Master Thesis
Software Engineering
Thesis no: MSE-2004:29
Month: August Year: 2004

Measuring Cohesion and Coupling of Object-Oriented Systems

- Derivation and Mutual Study of Cohesion and Coupling

Imran Baig

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Contact Information:
Author(s):
Imran Baig
E-mail: imba03@student.bth.se

University advisor(s):
Michael Mattsson
Department of Software Engineering and Computer Science

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.tek.bth.se
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

Cohesion and coupling are considered amongst the most important properties to evaluate the quality of a design. In the context of OO software development, cohesion means relatedness of the public functionality of a class whereas coupling stands for the degree of dependence of a class on other classes in OO system. In this thesis, a new metric has been proposed that measures the class cohesion on the basis of relative relatedness of the public methods to the overall public functionality of a class. The proposed metric for class cohesion uses a new concept of subset tree to determine relative relatedness of the public methods to the overall public functionality of a class. A set of metrics has been proposed for measuring class coupling based on three types of UML relationships, namely association, inheritance and dependency.

The reasonable metrics to measure cohesion and coupling are supposed to share the same set of input data. Sharing of input data by the metrics encourages the idea for the existence of mutual relationships between them. Based on potential relationships research questions have been formed. An attempt is made to find answers of these questions with the help of an experiment on OO system FileZilla. Mutual relationships between class cohesion and class coupling have been analyzed statistically while considering OO metrics for size and reuse. Relationships among the pairs of metrics have been discussed and results are drawn in accordance with observed correlation coefficients.

A study on Software evolution with the help of class cohesion and class coupling metrics has also been performed and observed trends have been analyzed.

Keywords: class cohesion, class coupling, relationships

CONTENTS

1 INTRODUCTION

Cohesion and coupling are considered amongst the most important metrics for measuring the structural soundness of OO system. In the OO paradigm of software development cohesion means extent to which the public methods of class perform the same task [Bieman & Kang 1995], whereas coupling means the degree of dependence of a class on other classes in the system. If we look into the existing OO metrics for cohesion, coupling, size and reuse, we will notice that all of these metrics use the shared input data. Usage of the shared input data encourages our basic supposition about the existence of relationships among the mentioned OO metrics. Based on our basic supposition, we have attempted to find out relationships among OO metrics for cohesion, coupling, size and reuse. Apart from this study on subject OO metrics, we have also the attempted to recognize the effects of software evolution on OO implementation using OO metrics for cohesion and coupling.

To capture the relationships among the mentioned OO metrics and to identify the effects of software evolution, we have performed an experiment that is comprised of number of studies on three versions of a MFC based system.

To perform the experiment, we have derived OO metrics for cohesion and coupling, however class reuse and class size metrics have been selected from existing OO metrics. From the literature review, we have seen that a lot of work has been done on class cohesion and class coupling, yet no generally accepted definitions or metrics exist for both of the metrics [Fenton & Pfleeger 1998].

For measuring class cohesion we have proposed a new metric that measures the relatedness among the public methods on the basis of their relative contribution to the overall public functionality of a class. To determine the relative contribution of public methods, a new concept of subset tree has been used. Our proposed metric, measures the class cohesion in interval of 0 to 1 inclusively. Almost all generally supported notions of OO paradigm are incorporated in the proposed metric.

As stated earlier class coupling means the degree of dependence of class among other class in OO system. We propose that one of the possible ways to measure the class coupling is by using the UML relationships among the classes. Such UML relationships are association, inheritance and dependency. This approach to measure class coupling has enabled us to distinguish among the different coupling types based on UML relationships and has also given us opportunity to analyze the effects of each UML relationships separately. Each type of UML relationship based coupling (i.e. association coupling, inheritance coupling and dependency coupling) has been studied against the class cohesion.

To perform the metrics calculation, based on the set of metrics, three versions of OO System FileZilla (MFC based Project) have been selected. All OO data from the three versions of FileZilla were inserted into MS Access DB using manual and automated methods. Based on the data in the database metrics calculation were performed by implementing specialized application designed for calculating selected set of OO metrics. To derive the results the statistical methods of correlation coefficient has been used to determine the type of relationship among the class cohesion and UML based coupling.

1.1 Related work

The related work, that we have mainly covered can be categorized into three areas namely, cohesion, coupling and software evolution. Following describes the related work in form of literature review.

1.1.1 Measuring Cohesion

According to Fenton, “The cohesion of the module is the extend to which its individual components are needed to perform the same task” [Fenton & Pfleeger 1998].

Yourdon and Constantine attempted to measure the cohesion by the classification of cohesion on ordinal scale [Yourdon and Constantine 1979]. Based on the idea that the cohesion is an intra-modular attribute [Fenton & Pfleeger 1998] simple measurement for cohesion can be defined but these measurements can not give in depth understanding of the module cohesiveness. In the previous decade, Bieman and Ott were the first known people to describe the functional cohesion based on the approach of data slices. According to Bieman and Ott data slices are the code statements that can make changes to data tokens (such as variables). Using their approach functional cohesion is measured by judging the occurrence of data tokens in the data slices of a method [Bieman & Ott 1994]. As suggested by the name, functional cohesion only applies to individual functions; its application to class cohesion is not apparent.

In OO paradigm of software development, the class cohesion can be though as “*the measurement of “relatedness” among the members of class”* [Bieman & Kang 1995].

In the context of OO implementation the word “relatedness” for a class means similarity in the methods exposed by a class. And, the members of class stand for the elements that a class composed of. Methods and attributes are the examples of member of a class. This implies if the member methods of a class are providing similar functionality then a class is said to be cohesive and if they are performing non-similar functionality then the class is said to be less cohesive or non-cohesive.

Existing metrics on class cohesion can be categorized into two types, namely implementation metrics and design metrics. As suggested by the names, implementation metrics are calculated from the source code, whereas the design metrics are calculated from the design of a system. Most of the existing metrics are implementation metrics. In this thesis, we will mainly focus on the implementation metrics for class cohesion. Following describes the some of the known attempts for measuring the class cohesion.

In the last decade, on of the most cited suite of object-oriented metrics is defined by Chidamber and Kemerer that is also known as CK metrics. In this suite LCOM (Lack of Cohesion of Methods) metric was used for class cohesion. LCOM was originally defined by [Chidamber & Kemerer 1991] as follows:

“Consider a Class *C1* with methods *M1, M2 ...Mn*. Let *Ii* set of instance variables used by method *Mi*: There are n such sets, *{I1}–{In}*.

LCOM=the number of disjoint sets formed by the intersection of the n sets.”

LCOM metric has been interpreted differently by different authors (e.g. [Hitz & Montazeri 1996], [Briand et al. 2000], [Henderson 1996] etc.,), as a result, different LCOM metrics are available from different authors. Interesting, in another incarnation of CK metrics [Chidamber & Kemerer 1994] LCOM is reinterpreted by its own originators. Following describes the reinterpreted definition of LCOM metric.

“Consider a Class C_1 with methods $M_1, M_2, \ldots, M_n$. Let $\{I_i\}$ set of instance variables used by method M_i. There are n such sets, $\{I_1\}$–$\{I_n\}$ Let $P = \{(Ii, Ij) \mid Ii \cap Ij = Ø\}$ and $Q = \{(Ii, Ij) \mid Ii \cap Ij \# Ø\}$. If all *n* sets *{Ii}... {In}* are Ø then let $P = Ø$. LCOM |P| - |Q|; if |P| > |Q| 0 otherwise”.

The problem with different CK and others LCOM metrics is that such metrics only help in identifying the non-cohesive classes, because they only measure the absence of cohesion rather than the presence of cohesion [Etzkorn et al. 2004]. Therefore, LCOM metrics do not help in distinguishing among the partially cohesive classes [Bieman and Kang 1995].

Bieman and Kang have presented a set of two metrics, namely LCC (Loose Class Cohesion) and TCC (Tight Class Cohesion) for measuring class cohesion. In their work, member attributes of the class are used for counting the pairs of connected methods in a class, based on the usage of common attribute either directly or indirectly. Direct usage of an attribute is identified when a method reads or writes to an attribute directly and indirect usage of an attribute is identified when a method calls another method that directly reads or writes to an attribute. Relative number of connected pairs of methods to the maximum possible number of pairs has been calculated to reflect the cohesion of a class [Bieman and Kang 1995]. TCC is a measure of the relative number of directly connected methods, whereas LCC is a measure of the relative number of indirectly or directly connected methods.

Recently, an evaluation study has been performed at University of Alabama Huntsville to compare the known OO metrics for class cohesion. Evaluation was conducted using the correlation between the human-oriented view of class cohesion and the statistical data produced by the metrics under the evaluation. Expert opinion has been used to get the human-oriented view of class cohesion. LCC and TCC metrics are found to be the best amongst all metrics [Etzkorn et al. 2004] as both of the metrics have shown higher correlation with human-oriented view of class cohesion.

In our opinion, TCC and LCC metrics are very useful for measuring the cohesion of classes with few public methods, conversely, the use these metrics on the classes with high number of public methods is not as suitable. Because, both of these metrics use maximum possible number of pairs of connected methods as denominator, that is given by NP= N (N-1)/2 (where N represents the total number of public methods). NP follows exponential behavior with increasing number of methods. Therefore, we think TCC and LCC metrics are not very suitable when the cohesion of classes with higher number of public methods is measured. Secondly, TCC and LCC metrics do not reflect the effect of class size when no pair of connected methods is found. In such cases, both TCC and LCC metric result zero class cohesion regardless of the number of the non-connected public methods in a class.

Bansiya also defines the cohesion as an assessment of the relatedness among the attributes and methods of the class. He uses a design metric CAM (Cohesion among Methods in Class) for measuring class cohesion. CAM measures similarity of parameter list to assess the relatedness among the methods of class [Bansiya 2002].

A reasonable metric for class cohesion is supposed to take care of all notions of OO paradigm, especially, the notions which are generally supported by all OO languages, because all of these notions may impact the cohesion of a class in some way.

The metrics, we have reviewed for class cohesion, either measure the absence of cohesion or measure the relatedness among the public methods of class on the basis of some common attributes. We propose a metric that can measure the relative relatedness of the public methods to the overall public functionality of the class.

In our opinion, to understand existing metrics from the literature is a challenging task, for instance, LCOM is a subject of criticism because of the different interpretation of its original definition [Henderson at el. 1996]. Such problems become more intense when the intension is to perform an experiment by applying existing metric for class cohesion. In such situations, some of the questions remained unanswered even after a complete literature review. On the other hand, literature on the existing cohesion metrics is very useful source to derive a new metric based on the similar type of ideas used for the derivation of existing metrics. Perhaps that is one of the reasons of not having any standard definition or metric for class cohesion. Non-existence of generally accepted standard definition or metric for class cohesion is often pointed out by papers e.g. [Fenton & Pfleeger 1998] and [Etzkorn et al. 2004]. In our research on the potential relationships between class cohesion and coupling we are supposed to use a class cohesion metric that can answer all of our questions for performing a compressive experiment on OO implementation.

1.1.2 Measuring coupling

Study of the Troy and Zweben on coupling suggests that the coupling is one of the most significant attributes affecting the overall quality of the design [Troy and Zweben 1981]. No generally accepted metric exists for coupling; however, generally, it is accepted that too much coupling in a design leads to increased system complexity [Harrison at el. 1998]; therefore, high coupling is considered as undesired property. Following describes some of the known efforts that are made for measuring coupling.

Yourdon and Constantine define the coupling as a degree of interdependence between modules [Yourdon and Constantine 1979]. Bansiya also defines coupling as a dependency of an object on other objects in a design. He uses DCC (Direct Class Coupling) metric that counts the number of classes that a class is directly related to. This metric includes the classes directly related by attribute declaration and message passing (parameter list) in methods [Bansiya 2002]. Chidamber and Kemerer have also discussed the coupling in the context of OO paradigm, in their opinion; two classes are coupled; if the method of one class uses any method or instance of other class [Chidamber and Kemerer 1994]. CBO (Coupling between object classes) metrics counts the number of coupled classes. In CBO metric, a class is coupled to other class if it uses the method or attribute defined in other class. However, CBO metric does not distinguish among different types of interactions between two classes [Briand et al. 1997]. Whereas, Briand and his colleagues have presented a detailed suite for measuring the C++ class coupling by distinguishing different types of interactions among the classes in C++ [Briand et al. 1997].

Fenton and Pfleeger recognize coupling as a pair-wise measurement of the modules. They have discussed about measuring the coupling on ordinal scale and they have classified the coupling in six pair-wise module relationships on ordinal scale [Fenton & Pfleeger 1998]. To measure coupling, Fenton and Pfleeger have presented an idea of set of classification of pair-wise relationships between modules x and y; starting from relation R_0, R_1, R_2 to R_n. Relations are subscripted from the least dependent at the start and the most dependent at the end, so that Ri >Rj for i>j. Modules x and y are said to be the loosely coupled if i value is somewhere in the start (near to zero) and modules x and y are said to be tightly coupled if i value is somewhere in the end (near to n). In [Fenton & Pfleeger 1998], they have not described their Model for measuring coupling in terms of OO paradigm.

Most of OO metrics to measure coupling are the counting metrics, which counts the number of times a class establishes an OO relationship with other class. Counting all types of OO relationships without distinguishing different types of interactions among the classes is not the best way to reflect to class coupling with other classes. In our opinion, class coupling can also be defined by distinguishing UML relationships (i.e. association, inheritance and dependency) among the classes.

1.1.3 Literature on Software Evolution

Parnas discusses that domain of a software system rarely remains same over the period of time [Parnas 1994]. One of the possible factors for the domain drift is the introduction of the similar products that causes the new requirements on existing products. There may be a number of other factors which bring changes in existing software system. Swanson points [Swanson 1976] out three types of changes corrective, adoptive and perfective.

Corrective: "Changes which are required to make software method correctly such as error and bug fixes".

Adaptive: "Changes which are made, in response to changing domain requirements".
Perfective: "Additional features, in form of new functionality".

According to the first law of software evolution [Lehman 1980], a certain software system stays useful as long as it shows ability to adopt with drifting domain; otherwise software system becomes less useful, and ultimately goes off the seen. And according to the second law of software evolution [Lehman 1980], structure of a certain software system becomes erosive while meeting the changes imposed by the evolution. First two laws of Lehman imply why evolution supportive software architecture is needed and how that architecture deteriorates when responding to evolution.

As described by the Lehman in the second law of software evolution, software architecture starts deteriorating while responding to the software evolution. This phenomenon of the architectural deterioration, because of the software evolution, is called software erosion. The software erosion result in form increasing complexity in the software architecture. Svahnberg also discusses that software complexity in the architecture also increase cost of maintenance of the software, as the effort required to determine how to implement a particular change increases with the complexity of the software architecture [Svahnberg et al. 2003].

Gurp and Bosch, based on an industrial case study, have discussed following reasons which lead to the software erosion [Gurp & Bosch 2002].

a. *Traceability of design decision:* Notions used during the software designing are hard to understand because of the lack expressiveness. Lack of expressiveness of design decision produces the traceability problems to reconstruct the design decision.

b. *Increasing maintenance cost:* Because of the growing complexity of the software architecture, sometimes, developers give up to understand the original architecture because either they find it too complex to understand or they realize high cost of maintenance, and then they go after less optimal solution.

c. *Accumulation of design decision:* New design decisions are made with the new releases. New decision and the previous design decision together make the overall architectural situation hard (costly) for the further requirement changes, if circumstances change.

d. *Iterative methods:* Gurp & Bosch identify the conflict between the designing for the incorporation of future changes and idea of iterative development models, which may introduce the new requirements (which are not considered as future requirement during designing) during the iterations. Gurp & Bosch say that a proper design should have knowledge about such changes in advance.

In our experiment, we will attempt to identify the software erosion by using our proposed metrics for class cohesion and class coupling.

1.2 Roadmap

Rest of the thesis is structured as follows: Chapter 2 presents our research questions. Chapter 3 of thesis describes the derivation process for the metric of class cohesion. In chapter 4 class coupling is presented based on UML relationships, namely, association, inheritance and dependency. This chapter also proposed a set of metrics to measure three types of coupling. Chapter 5 presents our methodology and planning for execution of experiment on the three versions of OO system using our proposed set of implementation metrics. Chapter 6 presents the results of the experiment and attempts to discuss the results. Chapter 6 also presents the limitation faced during our experiment on three version of source code. Chapter 7 of the thesis presents suggestions for future, based on the lessons learnt during the experiment. Thesis concludes its finding in chapter 8.

2 RESEARCH QUESTIONS

The purpose of this chapter is to describe and discuss our research questions. Our research questions are as follows.

? *What is class cohesion and how can it be measured?*

There are many metrics to find class cohesion but no standard metric or definition has been generally accepted, out of available [Fenton & Pfleeger 1998], [Counsell et al. 2002] and [Etzkorn et al. 2004]. A reasonable metric to measure class cohesion should give an insight to the relatedness among the methods of a class while considering the impacts of inheritance paradigm on local class cohesion. Frequency of attributes usage by the methods of class will be analyzed to measure class cohesion. We will also perform a literature review on the known metrics for class cohesion.

? *What is the class coupling and how can it be measured?*

Like class cohesion, there is no standard metric or definition for class coupling [Fenton & Pfleeger 1998]. However, In OO design class coupling is a measurement of class dependence on other classes. In our opinion, UML relationships among the classes provide the potential grounds to define a metric for class coupling. We will attempt to measure a class coupling on the basis of UML relationships.

? *What are the possible relationships which may exist between class cohesion and class coupling?*

Metrics for measuring class cohesion and class coupling are supposed to share same input data for their respective measurements. By the same set of input data we mean class member attributes, member methods, and usage of attributes by the methods. This sharing of input data strengthens our supposition about the existence of relationships between both metrics. We will attempt to find mutual relationships between class cohesion and class coupling metrics by analyzing the results of experiment statistically.

? *What types of results can be seen during the software evolution while using our set of metrics for class cohesion and coupling?*

Literature on the subject of the software evolution clearly introduces the erosive trends in the software architecture while meeting the changes imposed by the software evolution. In this thesis, we will attempt to identify such erosive trends with the help of class cohesion and coupling metrics. Based on the literature review, we suppose that both class cohesion and coupling should follow deteriorating trends while evolution in the software architecture. As a discussed by Bieman and Kang high cohesion and low coupling are always desired by software developers [Bieman & Kang 1995]. Consequently, undesired or deteriorating trends should be opposite to desired trends. We will attempt to find such trends by analyzing results from our experiment on three versions of OO implementations of system.

? *What is the relationship between class cohesion and class reuse?*

The reuse metric for the class *Alpha* is the number times class functionality is invoked in the methods of other classes through instances of class *Alpha*. We suppose that a class with higher value of cohesion is expected to be reused more than other classes in the OO system. In this thesis, we will try to verify our supposition about the class reusability and class cohesion. We will measure the class reuse in the context of private reuse- use of the class within a same software system [Bieman & Kang 1995].

? *What is the relationship between class size (NPM) and class reuse?*

As stated earlier by class reuse, we mean private reuse of class within the same software system. We will attempt to measure the class size by counting the public units of functionality of a class. Therefore, we have selected the number of public methods (NPM) as a measurement of class size. The public methods of a class act as carrier of functionality for the external classes. Each public method, in a class, can be thought as a unit of functionality. More public methods mean more units of functionality for the external classes. We suppose class reuse is directly related to class size. Because, more units of functionality increase the chances for a class to be reused more. We will try to verify this relationship between class reuse and size on the basis of results from our experiments.

? *What is the relationship between class cohesion and class size (NPM)?*

Both class cohesion and class size (i.e. number of public methods) are expected to support class reuse within the same OO design. Interestingly, by definition, we may expect decrease in cohesion with increase in the total number of public methods. Class objectives are met by the public methods and each public method may presents one complete objective or partial objective of the class. A class with larger size may present more objectives. More objectives through public methods may reduce the cohesiveness of a class. Based on stated expectations, it would be interesting to study cohesion and class size mutually.

3 COHESION

In this chapter, the derivation of a new metric for class cohesion is described. And, the derived metric is compared with TCC and LCC metrics.

3.1 Metric for Class Cohesion

The proposed metric for class cohesion measures the relatedness among the public methods of class on the basis of their relative contribution to overall public functionality of class. A new concept of subset tree has been used to determine the relative contribution of public methods to overall public functionality. Relatedness among the public methods of a class is determined on the basis of common usage of member attributes.

3.1.1 A Class in OO System

To describe the cohesion in term of OO paradigm of software development, we first need to understand how the construct of a class stands in OO paradigm. A class may be inherited from zero or more classes (*i.e. Multiple Inheritances in C++)* and a class may be derived by zero or more classes. An inherited class is also referred as base class or super class or parent class, whereas derived class is also referred as subclass or child class. Class is composed of its member(s). Term member stands for member attributes and methods of a class. Members of a class may have different access rules, based on the access rules; members can be public, protected and private. Private members can not be accessed directly from the outside of class; however pointer to private methods or attributes may cause violation to this rule. Protected and public members of inherited class become the part of derived classes and hence such members can be accessed by derived class. In C++, protected members of the class can also be accessed by the friend classes. However, public members of the class can be accessed by all other classes in the system.

The proposed metric for class cohesion can be used for any OO language. But, in this thesis, we have decided to express the metric for C++ language. We have only used those concepts from C++ that are generally supported by all OO languages and those concepts that are used in the selected system for experiment.

3.1.2 Inheritance and class cohesion

As stated earlier, In OO programming, a class can inherit the methods and attributes of other classes. For example, a class X can inherit the methods and attributes of other class Y if class X is the derived from class Y. This notion of OO programming makes the class Y as a subset of class X. In other words; this phenomena adds methods and attributes to derived class X from the base class Y. In our approach of measuring class cohesion, we will use relatedness among the public methods of a class. We can see that the notion of inheritance may add new members (i.e. protected and public attributes and methods) in the derived class and may produce an effect on the value of cohesion for the derived class. For measuring the class cohesion, we will add public and protected members of base class to the sub class and we will treat such derived members similar to other members of the class. Same approach is also used by [Dirk et al. 2000] and it is referred as flattening of derived classes. Dirk has reported the magnitude of changes caused by the inheritance on derived classes. He has observed considerable statistical variations in three types of OO metrics namely, cohesion, coupling and size before and after flattening [Dirk et al. 2000].

3.1.3 Scope rules and class cohesion

Scope rules attached with the members of a class decide the level of visibility of the methods and attributes of a class. According to [Bieman & Kang 1995], the cohesion is the measurement of the relatedness of visible methods of a class, therefore we will use the visible methods (i.e. public member methods) to measure the cohesion of the class. Public methods, being visible to outside users, work as carrier of functionality, therefore only their cohesiveness contributes to overall cohesion of the class. Although invisible methods can be used to identify the related groups of public methods if invisible methods are used by the public methods.

Global variables can also used for finding relatedness among the members of the class. However, we will not use them in our experiment to reduce the complexity of our work.

3.1.4 Methods to expel

For measuring the class cohesion, we will only use those public methods which are neither constructor nor destructor for a class. Constructor and destructor methods only serve the purpose of initialization of class attributes and they are not supposed to perform any kind of functionality for consumer environment. Bieman and Kang have also ignored the constructor and destructors methods while deriving TCC and LCC metrics for class cohesion [Bieman & Kang 1995]. Similarly to constructor and destructor methods, operator overloader methods will also be ignored because such methods read or write to all attributes of class and consequently act like an initializing method. Inclusion of operator overloading may mislead value of class cohesion metric. In the rest of this chapter, the term public methods will mean public methods excluding the constructors, destructors and operator overloaders.

3.1.5 Set representation of class

We can express public methods and all attributes of class X in a form of a set S(X); such that set S(X) represents the union of set M(X) and A(X) in following manner.

$$S(X) = M(X) \cup A(X)$$

Set M(X) contains public methods of class X, whereas the set A(X) contains the all attributes of the class X regardless of their scope and type.

M(X) = {m1, m2, m3} set of member public methods
A(X) = {a1, a2, a3} set of all member attributes

3.1.6 Group determination in a class

A group of methods can be identified by determining the similarity of work in the public methods of the class. For identifying the relatedness of work, we will use the set M(X) that represents all public methods in class X.

Methods in set M(X) can form different subsets of M(X) containing different numbers of methods as their elements. We will call these subsets of M(X) as groups and we will represent them with Gi.

In set M(X) following are the possible groups excluding the empty set.

G1= {m1}	*G1 is subset of M(X)*
G2= {m2}	*G2 is subset of M(X)*
G3= {m3}	*G3 is subset of M(X)*
G4= {m1, m2}	*G4 is subset of M(X)*

G5= {m2, m3}	*G5 is subset of M(X)*
G6= {m3, m1}	*G6 is subset of M(X)*
G7= {m1, m2, m3}	*G7 is subset of M(X)*

We supposed each of these groups is representing a functionally group by making the set of methods with respect to certain commonality. A coherent class is expected to represent minimum number of functionality groups through its public methods. Different approaches have been used to find commonality among the methods of class. In design metrics for class cohesion (such as CAM) the commonality among parameter in the parameter list is a choice but in the implementation metrics for class cohesion common usage of member attribute by the member methods is considered as better choice. Usage of common attribute provides a clearer insight to a method than the common parameter in the parameter list.

In our opinion, a class that is composed of related methods will result coherent as its methods will use the same set of attributes in their implementation. According to this idea of measuring of class cohesion; a class that has single public method will turn out be the most coherent, because all of its attributes will be used by the one public method. Bieman and Kang have also called such class as the most cohesive class [Bieman & Kang 1995].

Following describing some of the possible approaches for finding functionality groups of public methods.

- Methods working on a particular member attribute of class may form group of related functionality.
- Methods using a particulars external resource such as file, database table and memory may form a group of related functionality.
- Methods using same private, protected and public method may also form group of related functionality.

Above are just the hints for finding a group of related public methods. An analyzer may also use his domain knowledge to determine the group amongst the public methods of the class. They may be several reasons to identify the groups of related methods in a class. We will use the approach of common direct or indirect usage of attribute by the public methods to identify the groups of related functionality. This approach is adopted from [Bieman and Kang 1995]. A direct usage of an attribute is identified when a method reads or writes to an attribute directly, and an indirect usage of an attribute is identified when a method calls another method that directly reads or writes to an attribute.

Let's take the example of Java based class *Queue* to identify the similar groups of public methods. Class *Queue* is consisted of four member private attributes, namely count, front, end and temp and four public member methods, namely empty, count, append and serve.

```
class Queue {

      private:

      int count=0;
      Node front;
      Node end;
      Node temp;

      public:
      bool empty(){ return(count==0); }
      int count() {   return count; }

      void append(object obj){
           if(count==0){
                 front = end = new Node(obj, front);
                 }
                 else {

                 end.Next = new Node(obj, end.Next);
                 end = end.Next;
                 }
                 count++;
           }
      object serve(){

                 temp = front;
                 if(count == 0)
throw new Exception("tried to serve from an empty Queue");
                 front = front.Next;
                 count--;
                 return temp.Value;
       }

}
```

In the figure 3-1, we have shown the public methods with rectangles and attributes with ovals. A link between a rectangle and an oval is representing the usage of attribute by a method. For instance, in figure 3-1, there are four lines which are connecting attribute count with four different methods i.e. count, empty, append and serve. That means attribute count is used by four methods.

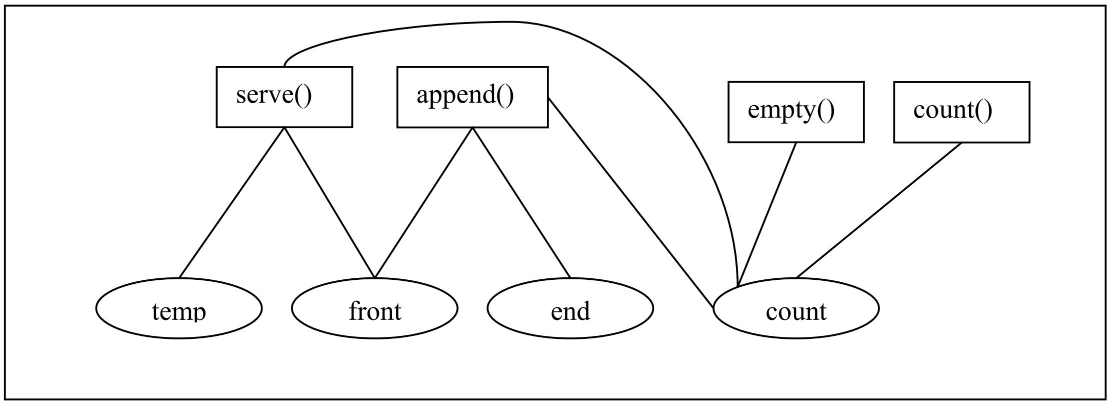


Figure 3-1

From the figure 3-1, we can identify following groups.

Group 1: Attribute temp is used by serve ()
Group 2: Attribute front is used by serve () and append ()
Group 3: Attribute end is used by append ()

Group 4: Attribute count is used by serve (), append (), empty () and count ()

We can also express the groups as follows:

G1 = {serve ()}
G2 = {serve (), append ()}
G3 = {append ()}
G4 = {serve (), append (), empty (), count ()}

In order to calculate the class cohesion of class Queue, we will first measure the relatedness of each method with the identified groups. We define the relatedness of methods with identified groups in following way.

Relatedness of public method M with identified group(s) is a ratio between number of its occurrences in all group(s) to total number of identified groups in class.

Relatedness R of Method M is given by the following formula.

$$R(M) = \frac{MO}{TG} \qquad \text{Equation 3-1}$$

Where MO represents number of times a method M occurs in all groups and TG represents total number of groups.

In following, we have calculated the values of relatedness for methods of class *Queue*.

R (append) = 3/4 = 0.75
R (count) = 1/4 = 0.25
R (empty) = 1/4 = 0.25
R (serve) = 3/4 = 0.75

Measurement of relatedness with identified groups for each method is an integral part of class cohesion; therefore we suppose the class cohesion as an average measurement of relatedness *R (M)* of all public methods and we define it by following expression for a class X.

$$Cohesion(X) = \frac{\sum R(M)}{TM} \qquad \text{Equation 3-2}$$

TM stands for total numbers of public methods in class X and R (M) stands for the measurement of relatedness for public method of class X. In following, the class cohesion for class Queue is calculated using the defined formula.

Cohesion (Queue) = (0.75 + 0.25 + 0.75 + 0.25) / 4 = 0.50

Class cohesion value for class *Queue* indicates the problem in our supposition of metric for Class Cohesion (Equation 3-2), which we have presented so far. In fact, we have neglected the subsets formation among the identified groups of public methods. Neglecting of subsets formation amongst the groups has affected the results of class cohesion. We were considering occurrence of method in all groups equally. Occurrence of methods in groups can not be weighted equally, because some groups are more associated with the overall public functionality of class and some groups are less associated with overall public functionality of class. Therefore, to measure class cohesion correctly, we need a mechanism to measure the relative relatedness of public methods to overall public functionality of class. To understand the relative relatedness of each public method, we will describe the idea of subset tree in following section.

3.1.7 Subset tree formation

A subset tree is composed of nodes representing sets with public methods as their elements. Such sets group the public methods using a same attribute. Identified sets are placed in relationship of subset to parent nodes in the hierarchy of subset tree. Using a subset tree helps in determining relative contribution of method to overall public functionality of a class.

From the example of the class *Queue*, we have identified following groups of related methods.

G1 = {serve ()}	*G1 is subset of G2 and G4*
G2 = {serve (), append ()}	*G2 is subset of G4*
G3 = {append ()}	*G3 is subset of G2 and G4*
G4 = {serve (), append (), empty (), count ()}	*G4 is subset to none*

As shown above groups are forming subsets, we can also show subset formation with help of a subset tree. In a subset tree, every node represents a group and child to that node represents the subset of the node. Figure 3-2 shows a subset tree for the groups of class Queue.

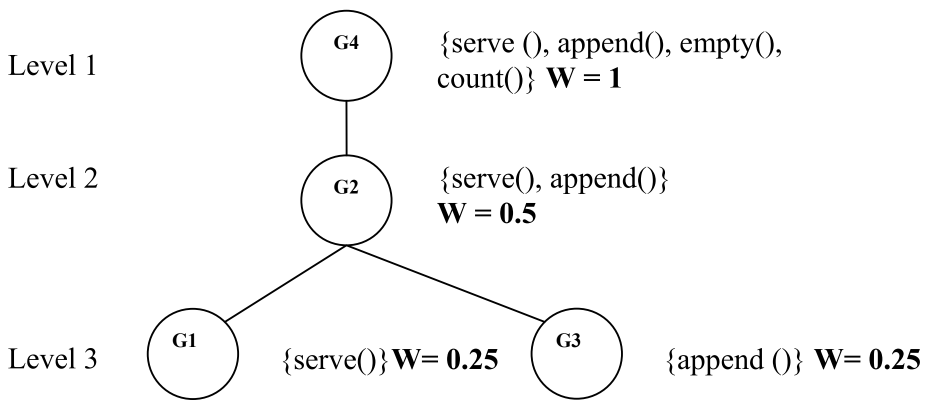

Level 1
G4
{serve (), append(), empty(), count()} W = 1
Level 2
G2
{serve(), append()} W = 0.5
Level 3
G1
{serve()}W= 0.25
G3
{append ()} W= 0.25

Figure 3-2

Subset tree in figure 3-2 is showing how different groups are coming on different level in the tree. The Group G4 on root level is weighted highly connected with the class as it contains all the public methods in it. The group G2 at the Level 2 is comparatively less connected with the class, whereas the groups G1 and G3 have a comparatively lower association than G2 and G4 with the class. As you can see, from the figure 3-2, we have assigned the weights Wi to all group Gi (nodes) in the subset tree using following equation 3-3.

$$\text{Weight of group G} = \frac{\text{Number of methods in a group}}{\text{Total number of public methods in subset tree}} \qquad \text{Equation 3-3}$$

Our previous definition for measuring the relatedness of a method R (M), in section 3.2.2, is ignoring the effect of relative association of methods with class. Now, we will redefine the formula for relatedness of methods. To redefine relatedness R of method M; we will first define S (M) the Sum of weight for a method M based on M appearance in groups of a subset tree.

$$S(M) = \sum WG(M)$$ Equation 3-4

WG (M) stands for the weight Wi of group Gi for a method M on it occurrence in a group Gi. If we recall the example of class *Queue* and keep the weights of subset tree in mind. The values of S(M) for method "append" will be following:

S (append) = WG2 (append) + WG4 (append) + WG3 (append) = 0.50+ 1.0 + 0.25 = 1.75

Method append occurs on three groups, namely G2, G3 and G4, and therefore, we have added the weights of these groups for method "append" to calculate S (M). We have followed the same guidelines for calculating the S(M) values for other three methods of the class *Queue*.

S (count) = WG4 (count) = 1.0
S (empty) = WG4 (empty) = 1.0
S (serve) = WG1 (serve) + WG3 (serve) + WG4 (serve) = 0.50 + 0.25 + 1.0 = 1.75

Every subset tree in a class contributes to overall functionality of the class and every method in the subset tree contributes to overall functionality presented by the subset tree, but such contribution may not be equal on both levels. If we just consider the contribution of method for a subset tree, we can argue that method which has highest S(M) value in the subset tree is the most related method to the overall functionality presented by the subset tree. Based on this argument, we redefine the relatedness of method as a relative measure to maximum S (M) for method in subset tree. Therefore, we have divided the S(M) for a method M by the maximum SM for a method in a given subset tree. The formula for measuring the relatedness of public methods can be defined using equation 3-5.

$$R(M) = \frac{S(M)}{\text{Maximum of S(M) for all methods in a subset tree}}$$ Equation 3-5

Based on the formula; we have calculated the following values of relatedness for all methods in the class Queue.

Maximum of S (M) for all methods in the subset tree is 1.75
R (append) = 1.75/1.75 = 1.0
R (count) = 1.0/1.75 = 0.57
R (empty) = 1.0/1.75 = 0.57
R (serve) = 1.75/1.75 = 1

For measuring the cohesion for the subset tree, we will calculate the average measurement of relatedness R(M) of all methods in the subset tree with the help of following equation 3-6.

$$Cohesion(SubsetTree) = \frac{\sum R(M)}{TM}$$ Equation 3-6

TM stands for the total number of methods in subset tree. For the class Queue we will get following results:

Cohesion (Sub Tree) = (1.0+0.57+0.57+1.0)/4 = 0.785
Or
Cohesion (Class Queue) = (1.0+0.57+0.57+1.0)/4 = 0.785

In the case of class *Queue*, we have only one subset tree, therefore the cohesion of subset tree is actually the cohesion of class *Queue*. This value of cohesion for the class *Queue* helps us to recognize it as a cohesive class. The difference between our previously calculated value (0.50) for cohesion of class *Queue* and this value is because of weights, we have assigned to each group in the subset tree. And, previously, we have ignored the fact that each group contributes to overall functionality of a class differently.

3.1.8 Multiple subset trees in a class

In the example of class *Queue*, we had only one subset tree. But, there is big possibility that a class may results into multiple subset trees. To explain, our method of calculating class cohesion under such scenarios; we have used figure 3-3. Figure 3-3 represents a class *alpha*. That results into seven groups, namely G1, G2, G3, G4, G5, G6 and G7 containing the methods, namely m1, m2, m3, m4 and m5. Here, we suppose that we have already used the attributes of class *alpha* to form seven groups.

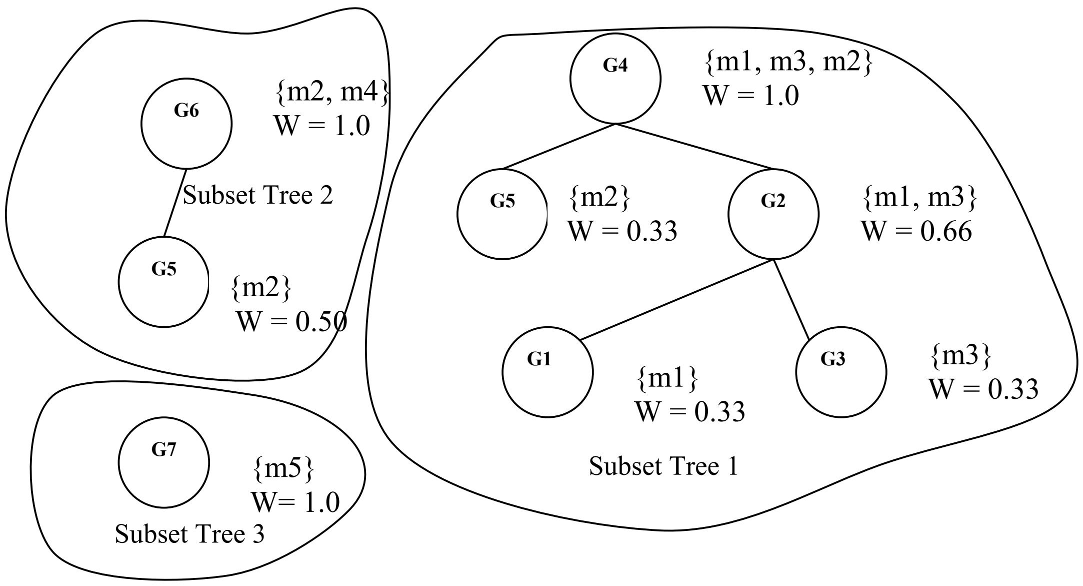


Figure 3-3

Figure 3-3 shows that class alpha has resulted into the formation of tree subset trees.

In the Table 3-1, we have calculated the cohesion values for the all three subset trees of class *alpha*.

Subset Tree 1	S(M)=∑GW (M)	R(M)	Cohesion (Subset Tree 1)
m1	1.0+0.66+0.33	1.0	2.67/3=0.89
m2	1.0+0.33	0.67	
m3	1.0+0.66+0.33	1.0	
Subset Tree 2	**S(M)=∑GW (M)**	**R(M)**	**Cohesion (Subset Tree 2)**
m2	1.0+0.5	1.0	1.66/2=0.83
m4	1.0	0.66	
Subset Tree 3	**S(M)=∑GW (M)**	**R(M)**	**Cohesion (Subset Tree 3)**
m5	1.0	1.0	1.0

Table 3-1

Overall class cohesion is dependent on the cohesion of subset trees in a class. A subset tree in a class represents partial public functionality of the class. Through, the representation of this public functionality, a subset tree contributes to the overall cohesion of class. We argue that not all subset trees contribute to the overall class cohesion equally. Based on this argument, we have assigned the weights to all subset trees by dividing the total number of public methods in a subset tree with the sum of total of the public methods in all subset trees using equation 3-7.

$$WST(SubsetTree) = \frac{NM(SubsetTree)}{\sum NM(SubsetTree)}$$ Equation 3-7

NM stands for number of methods in a subset tree and WST stands for the weight of a subset tree.

Table 3-2, shows the values of assigned weights of subset tree for the class alpha.

	NM (Subset Tree)	WST (Subset Tree)
Subset Tree 1	3	3/6=0.5
Subset Tree 2	2	2/6=0.33
Subset Tree 3	1	1/6=0.167

Table 3-2

Based on weights assigned to subset trees of class alpha in the table 3-2, we can also calculate the relative cohesion of subset trees just by multiplying the weight of subset tree to its cohesion value calculated in table 3-1. Equation 3-8 calculates the relative cohesion RC for subset tree.

$$RC(SubsetTree) = WST(SubsetTree) \times Cohesion(SubsetTree)$$ Equation 3-8

Table 3-3 shows the relative cohesion of the subset trees in class *alpha*.

	WST (Subset Tree)	Cohesion (Subset Tree)	RC (Subset Tree)
Subset Tree 1	3/6=0.5	0.89	0.445
Subset Tree 2	2/6=0.33	0.83	0.27
Subset Tree 3	1/6=0.167	1.0	0.167

Table 3-3

Relative cohesion calculated in table 3-3 for all subset trees, is showing how much each subset tree is contributing to the overall class cohesion. Here, we need to device a formula that can integrate all of relative cohesions based upon their contribution to overall class cohesion.

As shown by the values of relative cohesion in table 3-3, all subset tress have different relative cohesion. Here, we can also calculate relative cohesion ratio RCR between the relative cohesion RC for a subset tree and maximum relative cohesion MRC from all subset trees by applying following equation 3-9.

$$RCR(SubsetTree) = \frac{RC(SubsetTree)}{MRC}$$ Equation 3-9

Following are the measurements of RCR for all subset trees in class *alpha*.

MRC = 0.445 Maximum RC for all subsets
RCR (Subset Tree 1) = 0.445/0.445 = 1.0
RCR (Subset Tree 2) = 0.27/0.445 = 0.60
RCR (Subset Tree 3) = 0.167/0.445 = 0.37

Sum of relative cohesion SRC of all subset trees can be calculated by following equation 3-10:

$$SRC(Class) = \sum RC(SubsetTree)$$ Equation 3-10

SRC (Class) = RC (Subset Tree 1) + RC (Subset Tree 2) + RC (Subset Tree 3)

As stated earlier to measure the overall class cohesion, we need a formula that can integrate the relative cohesions of all subsets tree in a class. The value of SRC (sum of relative cohesions), we have just calculated, does not integrate correctly to represent overall class cohesion for class *alpha*. SRC does not represent overall class cohesion, because it does not show the effect of three subset trees representing three functionalities. To make it correct for overall class cohesion, we can not even take the average of RC of all subsets trees by dividing SRC with total number of subset trees. Because, dividing the SRC with total number of subset trees, implies that all subset trees are contributing equally to overall class cohesion. That is against to argument that all subset trees may contribute differently to overall class cohesion. To measure the overall class cohesion, we can divide the SRC by sum of the relative cohesion ratios. Dividing the SRC by the sum of relative cohesion ratios RCR of subset trees is in accordance with argument. Following equation 3-11 calculates the cohesion of class.

$$Cohesion(Class) = \frac{SRC}{\sum RCR} = \frac{\sum RC(SubsetTree)}{\sum RCR}$$ Equation 3-11

For the given class *alpha*, the measurement of over class cohesion is as follows:

Cohesion (alpha) = (0.445 + 0.27 + 0.167) / (1.0+ 0.60 + 0.37) = 0.88/ 1.97 = 0.45

The value of cohesion, we have calculated for the class *alpha* seems correct indicator for the class cohesion, because class *alpha* is containing very divergent groups of functionality that are forming three subset trees. Cohesion measurement for such classes should be low.

3.1.9 Comparison with TCC and LCC metrics

As mentioned earlier in Related Work, that TCC and LCC metrics are found to be the best among the known metrics for class cohesion. To compare TCC and LCC metrics with our proposed metric, we will present a comparison using following scenario.

Suppose, we have got a number of classes with all non-connected public methods i.e. pair of connected methods on basis of common usage of an attribute can not be found in these classes, because all the methods in classes are using only one member attribute. In other words, no public method can use more than one member attribute in any of given class. Table 3-4 shows the results of metrics calculation under mentioned scenario using TCC, LCC and our proposed metrics for class cohesion.

Number of Public methods in class	TCC	LCC	Our Proposed metric for Class Cohesion
1	1.0	1.0	1.0
3	0	0	0.33
4	0	0	0.25
50	0	0	0.02

Table 3-4

Table 3-4 shows that TCC and LCC metrics treat a class with larger size (i.e. number of public methods) or smaller size (more than one public method) equally when there exist no pair of connected public method, whereas our proposed metric for class cohesion not only differentiate among the classes on the basis of their respective class sizes but it also results into lower value of class cohesion because of non-connected methods for all given classes. This property of our metric for class cohesion proves it better than TCC and LCC metrics. This property also makes it closer to human-oriented view of class cohesion. When all public methods are non-connected in all classes, a human-oriented view of class cohesion is supposed to rank the classes more cohesive having relatively lower number of non-connected public methods than other classes with higher number of non-connected public methods.

Resulting into zero by TCC and LCC metrics, when there is no relatedness among public methods, has proved them the metrics that only measure the relatedness on the basis of common attribute usage. Our proposed metric for class cohesion measures the relative relatedness of public methods to overall public functionality of a class; therefore, it has resulted into non-zero value when all public methods are non-connected. Our metric for cohesion can only result into zero when there is no public method in the class or none of the public method uses any member attribute.

4 COUPLING

The aim of this chapter is to present the notion of class coupling based on UML relationships that are found in OO design.

4.1 Class coupling

In OO design, the coupling of a class means the measurement of the interdependence of class with the other classes. In a design of reasonable size (say design size is ten classes); normally classes do not exist in absolute isolation. By going through any OO source code of a working system, one can see that nearly all classes have some kind of relationships with other classes in the design. These relationships, among the classes, create pair-wise interdependencies. Such pair-wise relationships among the classes are the results of design decisions which are made on the specifications of the system. A good design decision may create a good relationship and a bad design decision may create a bad relation. Here, by good design decision, we mean a decision that makes the OO design easy to reuse, understandable and flexible for modification and adoption in future. Gurp and Bosch, based on an industrial case study, have presented five reasons for software erosion, which revolves around the design decisions made on different stages [Gurp & Bosch 2002]. During such stages designer team of the system decides which relationship should be used to fulfill the goals (specifications and constraint) of a particular system. Based on the goals of the system and the design skills, a team may design a system that exhibits low or high coupling among the classes.

4.2 UML relationships

As mentioned earlier, very few classes in OO design stand alone. Relatively a large number of classes collaborate with others in OO design. Therefore, while modeling OO design, the relations are also modeled based on how classes stand to each other [Booch el at. 1999].

According to [Booch el at. 1999], in OO modeling there are three types important relationships among the classes, namely dependency, inheritance and association relationships. Dependency represents the using relationships among the classes; inheritance relationship connects generalized classes to their specialized classes; and association relationship shows the structural relationship among the objects. In following section, we will discuss these relationships briefly.

4.2.1 Dependency relationship

Dependency relationship is a using relationship. In UML, it comes among the classes, when one class uses the functionality of other class through instances other. In OO implementation, class can use other classes in following ways:

- A class uses an object of other class as a parameter in one of its member method. Parameter dependency can be located by looking at class declaration as it is shown in figure 4-1.

```
class A {
….
public:
void methodA( B param );
void methodB();
…..

};
```

```
class B {

……
……

};
```

Figure 4-1

In figure 4-1; a methodA of class A is using the parameter of class B.

- A method of one class uses the object of other class as its local variable. This type of dependency can be identified by going through the method implementation. Local variable dependency is shown in figure 4-2 a.

```
A::methodB()
{
      B LocalVarTypeB;
      …….
      …….
}
```

Figure 4-2a

```
B & A::methodC()
{
      …….
      …….
      return B();
}
```

Figure 4-2b

In figure 4-2 a, a method of class A is using the variable of class B in its implementation.

- A method of one class uses the object of other class as a return parameter. This type of dependency can be identified by going through the method implementation. Return parameter dependency is shown in figure 4-2 b.

A dependency relationship among the classes is shown by the dashed line with arrow head. Where tail points to dependent class and head points to the class whom a class is dependent on. In the example, class A is dependent on class B, because class A is using class B. Only evidence of dependency from the mentioned types is enough to recognize the dependency relationship between the pairs of classes.

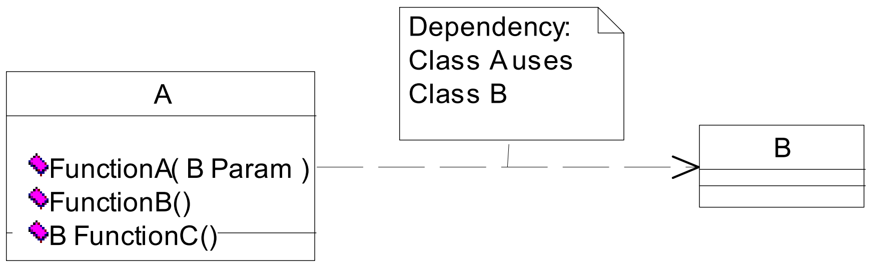


Figure 4-3

4.2.1.1 Metrics to calculate dependency

To measure dependency coupling we have found following metrics:

- *Number of used classes by dependency relation (NUCD):* In this measurement we will count the number of distinct classes with whom a particular class *alpha* is creating dependency relation. Only one evidence for dependency relation would be enough, caused by any of dependency types (e.g. parameter, local variable, return type) to recognize the dependency between two classes. One or more dependency evidences from a class *alpha* to class *beta* will increase the counter of this metric by one.

- *Total number of evidences for 'Used classes by dependency relation' (TNUCD):* This measurement will be used to count total number of evidences for a particular class *alpha* of *'Used classes by dependency relation'.* All types of dependencies (e.g. parameter, local variable, return type) will be used to count such evidences. Counter for this measurement will be increased by one with every found evidence for dependency.

- *Ratio of NUCD to TNUCD (RNUCD):* This metric measures the ratio between TNUCD and NUCD only for the classes where NUCD count is bigger than one. Higher value of ratio will indicate tightly coupled class and lower value of this ratio will indicate a loosely coupled class. It is given by following expression:

$$RNUCD = \frac{TNUCD}{NUCD} \qquad \textit{Where } NUCD > 0$$

- *Number of user classes for a class through dependency relation (NUCC):* This measurement represents the total number of distinct classes who are using a particular class *alpha* through dependency relations.

- *Total number of evidences for 'User classes through dependency relation' (TNUCC):* This measurement counts the total number of usage evidences of a particular class alpha by the other classes in OO design.

- *Ratio of NUCC to TNUCC (RNUCC):* This metric measures the ratio between TNUCC and NUCC only for the classes where NUCC count is bigger than one. It is given following expression.

$$RNUCC = \frac{TNUCC}{NUCC} \qquad \textit{Where } NUCC > 0$$

4.2.2 Generalization relationship

According to [Booch el at. 1999], most of the time generalizations are used among the classes and interfaces to show inheritance relationship. Generalization is called is-kind-of relationship. This type of relationship exists among the general kind of classes and their more specific kind of classes. In this way, the relationship links generalized class with specialized class. By a specialized class we mean derived class which is also referred as subclass or child class. And, by generalized class we mean base class or parent of the derived class. During OO modeling, the generalized relationships are established when one class is found as a more specific kind of other class. Notion of the generalization is also referred as inheritance. Dirk has reported the significance of inheritance or generalization relationship and he has emphasized that OO metrics like cohesion, coupling and size should be studied while considering the effects produced by the inheritance [Dirk et al. 2000].

In the UML, generalization relationship can also be created among the packages. To explain generalization more we have taken figure 4-4 from [Booch el at. 1999] that shows a classical example of generalization among the basic shapes of geometry.

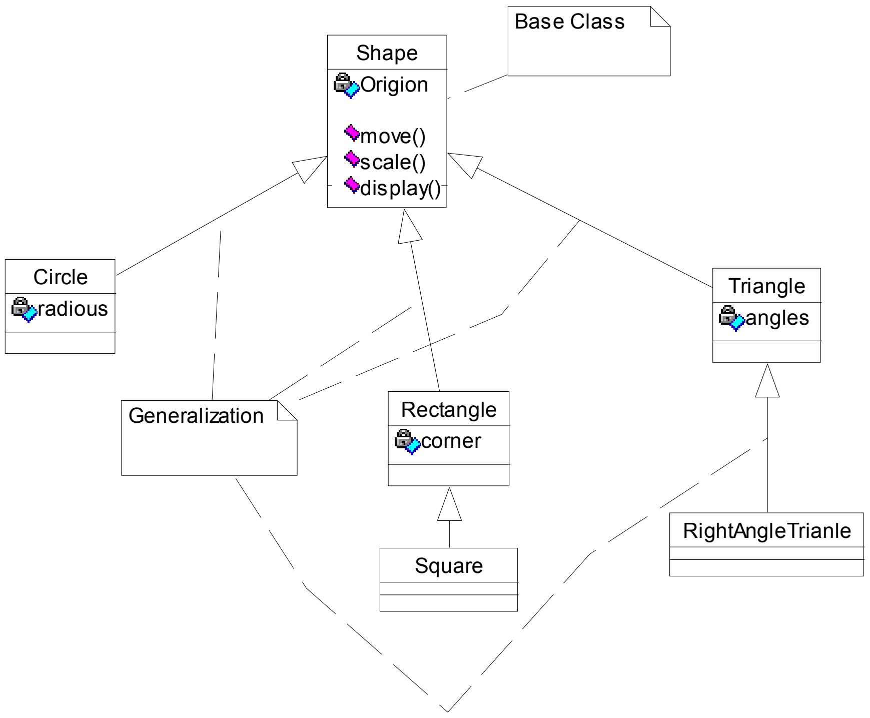


Figure 4-4

4.2.2.1 Inheritance of Interface or Realization

In OO programming languages, classes can also inherits an interface. This type of inheritance is called realization relation. According to [Booch el at. 1999], realization is kind of contract among two classifiers that one classifier specify and other classifier assures to carry out. In the UML, realization relationship is shown a dashed line with an arrowhead as shown by the following figure.

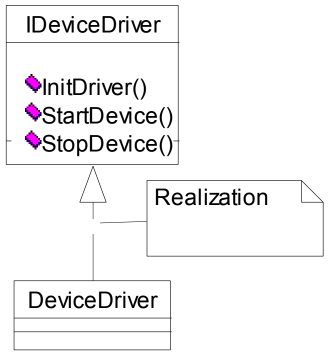


Figure 4-5

Figure 4-5 is showing the realization relationship between the interface IDeviceDrive and the class DeviceDriver that implements the interface specified by DeviceDriver.

All type of inheritance or generalization relationships are easy to understand and they can be identified from the UML class diagram and as well from OO source-code.

4.2.2.2 Metrics to calculate Inheritance

To calculate coupling caused by inheritance we have found following metrics:

- *Depth of class in inheritance tree (DCI):* This metric calculates the depth at which a class exists in inheritance tree. For the root class in inheritance tree DCI count is 0. In case of multiple inheritances, DCI metric adds 1 to DCI count of all immediate parents and then adds these sums to its own DCI count.

- *Number of derived classes from a class (NDC):* This metrics calculates the number of derived classes from a class by counting all of its child classes in inheritance tree.

4.2.3 Association relationship

Association is kind of structural relationship, in which classes are connected to each other by playing some type of role. Association is also kind of has-a relationship, in which a class contains the other class in order to play certain role for each other [Booch el at. 1999].

Let's take the example of class employee and company. Class employee plays the role of worker for the class company. Class Company has many employees in it, for them Class Company plays the role of employer.

In the UML, association relationship among the classes is depicted by a line as shown by the figure 4-6.

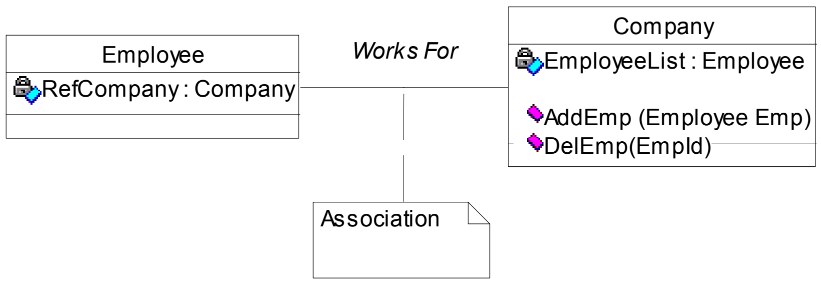


Figure 4-6

Figure 4-6 shows that Employee works for a company. And company knows all the employees it contains. Employee also contains a reference of the company from whom it works for. Figure 4-6 also shows that more than one employee may work for the company under this association.

4.2.3.1 Aggregation Relationships

Association relationship has a special kind which is called aggregation. This kind of has-a relationship is also called whole relationship [Booch el at. 1999]. This type of relationship models the situation where one big part is consisted of smaller parts. In the UML, aggregation is specified with help of line ending at empty diamond. Empty diamond points to container class and tail points to contained class. Aggregation relationship can be depicted with the example of school that has departments in it.

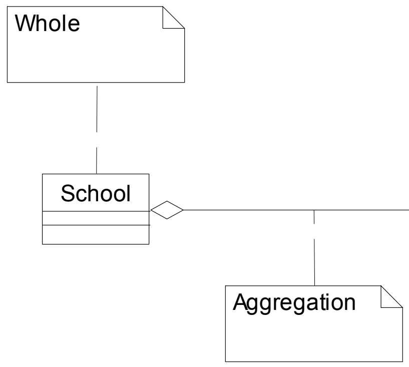


Figure 4-7

Figure 4-7 is showing an aggregation relationship between class School and Department where School contains the departments as its parts.

4.2.3.2 Composition relationship

Composition relationship is stronger kind of aggregation relationship. In which, container and contained object are constructed and destructed together [Booch el at. 1999]. In the composition relationship, the whole is responsible for managing the creation and destruction of its parts. In the UML, composition relationships are shown by a line with black diamond. Diamond point to whole or container whereas the tail of line starts from the contained part.

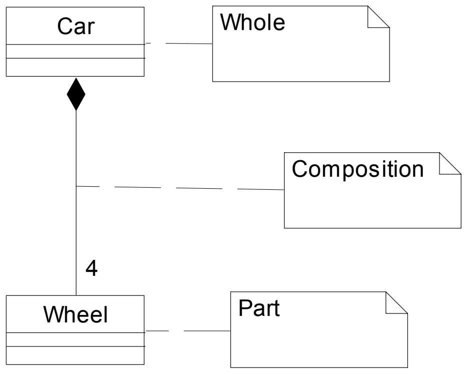


Figure 4-8

Figure 4-8 shows the composition relationship between container class Car and its part Wheel. Class Car will contain its four Wheel objects by value. And Car Wheel will also manage the creation and destruction of its wheels with its own creation and destruction.

Since, it is hard to distinguish among association, aggregation and composition relationships from OO source code; therefore, for measuring the association coupling, we will treat all types of association coupling equally. Association, aggregation and composition relationships can be selected alternatively in the design that also justifies our decision to treat them equally. Following table shows the conversion of association relationship to aggregation and composition relation.

Designing Concept	UML Relation
Employee works for the company	Association
Employee is a part of the company	Aggregation
Employee is owner of the company therefore Employee and company will exist together	Composition

Table 4-1

4.2.3.3 Metrics to calculate association

To calculate the coupling caused by association relations we have found following metrics:

- Number of associated classes with a class (NAC): This metric gives the number of associated classes with a particular class. All kind of associations (e.g. association, aggregation and composition) will be used to count the number of associated classes.
- Total associated class Usages (TACU): This metric gives the number of times all associated attributes of a particular class type are used by methods of a user class.

4.2.4 Dimensions of class coupling

Based on the types of UML relationships, we have found three types of coupling among the classes namely, dependency, inheritance and association. A class in OO implementation may have all such types of coupling with other classes. This observation makes the coupling a three dimension function that can be expressed by something like following equation.

$$\text{Coupling}(\text{Class}) = a\,A_C + b\,D_C + c\,I_C$$

Where a, b and c are the coefficients of unknown values and *A_{c,} Dc and Ic* are representing association, dependency and inheritance types of coupling respectively. We are also not sure about the degree of stated equation. Therefore, we are unable to form an equation based on three dimensions of coupling.

We can imagine intuitively all dimensions of coupling have a different contribution to over class coupling. But the weights of their contributions by the types of coupling are undetermined. Undetermined weights also hinder us to define an ordinal scale classification for measuring the overall class coupling.

5 RESEARCH METHODOLOGY

The purpose of this chapter is to introduce the input system FileZilla and to describe our strategy for performing the experiment on FileZilla source code with the short description of methods used for data acquisition and metrics calculation. We have also mentioned the planned set of metrics for performing experiment briefly.

5.1 FileZilla System

FileZilla is the name of an open source system that we have used as an input to perform our metrics calculation. FileZilla is a FTP client for Windows operating system. FileZilla has been developed using OO programming in MS Visual C++ 6.0. This system is implemented using MFC framework for Dialog based application. This open source system is available on www.sourceforge.net. FileZilla system was started in year 2003, and up till now eighteen versions of source code have been published on system website. Class count for our selected versions remains eighty to eighty five per version. Reasonable number of classes in FileZilla, easy access to its source-code and availability of its different versions with the release notes made us select FileZilla as an input system for our planned experiment. We have selected three non-consecutive versions of Filezilla (Version 2.1.6, Version 2.1.8 and Version 2.2.5). The reason to select these three non-consecutive versions is because of our research question to study the effects of software evolution using our proposed metrics for class cohesion and class coupling.

Number of MFC derived and non-MFC derived classes are almost same for all selected versions. A few new classes have been added and some of the old classes were dropped from our first Version 2.1.6 to last selected Version 2.2.5. Most of the changes are performed in the internal implementation of classes in form of code modification and addition. These changes, among the subject versions, are also reflected through class cohesion and class coupling metrics in our study on software evolution.

5.2 Data Acquisition

One of the most difficult and time taking part of the experiment was the data acquisition. In this part of the experiment, we have inserted all types of information from OO source code to a database file of MS Access. Table 5-1 shows the type of OO information we have stored for a hypothetical class *Alpha* that is derived from class Beta.

Class Declaration Information
• Class *Alpha* <Derived From Class *Beta*> o Member Attribute ▪ Attribute Type ▪ Scope of Attribute (e.g. public, private and protected) o Member Method ▪ Parameter List ▪ Return Type ▪ Scope of Method (e.g. public, private and protected)
Class Implementation Information
• Variable used by Member Method o Used Member Attribute of the class o Used Derived Attribute from the base class (e.g. public and protected variable) • Types instantiated local variable in the implementation of member method • Type of return variable by the member method of class • Variables used in parameter list in the member method of class
Table 5-1

E/R Diagram for database

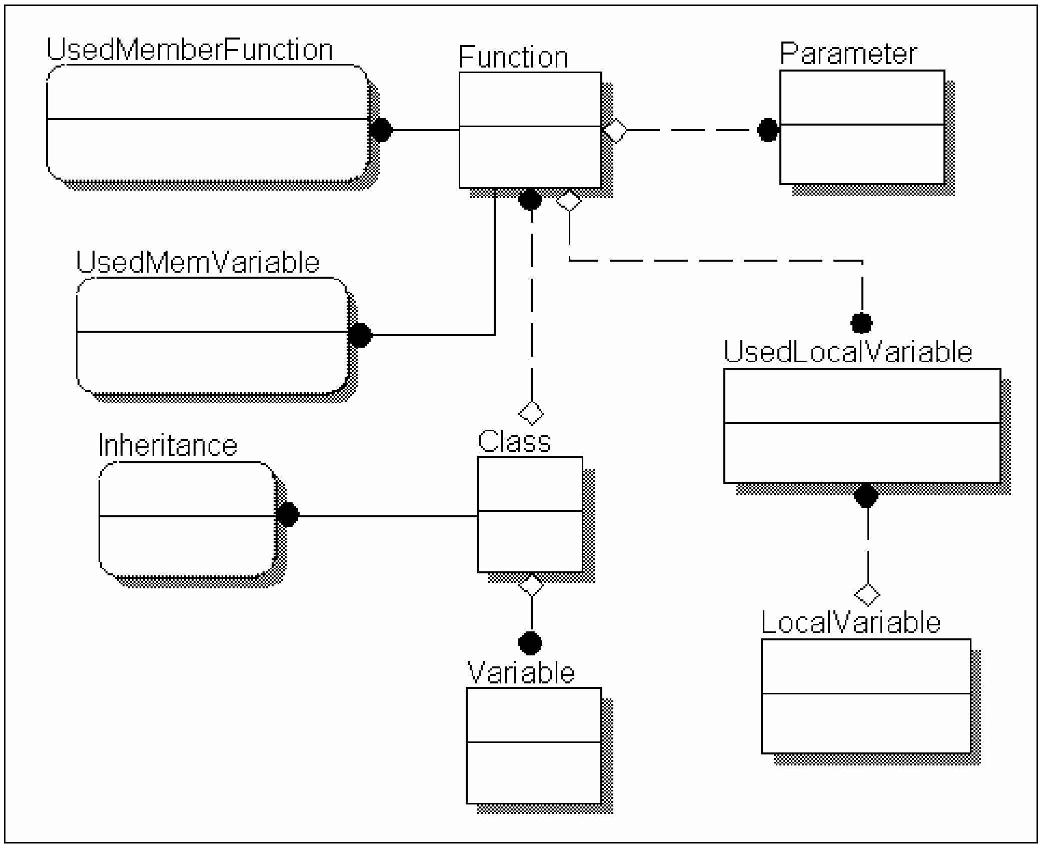


Figure 5-1

To store all OO information, we have created the database according to the ER diagram in figure 5-1. Many other tables are also added to the database on the need basis on the different phases of data acquisition and metrics calculations.

Our complete database file can be downloaded from following URL http://www.student.bth.se/~imba03/msthesis/experiment.zip. To understand this database file for the reuse purposes, a full fledged manual is needed that we are not providing because of time limitations. Results of metrics calculation in tabular form can be accessed using URL: http://www.student.bth.se/~imba03/msthesis/results.htm

5.2.1 Method used for data acquisition

To gather and analyze the data different methods have been used on the different phases of the data acquisition and analysis. We can classify these methods in to two types, namely manual and automated methods. A short description of these methods is as follows:

Manual Methods: At the start of data acquisition phase, it was realized that implementing a full parser for C++ or using already developed code for C++ parser to fulfill our particular requirements is quite a cumbersome, long and risky activity. Therefore, it was decided that basic data of a classes must be inserted into database tables manually. Following consists of what we are calling basic data of classes:

- Class *Alpha* <Derived From Class *Beta*>
    - Member Attribute
        - Attribute Type
        - Scope of Attribute (e.g. public, private and protected)
    - Member Method
        - Parameter List
        - Return Type
        - Scope of Method (e.g. public, private and protected)

Manual methods also have been used to verify that the correct data has been inserted into database from source code files.

Automated Method: In automated methods, we have implemented following type of applications to get the required data from the versions of source code.

- *Method implementation parser:* Method implementation parser is developed after the basic data of classes for three version of source code is inserted in to the database tables. Key responsibility of the method implementation parser is to read the member method implementation code from the CPP files into a buffer. After reading the implementation code, Method implementation parser is supposed to search and insert following information:

    - Member Attributes used by Member Method
        - Member Attribute of the class
        - Derived Attribute from the base class (e.g. public and protected attribute)
    - Member methods used by method
        - Used member methods (i.e. public, private and protected methods) of the same class
        - Used member methods (i.e. public and protected methods) from the base class
    - Types instantiated local variable in Member Method implementation
    - Type of Return variable by the Member Method

We have also used the manual inspection on the inserted data to verify that the correct information has been inserted into database tables through automated process.

- *Data Analyzer:* Inheritance found among the classes was making our metric calculation difficult and complex for class cohesion. Because, when a sub class is derived from a base class, it also derives the protected and public members of the base class. These derived members from a base class become the member of derived class implicitly. Addition of derived member makes our work difficult and complex. Therefore, we had to transform the data into a format that was easier for performing metrics calculation. For this purpose, Data analyzer application has been used. Data analyzer was also developed to perform the metrics calculation and insertion of calculated values into database tables.

5.2.2 Planned set of Metrics used for experiment

To perform a study on software evolution, class cohesion, class coupling, OO reuse and size metrics, the used set of metrics is as follows:

Class Cohesion: Our proposed metric measures the relative relatedness of public methods to overall public functionality of a class. The metric is given by following expression.

$$Cohesion(Class) = \frac{SRC}{\sum RCR} = \frac{\sum RC(SubsetTree)}{\sum RCR}$$

Refer back to chapter 3 of thesis to read the details about the above metric for class cohesion.

- *Class Coupling:* Set of metrics for the three forms of coupling namely, inheritance, association and dependency are used. All of them are studied with cohesion for all three versions of source code.

Inheritance Coupling		
Depth of class in inheritance tree	*DCI*	This metric calculates the depth at which a class exists in inheritance tree. For the root class in inheritance tree DCI count is 0. In case of multiple inheritances, DCI metric adds 1 to DCI count of all immediate parents and then adds these sums to its own DCI count.
Association Coupling		
Number of associated classes	*NAC*	This metric counts the number of associated classes as member attribute in a class via association relationship.
Total Associated Class Usages	*TACU*	This metric gives the number of times all associated attributes (of other class types) are used by methods of user class.
Dependency Coupling		
Number of used classes via dependency	*NUCD*	Number of distinct classes used by a class A by establishing dependency relationship.
Total number of usages of classes via dependency	*TNUCD*	Total number of times other classes are used by class A via dependency relationship.
Table 5-2		

- *Class Reuse:* Number of times class objects are used in the method both through the dependency coupling and association coupling. Usage of class object as return type, parameter in parameter list and local variable of method of a class are the examples class reuse through dependency. Usage of class object as a member variable in the methods of user class is an example of class reuse through association. For each instance of such reuse the metric count will be raised by one. In this experiment, by class reuse we always mean internal or private reuse- A measurement of the class reuse within one software system.

- *Class Size:* For an external user of class, the class size can be thought as size of functionality provided by a class. A class provides its functionality by means of public methods. Each public method of a class represents a unit of class functionality. Such units of functionality can be used to measure class size. We will use number of public methods (NPM) in a class for measuring the class size.

5.3 Technique for data analysis

We will be using the different techniques to analyze the resulted data from our experiment. We will show the data in tabular form and graphs. Pearson's Correlation Coefficient will also be used to analyze the linear relationship among subject metrics. Following reflects upon Pearson's Correlation Coefficient.

Pearson's Correlation Coefficient is measurement of strength of linear relationship between any two variables X and Y. Pearson's Correlation Coefficient '*r*' lies in the interval of -1 to 1 inclusively.

$$-1 \leq r \leq 1$$ Where r represent Pearson's Correlation Coefficient

Negative or positive sign of Coefficient indicates the direction of correlation between X and Y. A negative sign means an inverse relationship. In an inverse relationship one variable increases with decrease in other variable and vice-versa. On the other hand, the positive sign indicates the direct relationship between two variables, i.e. one variable increases and with increase in other variable and decreases with decrease in other variable. The coefficient of magnitude 1 means the perfect relationship between two variables, whereas the coefficient of value 0 indicates the absence of relationship between two variables [Richard 1999]. Researchers have ranked the type of correlation between two variables based on the magnitude of correlation coefficient.

According to [Cohen 1988], assuming a large data set, the correlation of magnitude 0.5 is large, the correlation of magnitude 0.3 is moderate and the correlation of magnitude 0.1 is small. Hopkins ranks the interval of correlation, according to him, correlation magnitude $r \leq |0.1|$ is trivial and it is as good as garbage, correlation magnitude 0.1-0.3 is minor, correlation magnitude 0.3-0.5 is moderate, correlation magnitude 0.5-0.7 is large, the correlation magnitude 0.7-0.9 is very large and correlation magnitude 0.9-1 is almost perfect [Hopkins 2003].

6 RESULTS AND DISCUSSION

Based on the methodology presented in the previous chapter, we have performed metrics calculation on the three versions of the source code of FileZilla system. Our experiment has been resulted into the statistical data against the set of metrics, we have described in previous chapters. In this chapter, we will present the faced limitations, results of the experiment and discussion on the results with the help of graphs and statistical techniques.

6.1 Limitations

During the experiment on OO source code, we have faced some limitations because of our selection of FileZilla system as an input for experiment. A brief description of these limitations is as follows:

- *Control Wrappers & Limited Access to Source Code:* Mattsson and Bosch have reported the problem of limited access to source code while discussing the Framework Composition problems [Mattsson and Bosch 2000]. We have also faced the problem of limited access to source code in relatively different context to Mattsson and Bosch. In our case, the set of metrics that was used for experiment takes source code as an input. A complete access to source-code of system provides an ideal environment for the execution of experiment. FileZilla application is developed using MFC framework for dialog based application with a number of MFC derived classes. Many of MFC derived classes are just the wrappers to ATL controls. Such wrapper classes call exposed functionality of dynamic link libraries which are actually implementing these controls. Due to the limited access, we had to ignore the effect of inheritance tree of MFC classes in order to solve this problem. Strategy to ignore inheritance tree of MFC class was used while calculating metrics determined for class cohesion and inheritance. In non-MFC classes experiment is performed as per normal procedure.

- *MFC Framework:* Studying evolution is difficult in a framework based system because, being framework, a major part of system remains constant from version to version and frameworks often do not give an absolute access to source-code. This decreases the opportunity to study the changes on all part of a subject system. Secondly, in MFC framework for a dialog based application, a major part of functionality is implemented inside the framework that leaves relatively little to do for a user of framework. Usually a user application of framework only implements specialized code in derived classes and most of instances it uses constant part of framework to implement new requirements. This also hinders to study the changes on a large scale.

  Known requirements: After going through FileZilla webpage, we have noticed that the types of changes, from version 2.1.6 to version 2.2.5, are bug fixes and addition of new requirements. Although, we have not got the feedback for our questions from the developer team yet, but we have got the impression from the release notes that most of new requirements from version 2.1.6 to version 2.2.5 were known at the start of system. Hence delayed design decisions become the part of architecture because the some of the future requirements were known at the start. Svahnberg and his colleagues discuss the software evolution is similar to software maintenance activities. He explains that adoptive and perfective maintenance activities, in form of addition and replacement of software requirements, affect software architecture at its root. This happens when architecture does not know about future requirements. On the other hand, the known requirements that are not implemented but supported by the software architecture are known as delayed design decisions. Implementation of such requirements causes the changes on leaf nodes of software architecture [Svahnberg el at. 2003]. This makes a system resistant to follow erosive trends from version to version. Software architecture may improve after a delayed design decision is implemented; we think we have also faced this situation in form of improved statistical results on instances. This phenomenon makes the verification of empirical knowledge difficult.

Based on the lesson learnt from the faced limitations, we suggest that for such experiments framework based systems are not the ideal choice. Instead of framework systems such experiment must be performed on systems that are built from scratch without the support of framework. Secondly, a better way to verify and validate the relationship between cohesion and coupling is to select a system in which the most of future requirements are unknown at the start.

6.2 Studying evolution with cohesion and coupling

To study evolution among the selected versions, we have divided resulted data of experiment into two sections. In first section we will study statistical changes from version 2.1.6 to version 2.1.8 and in second section we will study the statistical changes from version 2.1.8 to version 2.2.5. In both sections, we have only selected the classes with varying statistics with respect to our metrics for class cohesion and class couplings. Classes with constant statistics are filtered out. In each section varying values of cohesion and coupling metrics (association, dependency and inheritance) are shown in tabular format.

The expected results: Based on the literature review on software evolution are as follows:

- *Decreasing class cohesion:* High value of cohesion is always desirable because high value of cohesion is considered as a measure for the good design characteristics [Bieman & Kang 1995]. However, according to the law of software evolution [Lehman 1980] it is expected that with changes in design from version to version the architecture decays. This implies that cohesion of classes must follow undesirable trends (decreasing trend) during the evolution.
- *Increasing class coupling:* Contrary to cohesion, a lower value of coupling is desirable and it is expected that during the software evolution the value of coupling will follow increasing trend because of expected decay in architecture.

6.2.1 Changes from Version 2.1.6 to Version 2.1.8

Table 6-1 shows the changes of cohesion values from version 2.1.6 to version 2.1.8 against the classes.

It is evident from the table that most of changes in cohesion between two versions are not representing a big shift of values. We can see that instances of decreased and increased cohesions are equivalent to each other.

Class Name	Cohesion V 2.1.6	Cohesion V 2.1.8	Effect Type
CFtpControlSocket	0.1613	0.1641	Increased
CMainFrame	0.2536	0.2632	Increased
CQueueCtrl	0.2665	0.3156	Increased
CCommandQueue	0.4	0.4146	Increased
CAsyncSocketEx	0.404	0.362	Decreased
CAsyncProxySocketLayer	0.4056	0.3785	Decreased
CLocalFileListCtrl	0.4211	0.4118	Decreased
COptions	0.5	0.7108	Increased
CServerPath	0.5018	0.5	Decreased
CFtpListCtrl	0.5896	0.5688	Decreased
CSFtpControlSocket	0.7226	0.7157	Decreased
CFileZillaApi	0.7658	0.6864	Decreased
CStatusView	0.8334	1	Increased
CDirectoryCache	0.8928	0.9028	Increased
Summary			
Decreased	7		
Increased	7		
Table 6-1			

Table 6-2 shows the changes in association coupling (NAC) from version 2.1.6 to version 2.1.8 in the classes. Differences in the values of NAC metric are not very high but the most of the instances of changes are following increasing trend according to the expected results.

Class Name	NAC V 2.1.6	NAC V 2.1.8	Effect/Type
CQueueCtrl	0	1	Increased
CFtpListCtrl	2	1	Decreased
CIdentServerControl	5	4	Decreased
CTransferSocket	7	8	Increased
CControlSocket	7	8	Increased
CSFtpControlSocket	8	9	Increased
CFtpControlSocket	8	9	Increased
Summary			
Decreased	2		
Increased	5		
Table 6-2			

Table 6-3 shows the changes in dependency coupling (TNUCD) from version 2.1.6 to version 2.1.8 in the classes. Differences in the values of TNUCD metric are not very high but on the most of the instances of changes are following increasing trend according to the expected results.

Class Name	TNUCD V 2.1.6	TNUCD V 2.1.8	Effect Type
CStatusView	1	2	Increased
COptions	4	18	Increased
CSiteManager	21	23	Increased
CFileZillaApi	19	20	Increased
CSFtpControlSocket	40	39	Decreased
CMainFrame	101	98	Decreased
Summary			
Decreased	2		
Increased	4		
Table 6-3			

During our study on finding change in DCI metric from version 2.1.6 to 2.1.8, we have seen insignificant changes in DCI values between the versions. That is why; we will not discuss DCI changes from version 2.1.6 to 2.1.8.

6.2.2 Changes from Version 2.1.8 to Version 2.2.5

Table 6-4 shows the changes of cohesion values from version 2.1.8 to version 2.2.5 against the classes.

It is evident from the table that the most of changes in class cohesion between two versions are showing relatively bigger shift in values as compared to previous section. We can find more instances where the cohesion values of classes have decreased. Instances where the values of class cohesion have increased are less in number and showing very a little shift of value.

Class Name	Cohesion V 2.1.8	Cohesion V 2.2.5	Effect Type
CMainThread	0.1519	0.1765	Increased
CFtpControlSocket	0.1641	0.2544	Increased
CTransferSocket	0.2222	0.2045	Decreased
CMainFrame	0.2632	0.2922	Increased
CQueueCtrl	0.3156	0.3255	Increased
CAsyncSocketEx	0.362	0.3542	Decreased
CLocalFileListCtrl	0.4118	0.303	Decreased
COptions	0.7108	0.6561	Decreased
CServerPath	0.5	0.3902	Decreased
CAsyncSslSocketLayer	0.52	0.4857	Decreased
CFtpListCtrl	0.5688	0.4929	Decreased
CDirTreeCtrl	0.8095	0.4584	Decreased
CStatusView	1	0.8667	Decreased
CSplitterWndEx	0.8928	0.8852	Decreased
CFtpTreeCtrl	1	0.5	Decreased
Summary			
Decreased	11		
Increased	4		
Table 6-4			

Table 6-5 shows the changes in association coupling (NAC) from version 2.1.8 to version 2.2.5 in the classes. Differences in the values of NAC metric, between two versions, are not very high and that is why, these can be ignored.

Class Name	NAC V 2.1.8	NAC V 2.2.5	Effect/Type
CComboCompletion	1	0	Decreased
CFtpListCtrl	1	2	Increased
CIdentServerControl	4	5	Increased
Summary			
Decreased	1		
Increased	2		
Table 6-5			

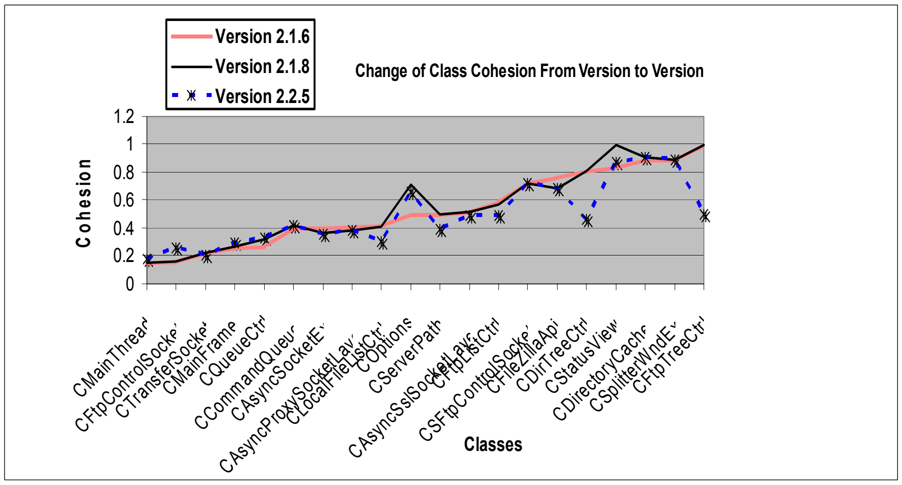
Table 6-6 shows the changes in dependency coupling (TNUCD) from version 2.1.8 to version 2.2.5 in the classes. Differences in the values of TNUCD metric are relatively higher than previous section, however, most of the instance of changes are following increasing trend akin to previous section.

Class Name	TNUCD V 2.1.8	TNUCD V 2.2.5	Effect Type
COptionsConnection2	1	0	Decreased
CComboCompletion	6	1	Decreased
COptions	18	11	Decreased
CFtpTreeCtrl	6	14	Increased
CSiteManager	23	24	Increased
CDirTreeCtrl	6	14	Increased
CFileZillaApi	20	21	Increased
CLocalFileListCtrl	22	34	Increased
CFtpListCtrl	40	49	Increased
CFtpControlSocket	49	50	Increased
CMainFrame	98	110	Increased
Summary			
Decreased	3		
Increased	8		
Table 6-6			

During our study on finding change in DCI metric from version 2.1.8 to version 2.2.5, we have not seen changes in DCI values between the versions. That is why; we can not discuss DCI changes from version 2.1.8 to version 2.2.5.

6.2.3 Class cohesion VS versions

Graph 6-1 has been plotted between class cohesion and three versions of FileZilla system. Classes that exist in all three versions have been selected and are shown on x-axis and their corresponding class cohesion is plotted on y-axis.



Graph 6-1

Lines representing three versions are bit vibrating across each other and are not following a common trend. However, on the most of instances version 2.2.5 is representing lower cohesion as compared to other two versions. On the most of the instances versions 2.1.6 and 2.1.8 are inseparable because of very small differences of functionality in both versions.

In section 6.2 and in its subsections, we have attempted to study software evolution with the help of class cohesion and class coupling metrics while keeping the expected results in mind. During this study, we have seen the existence of both expected and unexpected results. Although, we have not observed a large number of classes; yet, we have identified possible trends in cohesion and coupling during the evolution. In our opinion, expected results related to cohesion and coupling are not essentially always true. Rather such results have a more probability to be true than unexpected results. Because, cohesion and coupling are not just two important factors affecting software architecture, there may be many other influencing factors (e.g. delayed design decision, usage of framework, etc.,) besides cohesion and coupling. Such influencing factors also play an affective role to increase or decrease the erosive trends in software architecture during the evolution.

6.3 Cohesion VS Coupling

To study cohesion and coupling mutually and finding possible relationships between them, are important goals of our experiment. To analyze possible relationship, we have plotted the graphs and calculated the correlation coefficient between cohesion and coupling metric (i.e. DCI, TAC, and TNUCD) for all three versions.

6.3.1 Cohesion VS inheritance coupling

To identify the relationship between class cohesion and inheritance coupling DCI we have calculated the correlation coefficient between class cohesion and DCI metrics. Only non-MFC derived classes were used to calculate the cohesion from three versions. Calculated coefficient values are as follows in table 6-7a:

Version	Correlation Coefficient Between Cohesion and DCI
2.1.6	-0.5272
2.1.8	-0.5246
2.2.5	-0.4878

Table 6-7a

We have found considerable strength of negative correlation between class cohesion and DCI metrics. Correlation values remain between -0.48 and -0.52 and indicate a significant inverse relationship among the class cohesion and DCI metric. Based on the results, we can argue that it is more probable to observe inverse relationship between class cohesion and inheritance coupling.

6.3.1.1 Trends in inheritance tree

Dirk and his colleagues have emphasized on the need of studying OO metrics from inheritance perspective in their paper "Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems" [Dirk et al 2000]. We have also tried to study the relationships among OO metrics with and without considering inheritance tree. We have noticed that the some of the relationships among OO metrics become more visible when studied under the inheritance tree.

Due to the limited access to MFC source code as mentioned earlier, we have filtered out inheritance tree of MFC derived classes. We have found only one considerable inheritance tree among the non-MFC derived classes. In the found inheritance tree, metrics calculations were performed using the inheritance, class cohesion, size and reuse metrics. Table 6-7b shows the relationships among the pairs of metrics using correlation coefficient. Correlation coefficient between two metrics can be read by picking a non-empty cell where one metric in row crosses the other metric in column.

Correlation Coefficient among Cohesion, Size(NPM) and Reuse in Inheritance Tree from three Versions				
Version 2.1.6				
	DCI	***Cohesion***	***NPM***	***Reuse***
DCI	1			
Cohesion	-0.2286	1		
NPM	0.9408	-0.21164	1	
Reuse	-0.8032	0.350388	-0.67652	1
Version 2.1.8				
	DCI	***Cohesion***	***NPM***	***Reuse***
DCI	1			
Cohesion	-0.2778	1		
NPM	0.9027	-0.22165	1	
Reuse	-0.8825	0.296743	-0.63214	1
Version 2.2.5				
	DCI	***Cohesion***	***NPM***	***Reuse***
DCI	1			
Cohesion	-0.2286	1		
NPM	0.9029	-0.16577	1	
Reuse	-0.8875	0.266202	-0.64176	1
Table 6-7b				

Table 6-7b shows that correlation between class cohesion and DCI (Depth of class in Inheritance tree) is low and can not be used to derive a result from it. Contrarily, we have found a considerable inverse correlation between class cohesion DCI metric when not considering inheritance tree as shown in Table 6-7a.

Table 6-7b shows a nearly perfect relationship between class size (NPM) and DCI metrics with correlation of more than 0.9. Size metric NPM is increasing in downward direction in the inheritance tree, due to the inherited public methods from the base classes.

Correlation between class size (NPM) and class reuse is between -0.63 and -0.67 that implies class reuse is increasing in the upward direction in the tree; whereas class size is decreasing in the same direction. One of the possible explanations could be the high generalized nature of a class at higher level, which makes a class an obvious choice for reuse regardless of its small size among other classes in hierarchy.

We will also discuss the OO Size and Reuse metrics in sections 6.3.4 and 6.3.5 of this chapter.

6.3.2 Cohesion VS association coupling

To study the relationship between cohesion and association coupling, we have performed the metrics calculations using Class cohesion, NAC (Number of associated classes) and TACU (Total Associated Class Usages) metrics. To analyze the resulted data we have calculated the correlation among the mentioned metrics.

Table 6-8, shows the value of correlation coefficient among the metrics for Class Cohesion, NAC and TACU from the three versions. Correlation coefficient between two metrics can be read by picking a non-empty cell where one metric in row crosses the other metric in column.

Correlation Coefficient among Cohesion, TACU and NAC from three Versions			
Version 2.1.6			
	Cohesion	***NAC***	***TACU***
Cohesion	1		
NAC	-0.35391	1	
TACU	-0.36597	0.413652	1
Version 2.1.8			
	Cohesion	***NAC***	***TACU***
Cohesion	1		
NAC	-0.15394	1	
TACU	-0.35363	0.517194	1
Version 2.2.5			
	Cohesion	***NAC***	***TACU***
Cohesion	1		
NAC	-0.14927	1	
TACU	-0.32292	0.482849	1

Table 6-8

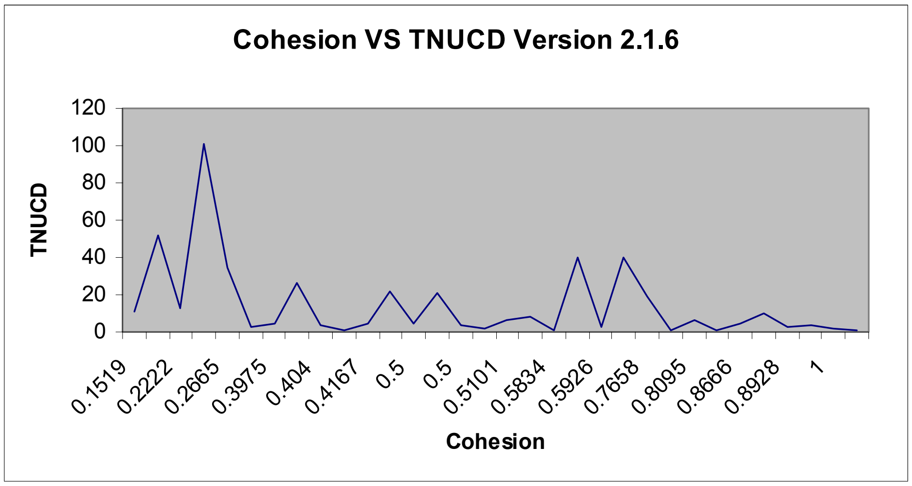
Class cohesion and NAC: From the table we can see that only in version 2.1.6 the correlation between class cohesion and NAC is above 0.3 with negative sign. In the remaining two versions 2.1.8 and 2.2.5 the correlation is nearly 0.1 that can be ignored. Correlation values from three versions, suggests no obvious relationship can be traced out between class cohesion and NAC.

Class Cohesion and TACU: TACU metric also represents association coupling by counting the number of times a class uses its associated attributes. Magnitude of correlation between class cohesion and TACU metric for class is slightly above 0.3 with a negative sign for all three versions. Resulted values of correlation coefficient suggest the possibility of inverse relationship between class cohesion and association coupling.

6.3.3 Cohesion VS dependency coupling

To study the relationship between cohesion and dependency coupling, we have plotted the graphs between class cohesion and TNUCD metrics for the three versions of source code and we have also calculated the correlation coefficient between class cohesion and TNUCD metric.

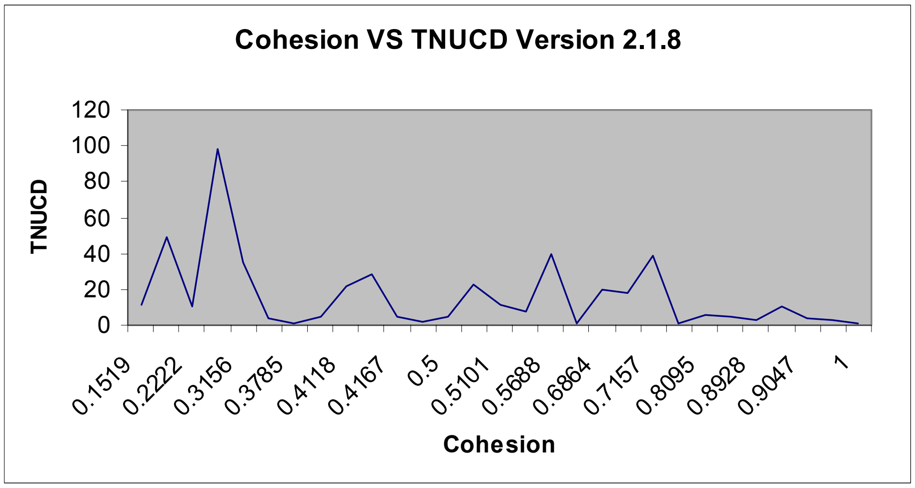
Graph 6-2 has been plotted between class cohesion and coupling metric TNUCD for version 2.1.6.



Graph 6-2

For the version 2.1.6, correlation coefficient between class cohesion and TNUCD is -0.3883.

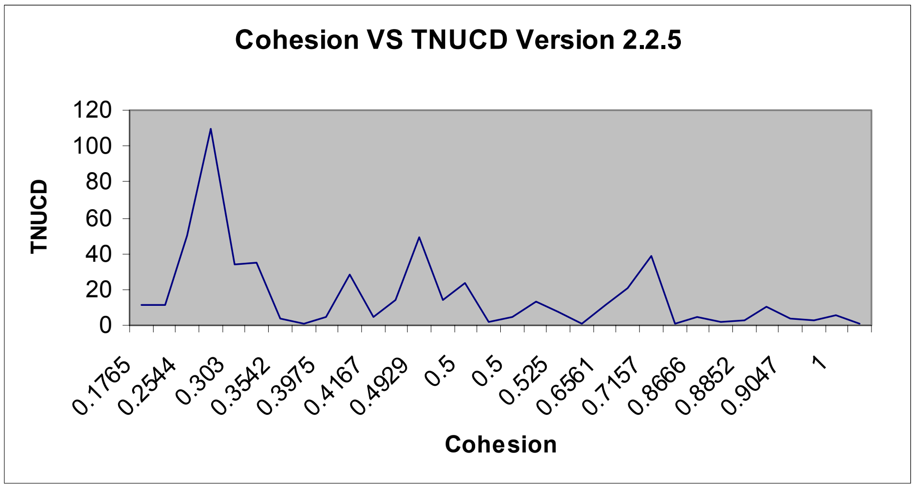
Graph 6-3 has been plotted between class cohesion and coupling metrics TNUCD for version 2.1.8.



Graph 6-3

For the version 2.1.8, correlation coefficient between class cohesion and TNUCD is -0.3856.

Graph 6-4 has been plotted between class cohesion and coupling metrics TNUCD for version 2.2.5.



Graph 6-4

For the version 2.2.5, correlation coefficient between class cohesion and TNUCD is -0.43513.

The value of correlation coefficient between class cohesion and TNUCD metric remains in the interval of -0.3856 to -0.4351 for three versions. According to Hopkins this is moderate correlation. Based on results from correlation, we argue that occurrence of inverse relationship between class cohesion and dependency coupling is probable but not certain.

In section 6.3 and all of its subsections, we have studied relationships between class cohesion and class coupling metrics with the help of correlation coefficients. Table 6-9 summarizes the results in form of correlation coefficients for three versions. Table 6-9 shows correlation value between three types of coupling (inheritance coupling, association coupling and dependency coupling) and class cohesion.

Version	**Class Cohesion Vs Inheritance Coupling Correlation**	**Class Cohesion Vs Association Coupling Correlation**	**Class Cohesion Vs Dependency Coupling Correlation**
2.1.6	-0.5272	-0.3659	-0.3883
2.1.8	-0.5246	-0.3536	-0.3856
2.2.5	-0.4878	-0.3229	-0.4351

Table 6-9

From the table 6-9, we can observe that all types of coupling have inverse relationships with class cohesion. However, none of correlation values depict a perfect linearity among cohesion and types of coupling. Relationship between class cohesion and inheritance coupling is stronger than others two forms of coupling. Correlation values for association coupling and dependency coupling metrics with class cohesion are also showing inverse relationships. Based on the collective results from table 6-9, we argue that it is more probable to observe an inverse relationship between class cohesion and class coupling metrics.

6.3.4 Cohesion and size

To find out connections between class cohesion and class size, we have performed the metrics calculation using class cohesion and class size metrics. Number of public methods (NPM) is used as measurement of class size. Values of correlation coefficient between two metrics are shown for three versions in table 6-10.

Version	Correlation Coefficient between Class Cohesion and Class Size (NPM)
2.1.6	-0.3862
2.1.8	-0.3888
2.2.5	-0.3503

Table 6-10

Table 6-10 shows that values of correlation for three versions, which remain between 0.35 and 0.38 with the negative sign. Such correlation values are ranked as moderate correlation. Negative coefficient values indicate the probability of an inverse relationship between class size and class cohesion. Because of the moderate correlation value, an inverse relationship between class cohesion and class size is possible, but not guaranteed.

If we look into class size and class cohesion metrics, both of them are related to the public functionality of a class; therefore, it was expected to observe a stronger relationship between class size and class cohesion. But resultant moderate correlation between two metrics implies that size of class is not only determinant of class cohesion, other factors may also affect class cohesion.

6.3.5 Studying OO reuse

For measuring OO reuse, we have focused on private reuse- reuse within one software system [Bieman & Kang 1995]. To measure class reuse, we have calculated four forms of the class usages. 1) Usage of the class object as a parameter in the methods of other classes 2) Usage of the class object as a local instance in the methods of other classes 3) Usage of the class object as a return parameter in the methods of other classes and 4) Number of times usage of class via member variable in other classes of system. On any instance of class usage, the reuse metric is increased by 1.

Table 6-11 shows the value of correlation coefficient between class cohesion and class reuse in three versions of source code.

Version	Correlation Coefficient between Class Cohesion and Reuse
2.1.6	-0.1177
2.1.8	-0.0876
2.2.5	-0.1343

Table 6-11

Table 6-11 is showing negative correlation with value of between -0.08 and -0.1343 that is very negligible; hence we are unable to present any view point on the relationship between class cohesion and class reuse.

We have also studied the relationship between class reuse and class size. Prior to this study, we were expecting to find a stronger correlation between class size and class reuse. But, from the results, shown in table 6-12, we have not noticed considerable relationship between class size and class reuse.

Version	Correlation Coefficient between Class Size (NPM) and Class Reuse
2.1.6	-0.0333
2.1.8	-0.0806
2.2.5	-0.1101

Table 6-12

As shown by the table the valued of correlation remain between -0.1101 to -0.0333. Resultant values are too small to discuss for finding any relationship between class reuse and class size. Therefore, we are unable to see any relationship between class size and class reuse. However, if we refer back to the results of table 6-7b that shows the study of class size and reuse in inheritance tree. We can see that negative correlation between reuse and size metrics in the interval of -0.6321 and -0.6765 for the three versions.

7 FUTURE WORK

This chapter presents some of our recommendations and ideas for the similar type of experiments based on the results and the pitfalls we have faced during our experiment. Our suggestions for the future work for the similar type of the experiments are as follows:

- *Framework independence:* The best way to perform such experiments is by the selection of the system that is developed without any framework. Framework independence will result into large a set of data points. Large set of data points will enable the researchers to come up with more obvious and clear results.

- *Measuring coupling on ordinal scale:* To define an ordinal scale for two dimensions of coupling namely, association, and dependency isolated design experiment could be an option. Suppose we have certain software requirements, we are implementing them using OO programming while keeping following constraints in mind:

    - Designing without dependency relationship
      In the design dependency relationships will not be used among the classes. Instead of dependency relationships association relationships will be used.
    - Designing without association relationship
      In the design association relationships will not be used among the classes. Instead of association relationships dependency relationships will be used.

  After the completion of both implementations, a complete set of metrics calculation will be performed on both. And results will be analyzed using the cohesion, size, reuse and other OO metrics. We think, a comparison of metrics results from both implementations may help us in defining an ordinal scale between dependency and association coupling.

- *Cohesion of data structures:* An experiment should be performed using our cohesion and coupling metrics on data structure classes (i.e. classes that implement the data structure) e.g. *Link List, Queue, Stack, Heap, Trees, Hash tables etc.* Data structure classes are the most reused classes. Study of data structure classes may help us in testing some of the empirical statements pertaining to cohesion and coupling. Such as, high cohesion and low coupling is considered as a good design characteristic. A study on a large number of data structure classes, from different implementations, may be helpful in justifying mentioned empirical statement.

- *Considering inheritance tree:* Although, we were only able to study one inheritance tree of non-MFC derived classes, because of limited access to MFC source code. Still, we recognize the fact that the results from the implementation metrics become more visible and apparent when studied in inheritance tree. Therefore, we emphasizes on the need of further research on inheritance trees using our set of metrics.

8 CONCLUSION

The reviewed metrics, for class cohesion, do not capture all aspects of class cohesion. The metrics, based on LCOM, only measure the absence of cohesion rather than presence of cohesion. TCC (Tight Class Cohesion) and LCC (Loose Class Cohesion) metrics are not suitable for measuring the cohesion of a class with large number of public methods or in case of all non-connected public methods. Our proposed metric, that measures class cohesion on the basis of relative relatedness of public methods to the overall public functionality of class, is resulted better than the metrics that only measure the class cohesion on the basis of relatedness among the public methods.

Approach to measure class coupling by counting the number of instances where a class is dependent on other classes via UML relationships (Association, Inheritance and Dependency), can enable us to analyze the class coupling more granularly.

Occurrence of Software erosion from version to version is an event more likely to be observed using class cohesion and class coupling metrics. The delayed design decisions, as a result of planned future requirements, may slow down or stop the phenomena of Software erosion.

It is more likely to be observed to find the inverse relationships between class cohesion and three types of class couplings (i.e. Association, Inheritance and Dependency). Nevertheless, occurrence of such inverse relationships between class cohesion and class coupling metrics can not be guaranteed always.

Finding the inverse relationship between class cohesion and class size is likely to be observed but not guaranteed.

As per results from experiment, no relationship between class cohesion and class reuse is found. Contrarily, the existence of a significant inverse relationship between class size and class reuse was found when they were studied in inheritance tree.

Usage of OO Framework based system reduces the opportunity to observe many classes, due to limited access to source code of framework, some classes are filtered out. Therefore, selection of framework independent system for performing similar kind of studies can be more useful than the framework dependent system.

9 REFERENCES

[Bansiya 2002]
J. Bansiya, “A Hierarchical Model for object- oriented Design Quality Assessment” IEEE Transaction on software engineering, Vol.28, No.1, January 2002.

[Bieman & Ott 1994]
J. Bieman and L. Ott, “Measuring functional cohesion”. IEEE Trans. Software Engineering, 20(8):644– 657, Aug. 1994.

[Bieman & Kang 1995]
James M. Bieman and Byung-Kyooh Kang “Cohesion and reuse in an object-oriented system”, ACM Press New York, NY, USA, Pages: 259 – 262, 1995.

[Booch et al. 1999]
Booch, G., Rumbaugh, J., Jacobson, I., “The Unified Modeling Language User Guide”, Addison- Wesley, Rational Software Corporation, 1999.

[Briand et al. 1997]
Briand, L., Devanbu, P., Melo, W.: “An investigation into coupling measures for C++”, Proceedings of ICSE 1997, Boston, USA, 1997.

[Briand et al. 2000]
L.C. Briand, J. Wust, J.W. Daly and D.V. Porter, “Exploring the relationships between design measures and software quality in object-oriented systems”, Journal of Systems and Software, page: 245–273, 2000.

[Chidamber & Kemerer 1991]
S.R. Chidamber, C.F. Kemerer, Towards a metrics suite for object-oriented design, Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA ’91), 1991.

[Chidamber & Kemerer 1994]
S. Chidamber and C. Kemerer, “A metrics suite for OO design”, IEEE Trans. Software Eng., 20 (1994) 476–493.

[Cohen 1988]
J. Cohen, Statistical Power Analysis for the Behavioral Sciences, second ed., Lawrence Erlbaum, Mahwah, NJ, 1988.

[Counsell et al. 2002]
Steve Counsell, Emilia Mendes and Stephen Swift, “Comprehension of object-oriented Software Cohesion: the empirical quagmire”, Proceedings of the 10th International Workshop on the Program Comprehension (IWPC, 02) 2002.

[Dirk et al. 2000]
Dirk Beyer, Claus Lewerentz, and Frank Simon, “Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems”, IWSM 2000.

[Etzkorn et al. 2004]

Etzkorn, Letha H., Sampson E. Gholston, Julie L. Fortune, Cara E. Stein, Dawn Utley, “A Comparison of cohesion metrics for object-oriented systems”, Information and Software Technology, page:667-687, 2004.

[Fenton & Pfleeger 1998]
Norman E. Fenton, Shari Lawrence Pfleeger, “Software Metrics: A Rigorous and Practical Approach”, PWS Publishing Company, 1998.

[Gurp & Bosch 2002]
J. van Gurp, J. Bosch, Design Erosion: Problems & Causes, Journal of the Systems & Software, 61(2), pp. 105-119, Elsevier, March 2002.

[Harrison at el. 1998]
Harrison R., Counsell S., Nithi R., “Coupling metrics for object-oriented design”, Journal: Software Metrics Symposium, Metrics 1998. Fifth International pages*:* 150-157, 1998.

[Henderson at el. 1996]
Henderson Sellers, B., Constantine, L.L. and Graham, I.M., Coupling and Cohesion (towards a valid metrics suite for object-oriented analysis and design, Object-Oriented Systems, 3, pp. 143-158, 1996.

[Henderson 1996]
B. Henderson Sellers, Software Metrics, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[Hitz & Montazeri 1996]
M. Hitz, B. Montazeri, Chidamber and Kemerer’s metrics suite: a measurement theory perspective, IEEE Transactions on Software Engineering 22, pages: 267–271, 1996.

[Hopkins 2003]
W.G. Hopkins, “A New View of Statistics”, SportScience, Dunedin, New Zealand, http://www.sportsci.org/resource/stats, 2003.

[Parnas 1994]
D.L. Parnas, Software Aging, in Proceeding of the Sixteenth International conference on Software Engineering, IEEE Computer Society Press, 1994.

[Mattsson & Bosch 2000]
M. Mattsson & J. Bosch, “Framework Composition: Problems, Causes and Solutions, University of Karlskrona/Ronneby, Department of Computer Science”, In “Building frameworks: Object-Oriented Foundations of Framework Design, M.E. Fayad, D.C. Schmidt, R.E- John Wiley & Sons, New York NY, pp.958-969, 1994.

[Richard Lowry 1999]
Richard Lowry, “Concepts and Application of Inferential Statistics”, http://faculty.vassar.edu/lowry/webtext.html , 1999.

[Svahnberg el at. 2003]
Mikael Svahnberg, Claes Wohlin, Lars Lundberg and Michael Mattsson, A Quality-Driven Decision-Support Method for Software Architecture Candidates, International Journal of Software Engineering, 2003.

[Troy and Zweben, 1981]
D.A. Troy and S.H. Zweben, "Measuring the Quality of Structured Designs," Journal of Systems and Software, Vol. 2, No. 2, June 1981, pp. 113 - 120.

[Yourdon and Constantine 1979]
Yourdon, E and Constantine, L.L., Structured design, Prentice Hall, Englewood Cliffs, NJ, 1979.

10 APPENDIX

10.1 Appendix A

10.1.1 Used Terminology

Following describes the terminology which is extensively used in the thesis.

Terms	**Definitions**
Member	The term member means the member of a class, such as member attributes and member methods of the class.
Attribute	The term attribute means the member variables of the class. Alternatively, the term attribute is also referred as properties of the class.
Method	The term method stands for the member procedures of the class that a class implements. This term is also referred as function in some of the programming languages such as C++.
Base class	The term base class means the class from whom a particular class is derived via inheritance relationship. A Base class is also referred as super class and parent class.
Derived class	The term derived class means the class that is derived from a base class. Derived class is also referred as subclass or child class.
Dependency relationship	Dependency relationship between two classes A and B is established when a class A uses the instance of other class B as a local variable or parameter in parameter list or the return type parameter in a member function of class A.
Association relationship	Association relationship between two classes A and B is established when A is composed class B as member attribute of class A. Aggregation and composition are also the kind of association relationships.
Inheritance relationship	This type of relationship exists among the general kind of classes and their more specific kind of classes. In this way, the relationship links generalized class with specialized class. By a specialized class we mean derived class which is also referred as subclass or child class. And, by generalized class we mean base class or parent of the derived class. Inheritance relationship is also referred as generalization relationship.
Group	A group stands for a set of public methods of the class which are using a common member attribute of the class.
Subset Tree	A subset tree is composed of nodes representing sets with public methods as their elements. Such sets group the public methods using a same attribute. Identified sets are placed in relationship of subset to parent nodes in the hierarchy of subset tree. Using a subset tree helps in determining relative contribution of method to overall public functionality of the class.

Appendix Table 1

10.1.2 Used Metrics

Metric Type	***Metric***	***Abbr.***	***Metric Definition***
Cohesion Metrics	*Weight of Group G*	WG	Weight of a group can be determined by divining the total number of methods in a group by the total number methods in a subset tree.
	Sum of Weight for a method	S(M)	Sum of weight for method M is calculated by adding the respective weight of groups on which a method M is occurring in a subset tree.
	Relatedness of Method	R(M)	Relatedness of method with subset tree can be determined for method M by dividing the S (M) of M by maximum value of S (M) determined for any method.
	Cohesion of Subset Tree	Cohesion (Subset Tree)	Cohesion of subset tree can be determined by dividing the sum of R (M) of all methods in a subset tree by total number of methods in the tree.
	Weight of Subset Tree	WST	Weight of subset tree is determined by dividing the total number of methods in a subset tree by the total number of public methods in a class.
	Relative Cohesion	RC	Relative Cohesion of subset tree can be calculated by multiplying the WST of subset tree with Cohesion of subset tree.
	Relative Cohesion Ratio	RCR	Relative Cohesion Ratio is calculated by dividing the RC of subset tree by maximum value of RC calculated among all found subset trees in a class.
	Sum of Relative Cohesion	SRC	Sum of relative cohesion is calculated by adding RC of all subset trees in a class.
	Cohesion of Class	Cohesion (Class)	Cohesion of the class can be calculated by dividing the SRC of class by the sum of RCR of the class.
Coupling Metrics	*Number of used classes by dependency relation*	NUCD	Number of distinct classes used by a class A by establishing dependency.
	Total number of evidences for 'Used classes by dependency relation'	TNUCD	Total number of times other classes are used by class A via dependency relationship.
	Ratio of NUCD to TNUCD (RNUCD	RNUCD	This measures the ratio between NUCD and TNUCD.
	Number of user classes for a class through dependency relation	NUCC	Number of distinct classes using a class A by establishing dependency.
	Total number of evidences for 'User classes through dependency relation'	TNUCC	Number of times, classes using a class A by establishing dependency.
	Ratio of NUCC to TNUCC (RNUCC):	RNUCC	This measures the ratio from NUCC to TNUCC.
	Depth of class in inheritance tree	CDI	This metric calculates the depth at which a class exists in inheritance tree. For the root class in inheritance tree DCI count is 0. In case of multiple inheritances, DCI metric adds 1 to DCI count of all immediate parents and then adds these sums to its own DCI count.

	Number of derived classes from a class	NDC	This metrics calculates the number of derived classes from a class by counting all of its child classes in inheritance tree.
	Number of associated classes with a class	NAC	This metric counts the number of associated classes as member attribute in a class via association relationship.
	Total associated class Usages (TACU)	TACU	This metric gives the number of times all associated attributes (of other class types) are used by methods of user class.
OO Size	*Class Size*	NPM	Class size is measure of number public method in a class excluding the constructor, destructor and operator overloading methods.
OO Reuse	*Class Reuse*	Class Reuse	Total number of times the functionality of class is invoked via instances of a class by the other classes.

Appendix Table 2

10.2 Appendix B

10.2.1 Results of Metrics Calculations

The results of metrics calculation in tabular form can be accessed by using following link:

http://www.student.bth.se/~imba03/msthesis/results.htm

Database file that contains all data produced during the experiment on three of versions of FileZilla system can be downloaded from following link:

http://www.student.bth.se/~imba03/msthesis/experiment.zip