CHAPTER 2

LITERATURE REVIEW

2.1 CLASSICAL METRICS FOR COMPLEXITY

In the field of software metrics there has been research on metrics to predict fault-proneness, change-proneness, identifying refactorable items, etc. Stevens introduced the concept of coupling into structured design, defined coupling to be "the measure of the strength of association established by a connection from one module to another." Highly coupled classes are considered bad design and low coupled components are advocated. As the degree of coupling increases so does the complexity of the class. The module becomes dependent on external classes to implement its functionality and is bound to reflect any changes the external classes may undergo in future maintenance. It is ideal to modify a class without having to take into consideration other modules or components within a system.

2.1.1 Halstead Measures

Halstead complexity measures are software metrics introduced by Maurice Howard Halstead as part of his treatise on establishing an empirical science of software development. Halstead makes the observation that metrics of the software should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. These metrics are therefore computed statically from the code. Halstead's goal was to identify measurable properties of software, and the relations between them. Thus these metrics are actually not just complexity metrics.
For a given problem, let: First we need to compute the following numbers, given the program:

- \( \eta_1 \) = the number of distinct operators
- \( \eta_2 \) = the number of distinct operands
- \( N_1 \) = the total number of operators
- \( N_2 \) = the total number of operands

From these numbers, several measures can be calculated:

- Program vocabulary: \( \eta = \eta_1 + \eta_2 \)
- Program length: \( N = N_1 + N_2 \)
- Calculated program length: \( \hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \)
- Volume: \( V = N \times \log_2 \eta \)
- Difficulty: \( D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2} \)
- Effort: \( E = D \times V \)

The difficulty measure is related to the difficulty of the program to write or understand, while doing code review. The effort measure translates into actual coding time using the following relation,

- Time required to program: \( T = \frac{E}{18} \) seconds

Halstead's delivered bug (B) is an estimate for the number of errors in the implementation.

- Number of delivered bugs: \( B = \frac{E^2}{3000} \) or, more recently, \( B = \frac{V}{3000} \) is accepted.
2.1.2 Cyclomatic Complexity Metric

Definition

Thomas McCabe defined Cyclomatic Complexity (CC) to measure the structural complexity of a procedure. McCabe's metric finds its foundation within graph theory. The metric transforms a procedure's statements into a graph. Each node within the graph represents a different conditional statement. CC can therefore be represented as a function:

\[ M = E - N + 2P \]

where \( E \) is the number of edges, \( N \) the number of nodes, and \( P \) the number of components within the graph. An alternative way to define the metric is to take the summation of the total number of decision points plus one. From a body of statements one can construct a control flow graph that represents the individual decision points within a procedure. The number of nodes within this control flow graph plus one is a popular alternative way to define CC.

Extended Cyclomatic Complexity (ECC)

This variation adds one additional factor, Boolean operators. These can be used within an if guard to increase the complexity of the decision. However, these Boolean operators can be re-factored out and turned into their own if statements. Therefore, this variation includes the complexity that Boolean operators add if they were if constructs.

Modified Cyclomatic Complexity (MCC)

In C based languages, switch statements are equivalent to a series of if /else if constructs. In the case of a switch the guard is evaluated each
time and then directs execution towards the correct case handler. This can be argued to be a single decision point instead of the alternative if / else if.

**Transitive Cyclomatic Complexity (TCC)**

CC computes a complexity value over a procedure. By McCabe's definition the number of connected components (method calls) or Pis merely multiplied by two. However, Transitive Cyclomatic Complexity attempts to further this value. Instead of only using $2P$, TCC will inject the summation of all Cyclomatic Complexities computed on all methods that can possibly be executed on the static types invoked. Therefore,

Class $C_1$ has methods $m_1, \ldots, m_n$ that are defined in the class:

$\text{McCabe}(m_i) = E - N + 2P$

$\text{CC}(m_i) = \text{Decisions} + 1$

$\text{TCC}(C) = \sum_{i=1}^{n} \text{CC}(\text{T(SIM}(m_i))))$

$\text{TWCC}(C) = \sum_{i=1}^{n} \text{McCabe}(\text{T(SIM}(m_i))))$

where, $E$ is the number of edges and $N$ is the number of nodes in the control flow graph of a method $m$. The plus one is used for when the CC of a method that has no decision points still maintains some complexity or more specifically a value of one.

**2.1.3 Information Flow Metric**

Lee et al (1995) measure coupling and cohesion of an object-oriented program based on information flow through programs. They define a measure, Information-flow-based coupling (ICP), that counts for a method
m of a class, and the number of methods that are invoked polymorphically from other classes, weighted by the number of parameters of the invoked method. This count can be scaled up to classes and subsystems. They go on to derive two more sets of measures which measure inheritance-based coupling (coupling to ancestor classes (IH-ICP)) and non inheritance-based coupling (coupling to unrelated classes (NIH-ICP)) and deduce that ICP is simply the sum of IH-ICP and NIH-ICP.

2.1.4 Slice based Coupling Metric

Mark Weiser introduced the concept of program slicing, taking the way that programmers debug their code as inspiration. The idea was to remove any lines of code not affecting the values computed at the point under consideration, as long as the removal did not impede the running of the program. Slicing algorithms act on intermediate representations of the program. Automatic slicing tools parse the program code to create one or more of these representations, such as control flow graphs (CFG), abstract syntax trees (AST) or system dependence graph (SDG). Weiser sliced using data flow analysis; modern tools generally trace the transitive control and data dependences by traversal of the SDG.

Harman et al (2001) introduces a slice is taken in respect of a variable or set of variables at a point in the program. The simplest form of slicing is static slicing; this includes all statements affecting or affected by the variable(s) at the point of interest, for all possible inputs to the program. There are other types of slicing which aim to reduce the size of slices; for example, dynamic slicing considers a particular set of input values, which may reduce the slice, as branches of the program not executed when using these inputs will be excluded; amorphous slicing reduces the slice by the use of transformations to the program. For each of these, in simple terms, backward slicing will result in the set of program statements, or points, which
affect the calculation of the value of the variable at the point of interest. Forward slicing finds the set of program statements, or points, which will be affected by computation at the point of interest.

Program slices offer a basis for measuring module cohesion. The slices used in calculating slice-based cohesion metrics are derived from the individual tasks performed by a module. Metrics based on these slices provide a means of quantifying the interaction between the computational strands of the module, and thus its cohesion. Meyers et al. (2004) report that their longitudinal study of slice-based cohesion metrics indicates that the metrics quantify overall code quality.

The tasks of a module are represented by the changed variables or output in its execution. The slices used in the calculation of slice-based metrics are taken in respect of these variables at the last point of change, or at the point of output. The variables on which the slices are based are known as “output variables”. However, there appears to be no standard definition or use of these variables in slice-based metrics and their description by different research groups varies.

Harman et al. (1995) designed a slice-based coupling metric as an alternative to Henry and Kafura’s information flow. They suggest that basing the metric on slices “refine the idea of information flow which has traditionally been associated with coupling measurement”. While information flow notes only the presence of flow between one function and another, slice-based measures provide the number of nodes in a slice which are from another function, giving the notion of the “bandwidth” of the coupling. The flow from one module to another is calculated in terms of the vertices in the slices on each of the output variables in the module. Coupling between two modules is calculated from the normalized values of the flow in each direction.
Harman et al (1995) define the flow between two modules $f$ and $g$ in a program $p$, containing both $f$ and $g$, ($FF_{(f,g)}^p$), as the relative amount of the body of which contributes to the computation of the result(s) of $g$.

$$FF_{(f,g)}^p = \frac{\# \left( \bigcup_{v \in SS(p, [v], E(g))} SS(p, [v], E(g)) \bigg) \right) N(f)}{N(f)}$$

where: $SS(p, [v], E(g))$ means the slice on program $p$ with respect to a set of output variables $[v]$ at the last line of the module $g$, $p(g)$ denotes the output (principal) variables of module $g$, $N(f)$ denotes the number of nodes in module $f$. $N(f \rightarrow g)$ to mean the number of vertices (nodes) in a module, $f$, which are in the union of the slices on the output variables of the module $g$, and using Harman et al’s notation for the total number of vertices in a function $f$, $N(f)$, this can be stated as:

$$FF_{(f,g)}^p = \frac{N(f \rightarrow g)}{N(f)}$$

The coupling between two functions $f$ and $g$ is then defined as,

$$\text{Coupling}(f, g) = \frac{FF_{(f,g)}^p \times N(f) + FF_{(g,f)}^p \times N(g)}{N(f) + N(g)}$$

i.e.

$$= \frac{N(f \rightarrow g) + N(g \rightarrow f)}{N(f) + N(g)}$$

Meyers et al (2004) generalize this coupling between modules to give the coupling for a module as the weighted average of its coupling with each of the other modules in the program, giving the formula for coupling of a module $f$ as:
\[
\text{Coupling}(f) = \frac{\sum_{i} \text{coupling}(f, g_i) \times |g_i|}{\sum_{i} |g_i|}
\]

Longitudinal study of slice-based coupling show that while intra-project comparisons of slice-based coupling metrics can provide information about the design of a program, inter-project comparisons may be of less use due to the difference in design styles between projects.

2.2 METRICS FOR OO SOFTWARE

A significant number of object oriented metrics have been developed in literature. For example, metrics proposed by Abreu et al. (1994), C.K metrics Chidamber et al (1994), Li et al (1993) metrics, MOOD metrics Abreu et al (1996). C.K metrics are the most popular (used) among them. Another comprehensive set of metrics is MOOD metrics. This subsection will focus on traditional metrics and above mention metrics (mainly C.K and MOOD metrics).

2.2.1 Traditional Metrics

Linda et al (1997) suggests an object-oriented system, traditional metrics are generally applied to the methods that comprise the operations of a class. Methods reflect how a problem is broken into segments. Traditional metrics have been applied for the measurement of software complexity of structured systems. The following discussion shows three popular traditional metrics.

McCabe Cyclomatic Complexity (CC)

Complexity metrics can be used to calculate essential information about constancy and maintainability of software system from source code. It also provides advice during the software project to help control the design. In
the testing and maintain phase, complexity metrics provide detail information about software module to identify the areas of possible instability. Cyclomatic complexity (McCabe) can be used to evaluate the complexity of a method, suggested by Linda et al (1997). This metric measures the complexity of the control flow graph of a method or procedure. The idea is to draw the sequence a program may take as a graph with all possible paths. The complexity is calculated as “connections - nodes + 2” and will give a number denoting how complex the method is. Since complexity will increase the possibility of errors, a too high McCabe number should be avoided as Jacobson et al, (1992).

As described by Laing Coleman and McCabe et al (1999) mention cyclomatic complexity is a measure of a module control flow complexity based on graph theory. Cyclomatic complexity cannot be used to measure the complexity of a class because of inheritance, but the cyclomatic complexity of individual methods can be combined with other measures to evaluate the complexity of the class. A high cyclomatic complexity indicates that the code may be of low quality and difficult to test and maintain.

**Source Lines of Code (SLOC)**

SLOC is used to estimate the total effort that will be needed to develop a program, as well as to calculate approximate productivity. The SLOC metric measures the number of physical lines of active code, that is, no blank or commented lines code. Logical SLOC measures the number of statements, but their specific definitions are fixed to specific language for example, in C programming language logical SLOC measure the terminating semicolon. Since functionality is not as much interconnected with SLOC, expert developers may be capable to develop the same functionality with less code. Therefore, one program with less SLOC may show more functionalities than another similar program. Programs with larger SLOC values usually take
more time to develop. Therefore, SLOC can be very effective in estimating effort. Thresholds for evaluating the SLOC measures vary depending on the coding language used and the complexity of the method.

**Comment Percentage (CP)**

The CP metric is defined as the number of commented lines of code divided by the number of non-blank lines of code. The comment percentage is calculated by the total number of comments divided by the total lines of code less the number of blank lines. The SATC has found a comment percentage of about 30% is most effective.

**2.2.2 CK Metrics Model**

Chidamber and Kemerer define the so called CK metric suite. This metric suite offers informative insight into whether developers are following object oriented principles in their design. They claim that using several of their metrics collectively helps managers and designers to make better design decision. CK metrics have generated a significant amount of interest and are currently the most well known suite of measurements for OO software. Chidamber and Kemerer proposed six metrics; the following section discusses their metrics in detail.

**Weighted Method per Class (WMC)**

WMC measures the complexity of a class. Complexity of a class can for example be calculated by the cyclomatic complexities of its methods. High value of WMC indicates the class is more complex than that of low values. So class with less WMC is better. As WMC is complexity measurement metric, we can get an idea of required effort to maintain a particular class.
**Depth of Inheritance Tree (DIT)**

DIT metric is the length of the maximum path from the node to the root of the tree. So this metric calculates how far down a class is declared in the inheritance hierarchy. This metric also measures how many ancestor classes can potentially affect this class. DIT represents the complexity of the behavior of a class, the complexity of design of a class and potential reuse. If DIT increases, it means that more methods are to be expected to be inherited, which makes it more difficult to calculate a class’s behavior. Thus it can be hard to understand a system with many inheritance layers. On the other hand, a large DIT value indicates that many methods might be reused.

**Number of Children (NOC)**

This metric measures how many sub-classes are going to inherit the methods of the parent class. The size of NOC approximately indicates the level of reuse in an application. If NOC grows it means reuse increases. On the other hand, as NOC increases, the amount of testing will also increase because more children in a class indicate more responsibility. So, NOC represents the effort required to test the class and reuse.

**Coupling Between Objects (CBO)**

The idea of this metric is that an object is coupled to another object if two object act upon each other. A class is coupled with another if the methods of one class use the methods or attributes of the other class. An increase of CBO indicates the reusability of a class will decrease. Thus, the CBO values for each class should be kept as low as possible. CBO metric measure the required effort to test the class.
Response for Class (RFC)

RFC is the number of methods that can be invoked in response to a message in a class. Since RFC increases, the effort required for testing also increases because the test sequence grows. If RFC increases, the overall design complexity of the class increases and becomes hard to understand. On the other hand lower values indicate greater polymorphism.

2.2.3 MOOD Metrics Model

Abreu et al (1996) defined MOOD (Metrics for Object Oriented Design) metrics. MOOD refers to a basic structural mechanism of the object-oriented paradigm as encapsulation (MHF, AHF), inheritance (MIF, AIF), polymorphism (POF), and message passing (COF). Each metrics is expressed as a measure where the numerator represents the actual use of one of those feature for a given design. In MOOD metrics model, two main features are used in every metrics; they are methods and attributes. Methods are used to perform operations of several kinds such as obtaining or modifying the status of objects. Attributes are used to represent the status of each object in the system. Each feature (methods and attributes) is either visible or hidden from a given class. We will now discuss MOOD metrics in the context of encapsulation, inheritance, polymorphism, and coupling.

Method Hiding Factor (MHF)

The MHF metric states the sum of the invisibilities of all methods in all classes. The invisibility of a method is the percentage of the total class from which the method is hidden. Abreu et al (1994) states, the MHF denominator is the total number of methods defined in the system under consideration. If the value of MHF is high (100%), it means all methods are private which indicates very little functionality. Thus it is not possible to
reuse methods with high MHF. MHF with low (0%) value indicate all methods are public that means most of the methods are unprotected.

**Attribute Hiding Factor (AHF)**

The AHF metric shows the sum of the invisibilities of all attributes in all classes. The invisibility of an attribute is the percentage of the total classes from which this attribute is hidden. MHF and AHF represent the average amount of hiding among all classes in the system. If the value of AHF is high (100%), it means all attributes are private. AHF with low (0%) value indicates all attributes are public.

**Method Inheritance Factor (MIF)**

The MIF metric states the sum of inherited methods in all classes of the system under consideration. It is the degree to which the class architecture of an object oriented system makes use of inheritance for both methods and attributes. MIF is defined as the ratio of the sum of the inherited methods in all classes of the system. If the value of MIF is low (0%), it means that there is no methods exists in the class as well as the class lacking an inheritance statement.

**Attribute Inheritance Factor (AIF)**

AIF is defined as the ratio of the sum of inherited attributes in all classes of the system. AIF denominator is the total number of available attributes for all classes. It is defined in an analogous manner and provides an indication of the impact of inheritance in the object oriented software. If the value of AIF is low (0%), it means that there is no attribute exists in the class as well as the class lacking an inheritance statement.
**Polymorphism Factor (POF)**

Abreu et al (1996) suggests Polymorphism measure the degree of overriding in the class inheritance tree. The POF represents the actual number of possible different polymorphic situation. It also represents the maximum number of possible distinct polymorphic situation for class $C_i$. The numerator represents the actual number of possible different polymorphic situation and the denominator represents the maximum number of possible distinct polymorphic situation for class $C_i$. The value of POF can be varied between 0% and 100%. If a project have 0% POF, it indicates the project uses no polymorphism and 100% POF indicates that all methods are overridden in all derived classes.

**Coupling Factor (COF)**

Coupling shows the relationship between modules. A class is coupled to another class if it calls methods of another class. The COF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of coupling is not imputable to inheritance argues that, although many factors affect software complexity, understandability, and maintainability. It is reasonable to conclude that as “the COF value” increases, the complexity of object oriented design will also increase, and as a result the understandability, maintainability, and the potential for reuse may suffer. The value of COF can be varies between 0% and 100%. 0%COF indicates no class is coupled and 100% COF indicates all class is coupled with all other classes. High values of COF should be avoided. The idea in COF metric is as same idea used in CBO metrics because they both use coupling factor. The main difference between COF and CBO is, in COF metric all variable accesses are counted whereas CBO metric does not count variables.
2.2.4 Other OO Coupling Metrics

Lorenz et al (1994) proposes metrics are focused on size, inheritance, internal, and external measurements. Size metrics for the object oriented class focus on counts of attributes and operations for an individual class. Inheritance based metrics focus on the method in which operations are reused through the class hierarchy. Internal metrics are focus on cohesion and code oriented issue. External metrics observe coupling and reuse.

Abreu et al (1994) categorized metrics are: design, size, complexity, reuse, productivity, quality, method, class and system levels. They provide a catalogue for object oriented design metrics. That taxonomy is based on a Cartesian product of the two vectors: (design, size, complexity, reuse, productivity, quality) and (method, class, system). His proposed metrics are CC2 (Class Complexity), CR1 (Class Reuse), CC3 (Class Complexity), CR2 (Class Reuse), CR3 (Class Reuse). In his measure, class and system quality metrics that the authors suggest are based on counts of observed defects, failures, and time between failures.

The metrics proposed by Chen et.al. (1993) are: 1.CCM (Class Coupling Metric), 2.OXM (Operating Complexity Metric), 3.OACM (Operating Argument Complexity Metric), 4.ACM (Attribute Complexity Metric), 5.OCM (Operating Coupling Metric), 6.CM (Cohesion Metric), 7.CHM (Class Hierarchy of Method) and 8.RM (Reuse Metric). Metrics 1 through 3 are subjective in nature; metrics 4 through 7 involve counts of features; and metric 8 is a Boolean (0 or 1) indicator metric. To validate these metrics, the authors conduct an experiment involving six "experts" whose subjective class scores are regressed against the eight metrics. The resulting regression equation is used to score future object classes.
Li et al (1993) states that metrics for the object-oriented paradigm are objects, classes, attributes, inheritance, method, and message passing. They propose that each object-oriented basic concept implies a programming behavior. They assembled metrics like: Data Abstraction Coupling (DAC), Number of methods (NOM), Message Passing Coupling (MPC), and Number of semicolons per class (Size1), Number of methods per attributes (Size2). There is no individual breakdown of which of these metrics is significant in the prediction.

Class

MPC (Message Passing Coupling) measures the complexity of message passing among classes. Although messages are passed between objects, the types of messages passed are defined in classes. So that, message passing is calculated at the class level instead of the object level. WMC (Weighted Methods per Class) discuss the complexity of the methods. In general methods are small enough so that the complexity of each could be considered as equal to unity. RFC (Response For Class) metrics states the response set of a class consists of the set \( M \) of methods of the class, and the set of methods invoked directly by the methods in \( M \).

Attribute

Attributes define the properties of data object and take an instance of the data object, describe the instance as well as make reference to another instance in another table. For example, the AHF metric is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes for all classes.
Method

A message is a request that an object makes of another object to perform an operation. The operation executed as a result of receiving a message is called a method. For example, WMC metric is the sum of the complexities of all class methods. It calculates all declared methods and constructors of class. The RFC metric uses a number of methods to review a combination of a class's complexity and the amount of communication with other classes. The MPC metric define a class which sends a number of statements. This send statement is a message sent out from a method in a class to a method in another class. Size1 is defined as the number of non-command lines of source code and Size2 defined as the total count of the number of data attributes and the number of external local methods in a class.

Coupling / Cohesion

The most potential outcome with object oriented metrics is obtained using coupling metrics. In the context of design metrics, coupling and cohesion are used to measure a system's structural complexity. These are also used to assess design. A class is coupled with one or more classes if the methods of one class use the methods or attributes of the other classes. CK metrics suite includes measures for coupling and cohesion, the suite provide descriptive power for administrative concern. Mainly high level of coupling and low level of cohesion were associated with problems and maintainability. For example, CBO (Coupling between Object Classes) is the number of other class with which a class is coupled. CCM (Class Coupling Metrics) measures the coupling between class and other class; MPC (Message Passing Coupling) measures the complexity of message passing between classes as well as objects. Although messages are passed among objects, the types of messages passed are defined is class.
Linda et al (1997) suggests a class is cohesive when its parts are highly correlated. It should be difficult to split a cohesive class. Cohesion can be used to identify the poorly designed classes. High cohesion indicates good class subdivision. Low cohesion increases complexity, thereby increasing the likelihood of errors during development. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion.

Preethi et al (2011) proposes few other metrics like: 1. Number of associated classes within 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. 2. Total Associated Class (TAC): This metric gives the number of times all associated attributes of a particular class type are used by methods of a user class.

Conceptual Coupling between Object classes (CCBO), is based on the well-known CBO coupling metric. The definition of CCBO relies on previous definitions for CoCC metric. Let \( c_k \in C \) and \( c_j \in C \) be two distinct classes in the system. Each class has a set of methods \( \mathcal{M}(c_k) = \{m_{k1}, \ldots, m_{kr}\} \), where \( r = |\mathcal{M}(c_k)| \) and \( \mathcal{M}(c_j) = \{m_{j1}, \ldots, m_{jt}\} \), where \( t = |\mathcal{M}(c_j)| \). Between every pair of methods \( (m_{ki}, m_{ji}) \) there is a similarity measure \( \text{CSM}^P(m_{ki}, m_{ji}) \). We can similarly define the conceptual similarity between two classes \( c_j \) and \( c_k \), that is \( \text{CSC}^P \) as follows:

\[
\text{CSC}^P(c_k, c_j, t) = \begin{cases} 
1 & \text{if } \text{CSC}(c_k, c_j) \geq t \\
0 & \text{else}
\end{cases}
\]

The definition ensures that the conceptual similarity between two classes is symmetrical, as \( \text{CSC}(c_k, c_j) = \text{CSC}(c_j, c_k) \). Therefore, the conceptual similarity for a class \( c \) is defined as:
which is the sum of the parameterized conceptual similarities between a class \( c \) and all the other classes in the system.

**Inheritance**

Inheritance shows the relationship among classes and reuse earlier defined objects as well as variables and operators. Inheritance decreases complexity by reducing the number of operations and operators. There are some metrics used to measure the amount of inheritance. For example NOC metric measures the number of direct subclasses of a class. The size of NOC approximately indicates how an application reuses itself. DIT metric calculates how far behind a class is declared in the inheritance hierarchy. MIF and AIF allow expressing similarity between classes; the portrayal of generalization and specialization relations; and simplification of the definition of inheriting classes, by means of reuse.

### 2.3 ESTABLISHED COUPLING METRICS

#### 2.3.1 Coupling between Objects (CBO)

An object of a class is coupled to another if the methods of one class use the methods or attributes of the other. Coupling has an adverse effect on the maintenance, reuse and testing of a design and that excessive coupling between object classes is detrimental to modular design and prevents reuse. As the more independent a class is, the easier it is to reuse in another application. They state that inter-object class couples should be kept to a minimum in order to improve modularity and promote encapsulation. The
larger the number of couples, the higher the sensitivity to changes in other parts of the design, is making the maintenance more difficult.

Coupling Between Objects (CBO) for a class is a count of the number of other classes to which it is coupled. This definition is flexible in three ways.

- Which direction a class is coupled to another
- How a class is actually coupled to another
- The value to give a coupling relationship to distinguish its strength from another coupling

Emam et al (2001), Arisholm et al (2002) Suggest CBO will be strictly efferent coupling, in other words, only focusing on the outward coupling to foreign classes. The value that will be given to the coupling will be defaulted to one, but this research will experiment with various other values as well. These variations will be the novel part the proposed metric. Efferent coupling was chosen because it has been shown to be stronger at predicting class quality when compared to afferent coupling.

2.3.2 Coupling Complexity between Objects (CCBO)

Coupling-Complexity Between Objects (CCBO) attempts to give a weight to a coupling between a pair of classes. This weight will be either McCabe or CC as defined in the established metrics. This is essentially a fuse between an object-oriented metric with a metric that does not measure within the object-oriented paradigm. When Class A is coupled to another through a method invocation then the CC of that method's body is the value to be applied to the weight of the coupling between Class A and the dynamic type of the coupled class. This will generate multiple values for a coupling between two classes. The set for each pair is evaluated and is assigned the
greatest value within the set. If a class is coupled to another through merely an attribute reference or a field access expression, then the value for a weight is instead given a value of one. This is justified because the class coupled is not utilizing anything as complex as a method, it is only accessing an attribute.

2.3.3 Transitive Coupling Complexity between Objects (TCBO)

Class A can be coupled to Class B by a method invocation. However, Class B can potentially have a significant amount of efferent coupling. This can affect the reliability of Class A because it relies on the method and the stability of Class B. Therefore the coupling between Class A and Class B is affected by the transitive relationship of the method invocation.

2.3.4 $\alpha$ Variation

A variation to the Transitive Cyclomatic Complexity is the $\alpha$ variation. This variation takes into account the actual depth of the execution path a method invocation could potentially create. It is exactly the same as TCC except in one minor aspect. The transitive function $T_\alpha(m)$ computes the CC for each method in the transitive closure. However, at each computation of the CC it is multiplied by a $\alpha$ value which is the current height of the call stack.

2.3.5 Response Set for Class (RS)

The response set (RS) of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. RFC is simply the number of methods in the set, that is, RFC = #(RS). A given method is counted only once. Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes.
\[ RS = \bigcup_{\text{all} i} R_i = \bigcup_{i \in M} R_i \]

which gives the response set for a class where \( R_i \) is the set of methods called by the method \( i \) and \( M \) is the set of all methods in the class. If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding on the part of the tester. The complexity of a class increases with the number of methods that can be invoked from it.

### 2.3.6 Number of Associations (NAS)

Harrison et al (2002) presents a coupling metric, Number of Associations (NAS) metric, which is defined as the number of associations of each class, counted by the number of association lines emanating from a class on an Object Model diagram. This measure therefore also includes inheritance. The NAS metric is directly collectible from design documents. NAS is similar to CBO except that in CBO, if a class uses a method of another class more than once, then the CBO metric will count each usage as a separate occurrence of coupling. NAS, on the other hand counts repeated invocations as a single occurrence of coupling. Hence, we expect values of CBO for a class to be greater than the NAS for the same class. According to the authors, NAS can be measured in the early design stage and hence helps the managers to obtain early coupling estimates. For a large scale system, measuring the coupling from object diagram is very difficult and costly as there will be a huge number of objects in a large scale system.
2.3.7 Message Passing Coupling (MPC)

Li et al (1993) proposes a coupling metric, Message Passing Coupling (MPC), which is defined as the number of method calls to other classes from within a given class. This is not to be confused with Response for Class (RFC) which measures all of the method calls, include those to the local methods. Only the method calls in local methods are counted. The authors also say “The local methods of a class constitute the interface increment.” Therefore method calls in inherited methods are not counted.

\[
\text{MPC}(c) = \sum_{m \in M_i(c)} \sum_{m' \in \text{SIM}(m)} \text{NSI}(m, m')
\]

where, NSI is the number of static invocations of methods not implemented in \(c\) by methods implemented in \(c\). \(m\) is a local method in class \(c\) which invokes a method \(m'\). This metric confines itself to class level, hence cannot be applied at subsystem level and thus is costly for large scale system.

2.3.8 Data Abstraction Coupling (DAC)

Data Abstraction Coupling (DAC), given by Li et al (1993) is defined as the number of abstract data types (ADTs) (say another class in the system) defined in a class. The number of variables having an ADT type may indicate the number of data structures dependent on the definitions of other classes. ‘\(T(a)\)’ is the type of attribute and ‘\(a\)’ is the attribute and C is the set of all classes in the system. This metric confines itself to class level, hence cannot be applied at subsystem level and thus is costly for large scale system.

2.3.9 Coupling Factor (COF)

The coupling factor (COF) metric, given by Abreu et al (2000) represents the actual number of client-server relationships between classes
that are not related via inheritance to the maximum possible number of such client-server relationships. It is normalized between 0 and 1 to allow comparisons between systems of different sizes. To obtain the actual number of couplings, the metric goes through each class in the system (considering all its methods and attributes) and finds its relationships to all other classes in the system. This metric confines itself to class level, hence cannot be applied at subsystem level and thus is costly for large scale system.

2.3.10 Coupling Dependency Metric (CDM)

The coupling dependency metric (CDM), given by Schach, is the sum of three components: a measure of the extent to which a program relies on its declarations remaining unchanged (referential dependency); a measure of the extent to which a program relies on its internal organization remaining unchanged (structural dependency); and a measure of the vulnerability of data elements in one module to change by other modules (data integrity dependency). This metric confines itself to class level, hence cannot be applied at subsystem level and thus is costly for large scale system.

2.3.11 Other Static Coupling Metrics

Li et al (1993) identifies a number of metrics that can predict the maintainability of a design. They define two measures, message passing coupling (MPC) and data abstraction coupling (DAC). MPC is defined as the number of send statements defined in a class. The number of send statements sent out from a class may indicate how dependent the implementation of the local methods is on the methods in other classes. MPC only counts invocations of methods of other classes, not its own. DAC is defined as the number of abstract data types (ADT) defined in a class.

An ADT is defined in a class if it is the type of an attribute of class. It is also specified that the number of variables having an ADT type
may indicate the number of data structures dependent on the definitions of other classes". Martin describes two coupling metrics that can be used to measure the quality of an object-oriented design in terms of the interdependence between the subsystems of that design. Afferent Coupling (CA) is the number of classes outside this category that depend upon classes within this category. Efferent Coupling (CE) is the number of classes inside this category that depend upon classes outside this category. A category is a set of classes that belong together in the sense that they achieve some common goal. Martin does not specify exactly what constitutes dependencies between classes.

Abreu et al (2000) presents a coupling metric known as Coupling Factor (COF) for the design quality evaluation of object-oriented software systems. COF is the actual number of client-server relationships between classes that are not related via inheritance divided by the maximum possible number of such client-server relationships. It is normalized to range between 0 and 1 to allow for comparisons for systems of different sizes. It was not specified how to account for such factors as polymorphism and method overriding.

Briand et al (1999) performs a comprehensive empirical validation of product measures, such as coupling and cohesion, in object-oriented systems and explore the probability of fault detection in system classes during testing. They define a number of measures which count the number of class-attribute (CA), class-method (CM) and method-method (MM) interactions for each class. They take into account which class the interactions originate from or are directed at and the number of ancestor or other classes. A CA-interaction occurs from class c to class d if an attribute of class c is of type class d. A CM-interaction occurs from class c to class d if a newly defined method of class c has a parameter of type class d. An MM-interaction occurs from class c to class d if a method implemented at class c statically invokes a newly
defined or overriding method of class d, or receives a pointer to such a method. This set has sixteen metrics in total.

### 2.3.12 Export Object Coupling (EOC)

Yacoub et al (1999) proposes a set of dynamic coupling metrics designed to evaluate the change-proneness of a design. These metrics are applied at the early development phase to determine design quality. The measures are calculated from executable object-oriented design models, which are used to model the application to be tested. They are based on execution scenarios that are “the measurements are calculated for parts of the design model that are activated during the execution of a specific scenario triggered by an input stimulus”. A scenario is the context in which the metric is applicable. The scenarios are then extended to have an application scope.

They define two metrics designed to measure the quality of designs, at an early development phase. Export Object Coupling $EOC_x(o_i, o_j)$ for an object $o_i$ with respect to an object $o_j$, is defined as the percentage of the number of messages sent from $o_i$ to $o_j$ with respect to the total number of messages exchanged during the execution of a scenario $x$. Import Object Coupling $IOC_x(o_i, o_j)$ for an object $o_i$ with respect to an object $o_j$, is the percentage of the number of messages received by object $o_j$ that were sent by object $o_i$ with respect to the total number of messages exchanged in the execution of a scenario $x$.

### 2.3.13 Arisholm Dynamic Coupling Metrics

Arisholm et al (2004) defines and validate a number of dynamic coupling metrics that are listed in Table 2.1. Each dynamic coupling metric name starts with either I or E to distinguish between import coupling and export coupling, based on the direction of the method calls. The third letter C.
or O distinguishes whether entity of measurement is the object or the class. The remaining letters distinguish three types of coupling. The first metric, C, counts the number of distinct classes that a method in a given class/object uses or is used by. The second metric, M, counts the number of distinct methods invoked by each method in each class/object while the third metric, D, counts the total number of dynamic messages sent or received from one class/object to or from other classes/objects.

Arisholm et al studies the relationship of these measures with the change proneness of classes. They find that the dynamic coupling metrics capture additional properties compared to the static coupling metrics and are good predictors of the change-proneness of a class. Their study uses a single software system called Velocity executed with its associated test suite, to evaluate the dynamic coupling metrics. These test cases are found to originally have 70% method coverage, which is increased to 90% for the methods that \might contribute to coupling” through the removal of dead code.
Table 2.1 Abbreviations for the dynamic coupling metrics of Arisholm

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_CC</td>
<td>Import, Class level, No. of Distinct classes</td>
</tr>
<tr>
<td>IC_CM</td>
<td>Import, Class level, No. of Distinct methods</td>
</tr>
<tr>
<td>IC_CD</td>
<td>Import, Class level, No. of Dynamic Messages</td>
</tr>
<tr>
<td>EC_CC</td>
<td>Export, Class level, No. of Distinct classes</td>
</tr>
<tr>
<td>EC_CM</td>
<td>Export, Class level, No. of Distinct methods</td>
</tr>
<tr>
<td>EC_CD</td>
<td>Export, Class level, No. of Dynamic Messages</td>
</tr>
<tr>
<td>IC_OC</td>
<td>Import, Object level, No. of Distinct classes</td>
</tr>
<tr>
<td>IC_OM</td>
<td>Import, Object level, No. of Distinct methods</td>
</tr>
<tr>
<td>IC_OD</td>
<td>Import, Object level, No. of Dynamic Messages</td>
</tr>
<tr>
<td>EC_OC</td>
<td>Export, Object level, No. of Distinct classes</td>
</tr>
<tr>
<td>EC_OM</td>
<td>Export, Object level, No. of Distinct methods</td>
</tr>
<tr>
<td>EC_OD</td>
<td>Export, Object level, No. of Dynamic Messages</td>
</tr>
</tbody>
</table>

Software metrics play a key role in the planning and in the control of software development projects. The Estimation of Coupling helps in the development of quality Software Products as well as in the maintenance. They reason out the structural complexity of software and to envisage the quality of the Software Product. Quality attributes such as Fault-proneness, ripple effect of changes and changeability are well predicted by coupling measures. Coupling or Dependency states the degree to which each individual program module relies on each one of the other modules. Coupling measures portray the static dependency among the classes in an object-oriented system. "Dynamic" couplings are not taken into an account due to polymorphism and may notably take too lightly the intricacy of software and misjudge the need for code inspection, testing and debugging. Most probably this results in hapless predictive exactness of the quality models that utilize static coupling measurement.