Evaluation of Software Complexity using Weighted Assignment Technique for Component Based System

Thesis submitted in partial fulfillment of the requirement for
the award of the degree of
Master of Science
in
Mathematics and Computing

Submitted by

Ramanpreet Kaur

Registration no.: 300803014

Under the guidance of

Dr. Rajesh Kumar

School of Mathematics and Computing
Thapar University
Patiala-147001 (PUNJAB)
INDIA

JULY-2010

Certificate

I hereby certify that the work which is being presented in the thesis entitled "Evaluation of Software Complexity using Weighted Assignment Technique for Component Based System" in the partial fulfillment of the requirements for the award of Master of Science, School of Mathematics and Computer Application, Thapar University, Patiala is an authentic record of my own work carried out under the supervision of Dr. Rajesh Kumar.

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

Ramanpreet Kaur

(Ramanpreet Kaur)

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

14/07/2010

(Dr. Rajesh Kumar)
Associate Professor
Supervisor

SMCA, Thapar University

Patiala

Countersigned by:

Dr. S.S. Bhatia

(Professor & Head)

School of Mathematics and Computer Applications

Thapar University, Patiala

23/7/10

Dr. R.K. Sharma

Dean of Academic Affairs

Thapar University

Patiala

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.

I would like to express my deep and sincere gratitude to my supervisor Dr. Rajesh Kumar, Associate Professor, School of Mathematics and Computer Application (SMCA). His wide knowledge and logical way of thinking have been of great value for me. His understanding and personal guidance have provided a good basis for the present thesis.

I owe my sincere thanks to Dr. S. S. Bhatia, Professor & Head of School of Mathematics and Computer Application (SMCA), Thapar university, Patiala for providing facilities. I wish to express my sincere thanks to Mr. Sudhir Goyal, System Analyst cum Programmer, Computer Center, Thapar University, Patiala, for their co-operation and help in the project work.

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.

Ramanpreet Kaur

(Ramanpreet Kaur)

Contents

Abstract

Component Based Software Engineering (CBSE) is focused on assembling existing components to build a software system, with a potential benefit of delivering quality systems by using quality components. It departs from the conventional software development process in that it is integration centric as opposed to development centric. Various metrics have been developed by the researchers for improving the quality of the software components. Measurement of an attribute involves a metric that depends only on the value of the attribute, but few or no software engineering attributes or tasks are so simple that measures of them can be direct. Thus, all metrics should be validated. In the present thesis, complexity metric is proposed for component based on the different constituents of the component like, methods and properties with different weights assigned to them. This metric is applied on various JavaBeans components for empirical evaluation. Further, correlation study has been conducted for this metric with several quality characteristics, like, customizability and readability. The study conducted shows negative correlation between them, which confirms the assumption that high complexity of the components leads to the high cost of maintainability.

1 Component Based Software Engineering

1.1 Introduction

The concept of building software from components is not new. A classical design of complex software systems always begins with the identification of system parts designated sub-systems or blocks, and on a lower level modules, classes, procedures and so on. The reuse approach to software development has been used for many years. However, the recent emergence of new technologies has significantly increased the possibilities of building systems and applications from reusable components. Both customers and suppliers have had great expectations from Component Based Systems, but their expectations have not always been satisfied. Experience has shown that component-based development requires a systematic approach to and focus on the component aspects of software development. Traditional software engineering disciplines must be adjusted to the new approach, and new procedures must be developed. *Component-based Software Engineering* (CBSE) has become recognized as such a new sub-discipline of Software Engineering. [11]

CBSE is a process that aims to design and construct software systems using reusable software components. Clements describes CBSE as embodying *"the 'buy, don't build' philosophy". He also says that "in the same way that early subroutines liberated the programmer from thinking about details, [CBSE] shifts the emphasis from programming to composing software systems"*. [7]

The purpose of CBSD is to develop large systems, incorporating previously developed or existing component, thus cutting down the on development time and cost. It can also be used to reduce maintenance associated with the upgrading of large systems. It is assumed that common parts (be it classes or functions) in a software application only need to written once and re-used rather than being re-written every time a new application is developed.

If one is familiar with Object oriented programming (OOP), it can be useful to think of CBSE in a similar way. In OOP, code is reused in the form of objects. These objects are often contained in vast libraries of reusable code. Frameworks take the process even further, providing more robust and disciplined systems of reuse. By obtaining and reusing parts of systems which have already been 'tried and tested', we can exploit the principal advantages of object-oriented programming techniques over procedural programming techniques, which is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify. In the same way, in CBSE, by reusing an existing component you cut out a lot of the hard work with establishing the usefulness and in testing that component – although, as we will see, some testing will still be required. [7]

1.2 Component Based Software Engineering:

CBSE is aiming at realizing long-awaited software reuse by changing both software architecture and software process. Because of the extensive uses of components, the Component-Based Software Engineering (CBSE) process is quite different from that of the traditional waterfall approach. CBSE not only requires focus on system specification and development, but also requires additional consideration for overall system context, individual components properties and component acquisition and integration process. This work presents an indicative literature survey of techniques proposed for different phases of the CBD life cycle. The aim of this survey is to help provide a better understanding of different CBD techniques for each of these areas. Component-based software engineering (CBSE) is an approach to software development that relies on software reuse. It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific. Components are more abstract than object classes and can be considered to be standalone service providers. [10]

1.2.1 Component

An individual component is a software package or a module that encapsulates a set of related functions (or data). A component is something that can be deployed as a black box. It has an external specification, which is independent of its internal mechanisms [29].Researchers have proposed certain definitions:

- A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard (Councill and Heinmann, 2001). [8]
- A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties (Szyperski, 1988).[1, 29]
- Brown and Wallnau describe a component as "a nontrivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture". In some ways, this is similar to the description of an object in OOP. Components have an interface. They employ inheritance rules. But the definition is taken even further. Components are defined to offer some level of service (a Service Level Agreement, perhaps). In the case of COTS (Commercial off-the-shelf components), little or nothing is known about the internal workings of the component by the software engineer. Instead, the software engineer is given only a well-defined external interface from which he must work. The level of service offered is therefore crucial and must be accurate if the integration of the component into the software system is to be successful. Brown and Wallnau describe a software component as "a unit of composition with contractually specified and explicit context dependencies only". [7]
- A software component is a unit of packaging, distribution or delivery that provides services within a data integrity or encapsulation boundary. (Microsoft Corp.) [16]
- A component is a language neutral, independently implemented package of software services, delivered in an encapsulated and replaceable container, accessed via one or more published interface. While a component may have the ability to modify a database, it should not be expected to maintain state information. A component is not platform-constrained nor is it application-bound (Sparling, 2000). [1]

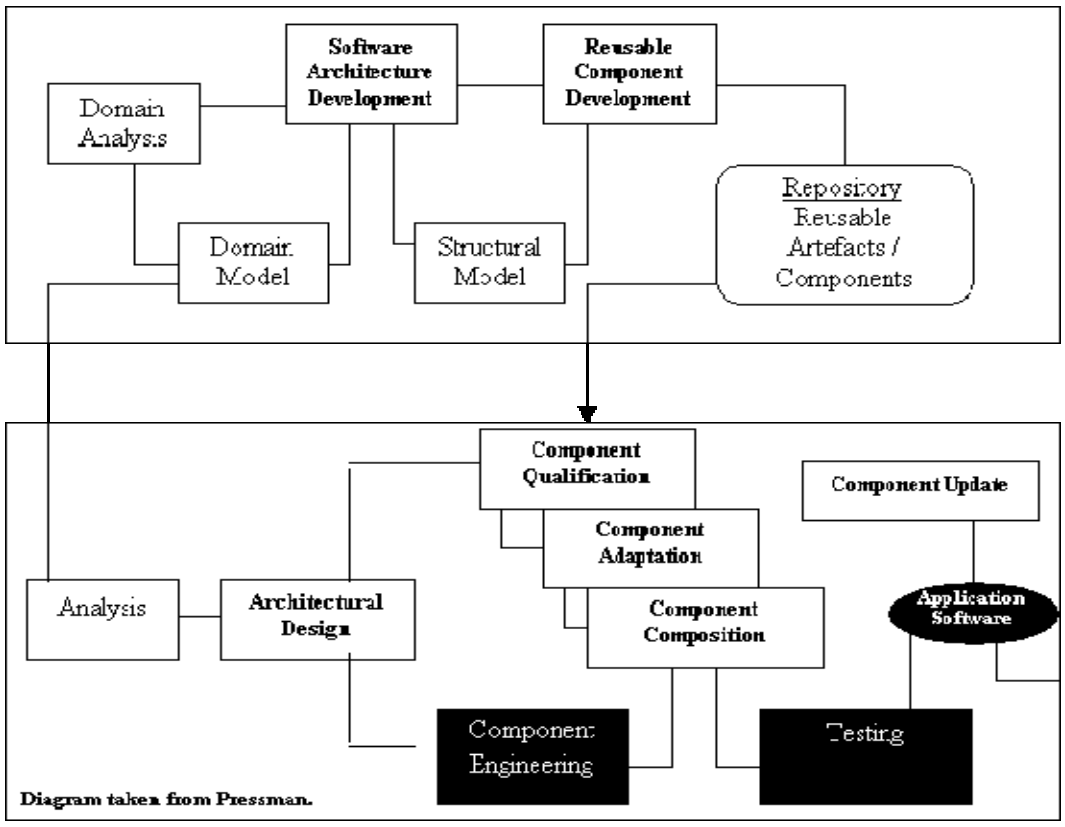
1.2.2 The CBSE-Process

CBSE is in many ways similar to conventional or object-oriented software engineering. A software team establishes requirements for the system to be built using conventional requirements elicitation techniques. An architectural design is established. Here though, the process differs. Rather than a more detailed design task, the team now examines the requirements to determine what subset is directly amenable to composition, rather than construction.

The CBSE process identifies not only candidate components but also qualifies each component's interface, adapts components to remove architectural mismatches, assembles components into selected architectural style, and updates components as requirements for the system change.

As shown in the diagram below, two processes occur in parallel during the CBSE process. These are:

- Domain Engineering
- Component Based Development



Domain Engineering:

This aims to identify, construct, catalogue, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. An application domain is like a product family – applications with similar functionality or intended functionality. The goal is to establish a mechanism by which software engineers can share these components in order to reuse them in future system.

"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".(Paul Clements) [7]

Examples of application domains are:

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

Domain engineering begins by identifying the domain to be analyzed. This is achieved by examining existing applications and by consulting experts of the type of application you are aiming to develop. A domain model is then realized by identifying operations and relationships that recur across the domain and therefore being candidates for reuse. This model guides the software engineer to identify and categorize components, which will be subsequently implemented.

One particular approach to domain engineering is Structural Modeling. This is a pattern-based approach that works under the assumption that every application domain has repeating patterns. These patters may be in function, data, or behavior that has reuse potential. This is similar to the pattern-based approach in OOP, where a particular style of coding is reapplied in different contexts. An example of Structural Modeling is in Aircraft avionics: The systems differ greatly in specifics, but all modern software in this domain has the same structural model.

Component based development:

Component based software development encompasses two processes:

- Assembling software systems from software components.
- Developing reusable component.

The activity of developing systems as assemblies of components may be broadly classified in terms of four activities [25]:

- Component Qualification.
- Component Adaptation.
- Component Assembly.
- System evolution and maintenance.

In the following subsections we give a brief description of each of the above activities:

- ***Qualification:***

Qualification is the process for determining the suitability of a component for use within the intended final system. When a marketplace of competing products exists, it also involves the selection of the most suitable component. Selection is dependent on the condition that measures exist for comparing one component against another and evaluating the fitness of use of components. It is during activity that the issues of trust and certification arise. The process of certification is two-fold (Bachman et al. 2000):

- To establish facts about a component and to determine that the properties a component possesses is also conformant with its published specification; and
- To establish trust in the validity of these fact, perhaps by having a trusted third-party organization attest the truth of this conformance and to provide a certificate to verify this.

- ***Adaptation:***

Individual components are written to meet different requirements, each one making certain assumptions about the context in which it is deployed. The purpose of adaptation is to

ensure that conflicts among the components are minimized. Different approaches to adaptation depend upon the accessibility of the internal structure of a component.

- ***Assembly:***

Assembly is the integration of components through some well defined infrastructure, which provides the binding that forms a system from disparate components. COTS components, for example, are usually written to some component model defined by, for e.g., Enterprise JavaBeans, COM, and COBRA.

- ***System evolution and Maintenance:***

Because components are the units of change, system evolution is based around the replacing of outdated components by new ones, or, at least, ideally. The treatment of components as plug-replaceable units is a simplistic view of system evolution. In practice, replacing a component may be a non-trivial task, especially when there is a mismatch between the new and old one, triggering another stage of adaptation with the new component.

1.3 CBSE v/s Object-oriented programming

Proponents of object-oriented programming (OOP) maintain that software should be written according to a mental model of the actual or imagined objects it represents. OOP and the related disciplines of object-oriented design and object-oriented analysis focus on modeling real-world interactions and attempting to create "verbs" and "nouns" which can be used in intuitive ways, ideally by end users as well as by programmers coding for those end users. In OOP, code is reused in the form of objects, several mechanisms such as inheritance and polymorphism let the developers reuse these objects in several ways.

Component-based software engineering, by contrast, makes no such assumptions, and instead states that developers should construct software by gluing together prefabricated components - much like in the fields of electronics or mechanics. Some peers will even talk of modularizing systems as software components as a new programming paradigm. [30]

1.4 Benefits of CBSE:

The goal of component-based software engineering is to increase the productivity, quality, and time-to-market in software development. One important paradigm shift implied here is to build software systems from standard components rather than "reinventing the wheel" each time. This requires thinking in terms of system families rather than single systems.

Increasingly, complex systems now need to be built with shorter amounts of time available. The “buy, don’t build approach” is more important and popular than ever. Over the past decade, many people have attempted to improve software development practices by improving design techniques, developing more expressive notations for capturing a system’s intended functionality, and encouraging reuse of pre-developed system pieces rather than building from scratch. Each approach has had some notable success in improving the quality, flexibility, and maintainability of application systems, helping many organizations develop complex, mission-critical applications deployed on a wide range of platforms. [22]

Despite this success, any organization developing, deploying, and maintaining large-scale software-intensive systems still faces tremendous problems, especially when it comes to testing and updating the systems. Unless carefully designed, it can be very expensive (in time as well as cost) to add further functionality to a system effectively, and then to test whether the addition has been successful. Furthermore, in recent years, the requirements, tactics, and expectations of application developers have changed significantly. They are more aware of the need to write reusable code – even though they may not always employ this practice.

- The business and organizational context within which applications are developed, deployed, and maintained has changed. There is an increasing need to communicate with legacy systems, as well as constantly updating current systems. This need for new functionality in current applications requires technology that will allow easy additions.
- CBSE allows use of predictable architectural patterns and standard software architecture leading to a higher quality end result.
- Lower cost of development and shorter delivery schedules.
- Better reliability and reduced maintenance costs.
- Lets developers focus on their business requirements and core competencies, rather than re-solving the same technical problems over and over.

- Provides extensibility because components can be assembled into many different configurations to provide unique variants of a system as needed. (This is especially common today for industries such as cellular technology, consumer electronics, and automotive systems).
- Components that use different languages and technologies can be mixed and matched.
- 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. [1, 22]

2 Software Metrics

2.1 Introduction:

Managing a process requires information upon which the management decisions are based. Hence, for effectively managing a process, objective data is needed. For this, software metrics is needed. Software Metrics are quantifiable measures that could be used to measure different characteristics of a software system or system development process. The basic purpose of metrics at any point during a development project is to provide quantitative information to the management process so that the information can be used to effectively control the development process. Unless the metric is useful in some form to monitor or to control the cost, schedule, quality or complexity of the project, it is of little use for the project. There are two types of software metrics used for software development: Product Metrics and Process metrics.

Product Metrics are used to quantify characteristics of product being developed i.e. software. *Process Metrics* are used to quantify characteristics of process being used to develop the software. Process Metrics aim to measure such considerations as productivity, cost and resource requirements, effectiveness of quality assurance measures, and the effect of development techniques and tools. Metrics and measurement are necessary aspects of managing software development project. Without proper metrics to quantify the required information, subjective decisions would have to be used, which are often unreliable and against the fundamental goals of engineering. [18]

We will discuss Software Complexity in detail in the following section.

2.2 Software Complexity Metrics

The first problem encountered when attempting to understand program complexity is to define what it means for a program to be complex. Software complexity has been studied for over many years now, during which time lots of measures have been proposed to capture many different aspects of software complexity however, there is no consensus about what software complexity actually is. What is accepted is that there are two main categories of software complexity: computational and psychological.

- Computational complexity refers to algorithm efficiency in terms of the time and memory needed to execute a program.
- Psychological (or cognitive) complexity refers to the human effort needed to perform a software task, or, in other words, the difficulty experienced in understanding or performing such a task. [5]

Basili (1980) defines complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing, or modifying software. [1, 5]

Henderson-Sellers (1996) challenged this definition, calling it is too broad, and proposed another definition, as follows: "The cognitive complexity of software refers to those characteristics of software that affect the level of resources used by a person performing a given task on it."[5]

2.2.1 Review of Complexity Metrics for Object Oriented Systems:

The most impressive findings related to Object-Oriented metrics were the one proposed by Chidamber and Kemerer. They have proposed six new metrics. [1, 4]
These Complexity Metrics are:

a) Weighted Methods per Class (WMC):

The Weighted Methods per Class Metric is defined as the sum of the complexity of a class' local methods and intended to count the combined complexity of local methods in a given class.

Consider a Class C_1 with methods $M_1 \ldots M_n$ that is defined in the class. Let $c_1 \ldots c_n$ the complexity of the methods. Then [28]:

$$WMC = \sum c_k$$

b) Depth of Inheritance (DIT):

Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritances, the DIT will be the maximum length from the node to the root of the tree.

c) Response for Class (RFC):

The Response for Class is the number of methods that can execute in response to a message sent to an object within this class, using to one level of nesting.

d) Number of Children (NOC):

The Number of Children metrics is defined as the number of sub-classes subordinated to a class in the class hierarchy.

e) Lack of Cohesion of Methods (LCOM):

This metric is count of number of disjoint method pairs minus the number of similar method pairs. The disjoint methods have no common instance variables, while similar methods have at least one common instance variable.

f) Coupling between Objects (CBO):

CBO for a class is a count of number of non-inheritance related couples with classes.

The problems associated with some of the Chibamber and Kemerer Metrics were discovered during the course of defining the unit definition model for the metrics.

Li proposed other metrics for complexity [1, 4], based on size measures, such as, number of methods and attributes. The following section presents six metrics proposed in order to overcome some of the limitations found in Chidamber and Kemerer Metrics:

a) Number of Ancestor classes (NAC):

The Number of Ancestor classes (NAC) metric was proposed, as an alternative to the DIT metric, to measure this attribute of a class. Li define the NAC as the total number of ancestor classes from which a class inherits in the class inheritance hierarchy.

b) Number of Descendent classes (NDC):

The Number of Descendent Classes (NDC) metric is proposed as an alternative to the NOC metric. It defined as the total number of descendent classes (subclass) of a class. The theoretical basis and viewpoints remain the same as NOC. Li reported that the attribute of a class that the NOC metric captures is the number of classes that may potentially be influenced by the class because of inheritance relations. Li claimed that the NDC metric captures the classes attribute better than NOC.

c) Number of Local Methods (NLM):

This is one of the metric proposed by Li in order to measure the attributes of a class that WMC metric intends to capture. The Number of Local Methods metric (NLM) is defined as the number of the local methods defined in a class which are accessible outside the class.

d) Class Method Complexity (CMC):

The Class Method Complexity (CMC) metric is defined as the summation of the internal structural complexity of all local method, regardless whether they are visible outside the class or not.

e) Coupling through Abstract data type (CTA):

The Coupling through Abstract Data Type (CTA) is defined as the total number of classes that are used as abstract data types in the data-attribute declaration of a class. Two classes are coupled when one class uses the other class as an abstract data type.

f) Coupling through Message passing (CTM):

The Coupling through Message Passing (CTM) defined as the number of different messages sent out from a class to other classes excluding the messages sent to the objects created as local

objects in the local methods of the class. Two classes can be coupled because one class sends a message to an object of another class, without involving the two classes through inheritance or abstract data type.

2.2.2 Complexity for Legacy System:

When faced with the task of reducing application complexity, developers may find that just determining where to focus their efforts is a challenge. A broad array of metrics could be used to measure complexity. Among these metrics developers often find that McCabe's cyclomatic complexity and Halstead's complexity metric are the most useful indicators of application complexity. [1, 17]

McCabe's Cyclomatic complexity (1976) is graph-theoretic-based concept. For a graph G with n nodes, e edges and p connected components, the cyclomatic number V (G) is defined as:

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

Halstead's complexity measurement (1977) serves a similar purpose as it measures complexity by summarizing how many operators and operands the program contains. It defines the following measurable quantities:

- n_1 is the number of distinct operators,
- n_2 is the number of distinct operands,
- $f_{1,j}$ is the number of occurrences of the j^{th} most frequent operator,
- $f_{2,j}$ is the number of occurrences of the j^{th} most frequent operator.

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

$$n= n_1+n_2$$

With the measurable parameters defined as,

$$N_1=\sum f_{1,j},\ N_2=\sum f_{2,j}$$

N_1 is the total occurrences of different operators in a program and N_2 is the total occurrences of different operands. The length of program is defined as

$$N= N_1+N_2$$

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

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

The difficulty of program is defined as

$$D= (n_1*N_2) / (2 * n_2)$$

And effort is defined as $E= D*V$

Henry and Kafura (1981) identified a form of the *fan in - fan out complexity* which maintains a count of the number of data flows from a component (fan-out) plus the number of global data structures that the program updates (fan-in). The data flow count includes updated procedure parameters and procedures called from within a module.

$$\text{Complexity} = \text{Length} * (\text{Fan-in} * \text{Fan-out})^2$$

Length is any measure of length such as lines of code or alternatively McCabe's cyclomatic complexity is sometimes substituted. [1, 17]

2.2.3 Complexity Metrics for Component-based systems

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

Tullio Vernazza *et al*. (2000) extended the CK metrics (Chidamber and Kemerer, 1994). Authors proposed new metrics corresponding to each CK metric. Like, for number of methods (NOM), they proposed weighted class per component, for number of children (NOC), the proposed metric was number of children for a component and so on. The proposed metrics are also validated against the theoretical properties proposed by Briand *et al*. (1999). However, there is no empirical validation conducted for these metrics against any industry project, thus leaving the work incomplete. [1]

Cho *et al.* (2001) proposed a suite of complexity metrics for software components. The approach considers the classes and their methods of each component and captures the dynamic complexity, based on analysis of source code of the component. However, it cannot be used to measure the complexity for black-box components, as the source code of these components is not available. [1, 20]

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

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

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

Nael (2006) proposed several metrics for components, connectors between components, interface of each component and composition tree. However, the proposed metrics are very basic in nature and are based on just the total numbers and does not consider the complexity of individual interface. [1]

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

Sharma *et al.* (2008a) proposed interface complexity metric for software components by considering interface methods and their associated properties, argument types and return types. Authors evaluated this metric on several Java Beans components and finally validated it against execution time, readability and customizability. Results concluded that complex components take much time to execute and these are very difficult to maintain. [1]

3 Complexity Metrics

3.1Introduction

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

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

$$ICM(C) = a \sum CIM_i + b \sum CP_j \quad \ldots (3.1)$$

Where CIM_i is the complexity of i^{th} interface method and CP_j is the complexity of j^{th} property. a and b are weight values for methods and properties respectively, as complexity of a interface method may have different weight value than the complexity of a property [1].

Interface methods are divided into the following categories:

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

Complexity of interface methods may be measured based on its return type and arguments passed to it. This complexity metric can be correlated with other quality criteria such as customizability, performance, understandability etc.

The theoretical and empirical evaluation done is explained in the following sections.

3.2 Theoretical Evaluation of Proposed Complexity Metric:

Weyuker's properties have been suggested as a guiding tool in identification of a good and comprehensive complexity measure by several researchers. Weyuker proposed nine properties to evaluate complexity measure for traditional programming. The proposed interface complexity is evaluated against these properties. These properties are [1, 23]:

Property 1: There exists P & Q, ($|P| \neq |Q|$), where P and Q are the two different programs.
This property states that a measure should not rank all programs as equally complex.

Property 2: Let c be a non-negative number, and then there are only finite number of programs and programs of complexity c.
There are only a finite number of programs of the same complexity.

Property 3: There are distinct programs P and Q such that $|P| = |Q|$
This property states that there are multiple programs of the same complexity.

Property 4: There exists P and Q, such that ($P=Q$ & $|P| \neq |Q|$)
This property states that implementation is important.

Property 5: For all, (|P| ≤ |P; Q |&|Q| ≤ |P; Q|).

This property states that if the combined program is constructed from class P and class Q, the value of the program complexity for the combined program is larger than the value of the programs complexity for the class P or the class Q.

Property 6a: There exists P, Q and R, (|P| = |Q|) & |P; R| ≠ |Q; R|).

6b: There exists P, Q and R, (|P| = |Q|) & |R; P| ≠ |R; Q|).

This property states that if a new program is appended to two programs which have the same program complexity, the programs complexities of two new combined programs are different or the interaction between P and R can be different than interaction between Q and R resulting in different complexity values for P+ R and Q + R.

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

This property states that permutation of elements within the item being measured can change the metric values. The intent is to ensure that metric values change due to permutation of programs.

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

This property requires that when the name of the program changes it will not affect the complexity of the programs. Even if the member function or member data name in the programs change, the programs complexity should remain unchanged.

Property 9: There exists P and Q, (|P|+ |Q|) < (|P; Q|).

This property states that the programs complexity of a new programs combined from two programs is greater than the sum of two individual programs complexities. In other words, when two programs are combined, the interaction between programs can increase the complexity metric value.

These properties are evaluated for the proposed complexity metric:

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

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

3.3 Empirical Evaluation of Proposed Complexity Metric using weighted Assignment technique:

To obtain the values of the proposed complexity metric, an experiment is conducted on various JavaBeans components (these are available at various websites www.elegantjbeans.com, www.java.sun.com, www.java2s.com). These JavaBeans components vary from very simple and small to complex and large. These have different number of attributes and methods.

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

Data types are categorized as:

- Very simple include integer, double, Boolean, float type.
- Simple include structured data type.
- Medium include Class type and Object type.
- Complex includes pointers, built-in data type.
- And highly complex includes User-defined data types.

Method having no argument (e.g. constructor) may be considered as simplest method and we assign the weight value to these methods 0.025. All other interface methods are assigned weight values depending on the type and total number of arguments and return types.

The following table shows these weight values:

Data type → / No. of data types ↓	Very Simple	Simple	Medium	Complex	Highly Complex
1-3	0.05	0.10	0.15	0.20	0.25
4-6	0.10	0.20	0.30	0.40	0.50
7-9	0.15	0.30	0.45	0.60	0.75
>=10	0.20	0.40	0.60	0.80	1.00

Table 3.1: Weight values for Interface Methods

The same table can be used for getting the weight values for properties used in the component. Now, by referring to these tables, we can measure the complexity of each interface method and property, and finally by assigning 'a' and 'b' an appropriate value, the interface complexity of the component can be calculated by using equation (3.1).

For finding the weight values of 'a' and 'b', quality characteristics such as, Customizability and Readability are evaluated:

Evaluation of Customizability

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

$$Customizability = \frac{No.\ of\ Set\ Methods}{Total\ number\ of\ Properties}$$

The following table shows the set methods, number of properties and Customizability of various JavaBeans component:

JavaBeans	Set methods	Properties	Customizability
Chart	11	20	0.55
Timer	2	3	0.67
Spinner	1	2	0.50
Unitconverter	15	29	0.52
Scrolldemo	5	14	0.36
Calculator	1	12	0.08
Calender	35	79	0.44
SimpleBean	3	3	1.00
Clock	3	3	1.00
StatusBar	6	6	1.00

Table 3.2: Values of customizability of various JavaBeans components

Evaluation of readability:

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

$$Readability = \frac{No.\ of\ Get\ Methods}{Total\ number\ of\ Properties}$$

The following table shows the get methods, number of properties and Readability of various JavaBeans component:

JavaBeans	Get methods	Properties	Readability
Chart	10	20	0.50
Timer	2	3	0.67
Spinner	2	2	1.00
Unitconverter	14	29	0.48
Scrolldemo	7	14	0.50
Calculator	4	12	0.33
Calender	29	79	0.36
SimpleBean	3	3	1.00
Clock	5	3	1.67
StatusBar	1	6	0.17

Table 3.2: Values of Readability of various JavaBeans components

In this work, an experiment is conducted to assign appropriate values to ‘a’ and ‘b’ in equation (3.1). Karl Pearson’s Correlation Coefficient was used to get the correlation coefficient between interface complexity and quality characteristics, Customizability and Readability.

The following table shows the various values:

Values of a and b	Cor. Coeff. (Cust.)	Cor. Coeff. (Read.)
a=1.2, b=0.8	-0.431	-0.346
a=1.1, b=0.9	-0.434	-0.348
a=1.0, b=1.0	-0.436	-0.350
a=0.9, b=1.1	-0.439	-0.353
a=0.8, b=1.2	-0.442	-0.356
a=0.7, b=1.3	-0.443	-0.356
a=0.6, b=1.4	-0.440	-0.355
a=0.5, b=1.5	-0.437	-0.352
a=0.4, b=1.6	-0.435	-0.347
a=0.3, b=1.7	-0.432	-0.346

Table 3.2: Various values of Correlation Coefficient for different values of a and b

From the above table, we can infer that for a= 0.8 and b=1.2, the value of correlation coefficient is best. The number of methods, properties and their respective interface complexities using these weights are calculated as:

JavaBeans	Methods	Properties	Interface Complexity
Chart	34	20	1.140
Timer	11	3	1.740
Spinner	5	2	5.680
Unitconverter	44	29	1.000
Scrolldemo	23	14	4.360
Calculator	22	12	2.828
Calender	100	69	2.440
SimpleBean	6	3	11.940
Clock	13	3	0.800
StatusBar	12	6	1.780

Table 3.3: Complexity Metrics values

3.4 Validation of the Proposed Metric:

Correlation analysis was performed for the proposed metric with several quality characteristics, like, customizability and readability on same JavaBeans components used for proposed complexity metric.

Characteristics	Correlation Coefficient
I.C. v/s Cust.	-0.442
I.C. v/s Read.	-0.356

Table 3.4: Correlation Coefficient

As there are negative correlation between Complexity and Customizability, and Complexity and Readability, which confirms that highly complex components are hard to customize and to understand which leads towards poor reusability and maintainability. These correlation coefficients and their interpretation validate the proposed complexity metric for components.

4 Conclusion

Component-based software development promises to reduce development costs by enabling rapid development of highly flexible and easily maintainable software systems. In this thesis, we have described the different stages of the component-based development process. Several previous papers regarding Software complexity Metrics are discussed. The present thesis assumes that the complexity of the whole system can be considerably reduced if the components used are not so complex. Thesis proposes a Complexity Metric, which is evaluated on Weyukar's properties. Higher complexity leads to high cost of maintainability. It is difficult to customize an application which is highly complex. The thesis conducts an empirical evaluation on various JavaBeans components and ensures the same.

References

1. Arun Sharma, "Design and Analysis of Metrics for Component-Based Software Systems", PhD Thesis, 2009

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

3. Cay S. Horstmann and Gary Cornell, "Core Java: Advanced features", Publisher: Prentice Hall, Volume 2, 8th Edition, 2008

4. Chidamber, S. and Kemerer, C., "A Metrics Suite for Object -oriented Design, Published in the IEEE Transactions on Software Engineering", Volume 20, pp: 476-493, 1994

5. De Tran-Cao, Ghislain Lévesque, and Jean-Guy Meunier, "Software Functional Complexity Measurement with the Task Complexity Approach" pp: 1, 2005

6. http://www.elegantjbeans.com/

7. Faisal Siddiqui "Component Based Software Engineering: A look at reusable software components", available at: http://www.smb.uklinux.net/reusability/

8. George T. Heineman and William T. Councill "Component Based Software Engineering" Publisher: Addison Wesley Longman, 2001.

9. Ian Sommerville "Software engineering ",7th edition, 2004, available at: http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/Presentations/PDF/Ch19.pdf

10. Iqbaldeep Kaur, Parvinder S. Sandhu, Hardeep Singh, and Vandana Saini "Analytical Study of Component Based Software Engineering", World Academy of Science, Engineering and Technology, Volume 50, pp: 437-442, 2009

11. Ivica Crnkovic, "Component-based Software Engineering – New Challenges in Software Development"

12. http://www.java2s.com/Code/Java/Swing-JFC

13. http://java.sun.com/docs/books/tutorial/uiswing/examples/learn/index.html

14. Lord Kelvin, "Applying Object-Oriented Metrics to Ada",1995

15. Lou Marcos, "Measuring software complexity", Enterprise system journal, 1997

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

17. Mit Karaka and Sencer Sultao lu "Complexity Metrics and Models", 1998, available at: http://yunus.hacettepe.edu.tr/~sencer/complexity.html

18. Pankaj jalote, "An Integrated approach to Software Engineering", Publisher: Narosa Publishing House, 3rd edition, 2005.

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

20. Puneet Goswami, P. K. Bhatia and Vijender Hooda, "Effort estimation in Component Based Software Engineering", International Journal of Information Technology and Knowledge Management, Volume 2, issue 2, pp. 437-440, 2009

21. Sajjad Mahmooda, Richard Laia,Yong Soo Kimb, Ji Hong Kimb, Seok Cheon Parkb and Hae Suk Oh, "A survey of component based system quality assurance and assessment"

22. Sandeep Khimta, Parvinder S. Sandhu, and Amanpreet Singh Brar, "A Complexity Measure for JavaBean based Software Components", World Academy of Science, Engineering and Technology, Volume 42, 2008

23. Sanjay Misra and Ibrahim Akman, "Applicability of Weyuker's Properties on OO Metrics: Some Misunderstandings", 2008, available at: http://www.doiserbia.nb.rs/img/doi/1820-0214/2008/1820-02140801017M.pdf

24. http://www.sei.cmu.edu/reports/00tr008.pdf

25. Terence Zhao, "Component Based Software Development- An overview", 2007, available at: http://hi.baidu.com/lovelink/blog/item/bed292823f328593f603a699.html

26. Tieng Wei Koh, Mohd Hasan Selamat, Abdul Azim Abdul Ghani, and Rusli Abdullah, "Review of Complexity Metrics for Object Oriented Software Products", IJCSNS International Journal of Computer Science and Network Security, Volume 314, Issue 11, pp: 314-320, 2008.

27. http://www.users.globalnet.co.uk/~rxv/CBDmain/cbdfaq.htm#Component-Based%20Development

28. Vandana Bhattacherjee, Prabhat Kumar Mahanti, and Sanjay Kumar, "Complexity Metric for Analogy based Effort Estimation", Journal of Theoretical and Applied Information Technology, Volume 6, issue 1, pp: 01-08, 2005-2008.

29. V. Lakshmi Narasimhan, and Bayu Hendradjaya, "Theoretical Considerations for Software Component Metrics", World Academy of Science, Engineering and Technology, Volume 10, 2005.

30. http://wapedia.mobi/en/Software_componentry

Abbreviations

CBD	Component-Based Development
CBO	Coupling between the objects
CBS	Component-Based System
CBSD	Component-Based System Development
CBSE	Component-Based System Engineering
CIM	Complexity of Interface Method
CK	Chidamber and Kemerer
CMC	Class Method Complexity
COBRA	Component Object Request Broker Architecture
COTS	Components off-the-shelf
CP	Complexity of property
CTA	Coupling through Abstract data type
CTM	Coupling through Message passing
DIT	Depth of Inheritance Tree
ICM	Interface Complexity Metric
ISO	International Organization for Standardization
LCOM	Lack of Cohesion in Methods
NAC	Number of Ancestor Children
NDC	Number of Descendent Children
NLM	Number of Local Methods
NOC	Number of Children
OOP	Object-Oriented programming
RFC	Response for Class
WMC	Weighted Methods per Class