PDF/Ch19.pdf
9. Sandeep Khimta, Parvinder S. Sandhu, and Amanpreet Singh Brar, "A Complexity Measure for Java Bean based Software Components," World Academy of Science, Engineering and Technology, vol.2, no.18, pp. 449-452, May 2010.

10. V. Lakshmi Narasimhan and Bayu Hendradjaya, "Theoretical Considerations for Software Component Metrics," World Academy of Science, Engineering and Technology, vol.1, no.10, October 2005.
11. Wook K. Kim and Jongmoon Baik, "Dynamic Model for COTS Glue Code Development and COTS Integration," http://sunset.usc.edu/classes/cs599_99/projects/COTS.pdf
12. Joakim Froberg, "Software Components and COTS in Software System Development," http://www.idt.mdh.se/cbse-book/extended reports/15_Extended_Report.pdf
13. Bogdan Korel, "Black-Box Understanding of COTS Components," in *Proc. Seventh Int. Workshop on Program Comprehension*, pp. 92-99, 1991.
14. Debayan Bose, "Component Based Development - Application in Software Engineering," http://arxiv.org/ftp/arxiv/papers/1011/1011.2163.pdf
15. Kung-Kiu Lau, Faris M. Taweel, and Cuong M. Tran, "The W Model for Component-based Software Development," in *Proc. 37th EUROMICRO Conf on Software Engineering and Advanced Applications (SEAA),* pp. 47–50, 2011.
16. K. Kaur and H. Singh, "Candidate Process Models for Component Based Software Development," Journal of Software Engineering, vol.4, pp.16-29, 2010.
17. U. A. Khan, "The Evolution of Component Based Software Engineering from the Traditional Approach to Current Practices," International Journal of Engineering and Management Research, vol.2, no.3, pp. 45-52 , June 2012.
18. Rod D. Kuhns, "Strategies for Designing and Building Reusable GIS Application Component," http://proceedings.esri.com/library/userconf/proc98/proceed/to600 /pap557/p557.htm , April 1998.
19. Usha Chhillar and Shuchita Bhasin, "A New Weighted Composite Complexity Measure for Object-Oriented Systems," International Journal of Information and Communication Technology Research, vol.1, pp. 101-108, July 2011.
20. Ivica Crnkovic, Stig Larsson, and Michel Chaudron, "Component-based Development Process and Component Lifecycle," in *Proc. Int. Conf on Software Engineering Advances (ICSEA'06),* 2006.

21. Bill Curtis, "Measurement and Experimentation in Software Engineering," in *Proc. The IEEE,* pp. 1144-1157, 1980.
22. Sheng Yu and Shijie Zhou, "A Survey on Metric of Software Complexity," in *Proc. The 2nd IEEE Int. Conf on Information Management and Engineering (ICIME)* , pp. 352–356, April.16-18, 2010.
23. Dr. P. K. Suri and Neeraj Garg, "Software Reuse Metrics: Measuring Component Independence and its Applicability in Software Reuse," International Journal of Computer Science and Network Security, vol.9, no.5, pp. 237-248, May 2009.
24. Seyyed Mohsen Jamali, "Object Oriented Metrics," http://www.cs.sfu.ca/~sja25/personal/resources/papers/ObjectOrientedMetrics.pdf
25. Arun Sharma, Rajesh Kumar, and P. S. Grover, "Empirical Evaluation and Critical Review of Complexity Metrics for Software Components," in *Proc. The 6th WSEAS Int. Conf on Software Engineering, Parallel and Distributed Systems*, pp. 24-29, 2007.
26. Parvinder Singh Sandhu and Dr. Hardeep Singh, "A Critical Suggestive Evaluation of CK Metric," http://www.pacis-net.org/file/2005/158.pdf , 2005.
27. Shyam R. Chidamber and Chris F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering, vol.20, no.6, pp. 476-49, June 1994.
28. Sanjay Mishra, "An Object Oriented Complexity Metric Based on Cognitive Weights," in *Proc. The 6th IEEE Int. Conf on Cognitive Informatics (ICCI'07)*, pp. 134 – 139, Aug. 6-8, 2007.
29. A. Fothi, J. Nyeky-Gaizler, and Z. Porkolab, "The Structured Complexity of Object-Oriented Programs," Mathematical and Computer Modeling, vol.38, pp. 815-827, 2003.
30. Tullio Vernazza, Giampiero Granatella, Giancarlo Succi, Luigi Benedicenti, and Martin Mintchev, "Defining Metrics for Software Components," in *Proc. World Multiconference on Systemic, Cybernetics and Informatics*, pp.16-23, 2000.

31. M. F. Bertoa, J. M. Troya, and A. Vallecillo, "Measuring the Usability of Software Components," Journal of Systems and Software, vol.79, pp. 427-439, 2006.
32. Eun Sook Cho, Min Sun fim, and Soo Dong Kim, "Component Metrics to Measure Component Quality," in *Proc. Eighth Asia-Pacific conf on Software Engineering*, pp. 419-426, Dec. 4-7, 2001.
33. Nasib S. Gill and P. S. Grover, "Few Important Considerations for Deriving Interface Complexity Metric for Component-based Systems," ACM SIGSOFT Software Engineering Notes, vol.29, no.2, March 2004.
34. Octavian Paul Rotaru and Marian Dobre, "Reusability Metrics for Software Components," in *Proc. The 3rd ACS/1EEE Int. Conf on Computer Systems and Application*, 2005.
35. Marcus A. S. Boxall and Saeed Araban, "Interface Metrics for Reusability Analysis of Components," in *Proc. The Australian Software Engineering Conference (ASWEC'04)*, pp. 40-51, 2004.
36. Nael Salman, "Complexity Metrics AS Predictors of Maintainability and Integrability of Software Components," Journal of Arts and Sciences, vol.5, 2006.
37. Elaine J. Weyuker, "Evaluating Software Complexity Measures," IEEE Transactions on Software Engineering, vol.14, no.9, pp.1357-1365, September 1988.
38. Norman E. Fenton and Shari Lawrence Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, 2nd ed., London: PWS Publishing,1997.
39. Miguel Goulao and Fernando Brito e Abre, "Formalizing Metrics for COTS," in *Proc. The ICSE Workshop on Models and Processes for the Evaluation of COTS Components*, 2004.
40. Abhikriti Narwal, "Empirical Evaluation of Metrics for Component Based Software Systems," International Journal of Latest Research in Science and Technology, vol.1, no.4, pp. 373-378, 2012.
41. Jarallah S. Alghamdi, "Measuring Software Coupling," The Arabian Journal for Science and Engineering, vol.33, pp.119-129, April 2008.

42. Martin Hitz and Behzad Montazeri, "Measuring Coupling and Cohesion in Object-Oriented System," http://www.isys.uni-klu.ac.at/PDF/1995-0043-MHBM.pdf
43. Puneet Goswami, Pradeep Kumar, and Keshav Nand, "Evaluation of Complexity for Components in Component Based Software Engineering," International Journal of Research in Engineering and Applied Sciences, vol.2, no.2, February 2012.
44. "Reusability," https://en.wikipedia.org/wiki/Reusability, July 2012.
45. D. L. Parnas, "Software Aging," in *Proc. The 16th Int. Conf on Software Engineering*, vol.33, pp. 279-287, April 2008.
46. T. Ravichandran and Marcus A. Rothenberger, "Software Reuse Strategies and Component Markets," Communications of the ACM, vol.46, no.8, pp.109-104, August 2003.
47. Pascal Beys, "Different Types of Reuse," http://hcs.science.uva.nl/usr/beys/research/node3.html , July 2011.
48. Stephen R. Schach, Tokunbo O. S. Adeshiyan, Daniel Balasubramanian, Gabor Madl, Esteban P. Osses, Sameer Singh, Karlkim Suwanmongkol, Minhui Xie, and Dror G. Feitelson, "Common Coupling and Pointer Variables, with Application to a Linux Case Study," http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.6868 , June 2005.
49. Krzysztof Czarnecki, "Modularity and Coupling and Cohesion," https://ece.uwaterloo.ca/~se464/06ST/lecture/07_modularity.pdf
50. "Coupling and Cohesion," http://www.cs.ucc.ie/~herbert/CS6321/.../B12%20CouplingCoh.pdf
51. Dr. Michelle L. Lee, "Coupling and Cohesion," http://www.cs.ucc.ie/~adrian/CouplingCohesion.pdf , 1999.
52. Arun Sharma, Rajesh Kumar, and P. S. Grover, "A Critical Survey on Reusability Aspects for Component-Based Systems," World Academy of Science, Engineering and Technology, vol.1, no.9, pp. 35-39, September 2007.
53. Pasquale Ardimento, Alessandro Bianchi, and Giuseppe Visaggio, "Maintenance-oriented Selection of Software Components," in *Proc. Eighth*

European Conf on Software Maintenance and Reengineering (CSMR'04), pp. 115-124, March. 24-26, 2004.

Abbreviations

ANIC	Average Number of Interfaces per Component
ANLC	Average Number of Links between Components
ANLI	Average Number of Links per Interface
ANMC	Average Number of Methods per Component
CBD	Component Based Development
CBO	Coupling Between Objects
CBS	Component Based Software
CBSD	Component Based Software Development
CBSE	Component Based Software Engineering
CC	Component Cohesion
CC_{BB}	Coupling Complexity metric for Black Box component
CICM	Component Interface Complexity Metric
COTS	Commercial – Off - The-Shelf
CPC	Control Parameter Complexity
DCT	Depth of the Composition Tree
DIT	Depth of Inheritance
DPC	Data Parameter Complexity
EXTCBO	External CBO
IC_{BB}	Interface Complexity metric for Black Box components
IMC	Interface Method Complexity
LCOM	Lack of Cohesion Method
LOC	Lines of Code
MAXDIT	Maximum of the DIT
MUT	Mean of Unrelated Trees
NOC	Number of Children
NOCC	Number of Children for a Component
OOP	Object Oriented Programming
PC(M)	Parameter Complexity for Method
RFC	Response for Class

RFCOM	Response Set for a Component
SCCp	Self-Completeness of Component's Parameter
SI	Student Information
SPC	Stamp Parameter Complexity
TNC	Total Number of Components
TNIC	Total Number of Implemented Components
TNL	Total Number of Links
TT	Training Time
WCC	Weighted Classes per Component
WCT	Width of the Composition Tree
WMC	Weighted Methods per Class

List of Published Papers

- Navneet Kaur and Ashima Singh, "Iterative and Non–Iterative Approaches for Architecture Based Software Development," CiiT International Journal of Software Engineering and Technology, vol.5, no.1, pp. 33-38, January 2013.
- Navneet Kaur and Ashima Singh, "Generating More Reusable Components while Development: A Technique," International Journal of Innovative Technology and Exploring Engineering, vol.2, no.3, pp. 215-221, February 2013.
- Navneet Kaur and Ashima Singh, "A Technique for Component Based Software Development," International Journal of Computer Science and Engineering, vol.2, no.2, pp.123-134, May 2013.
- Navneet Kaur and Ashima Singh, "A Complexity Metric for Black Box Components," International Journal of Soft Computing and Engineering, vol.3, no.2, pp. 179-184, May 2013.
- Navneet Kaur and Ashima Singh, "Component Complexity Metrics: A Survey," International Journal of Advanced Research in Computer Science and Software Engineering, vol.3, no.6, pp.1056-1061, June 2013.
- Navneet Kaur and Ashima Singh, "A Metric for Accessing Black Box Component Reusability," International Journal of Scientific and Engineering Research, vol.4, no.7, pp. 1114-1121, July 2013.