CHAPTER 4

METRICS FOR OBJECT-ORIENTED SYSTEMS

4.1 PROPERTIES OF DYNAMIC RELATIONSHIPS

Designing new dynamic metrics by Thomas et al (1999) must ensure that they effectively capture the aspect of software behavior that they are intended to measure. New metrics Lorenz et al (1994) must also render clear and comparable numbers for any kind of program. Therefore, we discuss some general requirements for dynamic metrics, which address some of the most important factors, which may impact their usefulness. These properties are not only helpful in designing the metrics, but can also be used in the evaluation of the applicability of a particular metric to specific purposes. These desirable properties Fenton et al (1999) are only presented informally; it may not be possible to realistically achieve all of them for every metric.

a) Dynamic: A metric should measure an aspect of a program that can only be obtained by actually executing it. The dynamic nature of a metric makes it unaffected by the addition of unexecuted code to the program; because code that is never executed will obviously never contribute to the measured value.

b) Robust: A robust metric should not be overly sensitive to the size of a program’s input. Using dynamic metrics the measures are heavily influenced by program behavior. A dynamic metric is robust if a “small” change in program behavior results in a correspondingly small change in the measured value.
c) Discriminating: A metric is discriminating if a large change in behavior causes a correspondingly large change in the resulting metric.

d) Unambiguous: It is crucial to provide a clear, precise and unambiguous definition of all dynamic metrics.

e) Platform Independent: When Metrics pertain to program behavior; they should not change if the measurement takes place on a different platform. While it may seem like platform independence, it is easily achieved in any language.

The structural relationships in classes are classified into the aggregation relationship, inheritance relationship and association relationship. If relationship of classes is aggregation relationship, in other words, class A includes class B and class C, change of class A propagate to class B and class C directly and then life cycle among objects will be same. Namely, class B and class C should exist to execute the function of class A. If relationship of classes is inheritance relationship, in other words, class A is parent class and class B is a child class, class B should call the method to class A to use the method of class A when arbitrary class C calls the method to class B. That is because class B inherits a member variable and method of class A. If relationship of classes is association relationship, it is relationship by the method calls between class A and class B. These properties by the aggregation relationship, inheritance relationship and association relationship can be measured by the method calls among objects in object level. But existing dynamic coupling metrics do not consider the structural property between classes and if it considers the property, it does not consider aggregation relationship.

The interaction by method call among objects can be classified into reading method type and writing method type. If object A calls a reading
method to object B, object B does not affect other objects because it does not change a value. But if object A calls a writing method to object B, it affects other objects that operate the function by referring object B. Therefore, the weight should be endowed to measure coupling correctly between classes in object level. But the conventional coupling metrics are measured without method call types. The coupling between classes in object level is different according to the number of method calls. It Collects names from between classes, several times of method calls by objects that have much higher coupling than one time of method call. Therefore the coupling between classes in object level by the number of method calls increases linearly.

4.2 ESTABLISHED OBJECT-ORIENTED METRICS

Researchers have proposed metrics to measure a wide variety of attributes related to coupling and cohesion within Object-Oriented systems. Hitz et al (1995) categorized these metrics according to class level coupling (CLC) and object level coupling (OLC). CLC metrics are obtained from examining class diagrams and programming code, and measure the extent to which one class is likely to impact another class. They are useful in determining the difficulty of maintaining an application during its lifetime. In contrast, OLC metrics are obtained by examining models such as sequence diagrams and objects interact, and are useful in planning and managing activities such as testing and debugging. Neither of the categories of metrics has the ability to reflect true runtime behavior of a system, Briand et al, (1999). In true object-level coupling, the level of interaction is defined by the number of static links between two objects, the type of those links, and the frequency with which each of those links is used.
4.2.1 Effort Estimation and Prediction Metrics

The cost of development is one of the main issues that must be kept under control. To this end, a linear/ non-linear relationship between software complexity/size and effort is commonly assumed. The effort can be expressed in person-months, -days or -hours needed for system development, from requirements analysis to testing or in some cases only for coding. In this way, the problem of effort evaluation is shifted into the problem of complexity or size evaluation. When software complexity evaluation is performed on code, this can be useful for controlling costs and development process efficiency, as well as for evaluating the cost of maintenance, reuse, etc. When software evaluation is performed before system building, metrics are used for predicting costs of development, testing, etc. As pointed out by many authors, traditional metrics for complexity/ size estimation, often used for procedural languages, can be difficult to apply for evaluating object-oriented systems.

Several interesting studies on predicting and evaluating maintainability, reusability, reliability, and effort for development and maintenance have been presented. These relationships have been demonstrated by using validation processes Henderson et.al. (1996); Briand et al (1998); Nesi et al (1998); Basili et al (1996); Kemerer et al (1999).

4.2.2 Code Metrics

Traditional code metrics for complexity/size estimation, often used for procedural languages, are unsuitable to be directly applied for evaluating object-oriented systems Henderson et al (1994), Nesi et al (1998). by using the above procedural metrics, data structure and data flow aspects related to method parameters are neglected. More general metrics have been defined in which the external interface of methods is also considered in order to avoid this problem.
Operating with OOP leads to move human resources from the design/code phase to that of analysis, where class relationships are identified. Following evolutionary models for the development life-cycle (e.g., spiral, fountain, whirlpool, pinball), the distinction among phases is partially lost, e.g., some system parts can be under design when others are still under analysis. These aspects must be captured with specific metrics; otherwise, their related costs are immeasurable (e.g., the costs of specialization, the costs of object reuse, etc.). In order to cope with the above-mentioned drawbacks, specific code metrics for evaluating size and/or complexity of object-oriented systems have been proposed. Some of these metrics are based on well-known functional metrics, such as LOC, McCabe, etc. In Henderson et al (1994) and Nesi et al (1996), issues regarding the external and internal class complexities have been discussed by proposing several metrics. Early metrics, such as the number of local attributes, the number of local methods or the number of local attributes and methods have been frequently used for evaluating the conformance with the “good” application of the OOP. These are unfortunately too coarse for evaluating in details the development costs. The above metrics have been generalized and compared, by adding terms and weights opportune ofy estimated during a validation phase.

4.2.3 Method Level Metrics

At method-level classical size and volume metrics can be profitably used. In some cases, specific metrics including also the cohesion of methods are considered, for instance, by taking into account the complexity of method parameters. On the other hand, these metrics are only marginally more precise in estimating development effort than pure functional metrics, while they are quite useful for estimating effort for reuse.
4.2.4 Class Level Metrics

Class-level metrics have also to be taken into account of class specialization, class association and aggregation, to assess all the characteristics of system classes. A fully object-oriented metric for evaluating class complexity/size has to be considered also attributes and methods both locally defined and inherited. These factors must be considered for evaluating the cost/gain of inheritance adoption, and that of the other relationships. Therefore, the class complexity, $CC_m$, is regarded as the weighted sum of local and inherited class complexities (recursively, till the roots are reached), where $m$ is a basic metric for evaluating functional/size aspects such as, $Mc$, LOC, Ha. This is a generalization of the metrics proposed in and Chidamber et al (1994):

$$CC_m = w_{CACI_m}CACI_m + w_{CMICL_m}CMICL_m + w_{CL_m}CL_m + w_{CACI_m}CACI_m$$

$$+w_{CMICL_m}CMICL_m + w_{CI_m}CI_m$$

where $CACI_m$ is the class attribute complexity local, $CACI_m$ the class attribute complexity inherited, $CMICL_m$ the class method interface complexity of local methods; $CMICL_m$ the class method interface complexity of inherited methods; $CL_m$ class complexity due to local methods; $CI_m$ class complexity due to inherited methods (e.g., complexity reused). In this way, $CC_m$ is taken into account both structural and functional/behavioral methods, method “cohesion” by means of $CMICL_m$ among the possible values. These measures are in most cases an over-simplification of the real conditions because the complexity of a class obviously depends on all the super classes. In order to solve this problem, metric NSUP (number of super classes) has been proposed and compared with DIT in Bucci et al (1998).
In order to better analyze the class relevance within the class hierarchy, the number of its direct subclasses is very important to be evaluated. To this end, the so called NOC, number of children, metric has been defined. Metric NOC counts the number of children considering only the direct sub-classes. It ranges from 0 to N (where 0 is obtained in the case of a leaf class). Classes with a high number of children must be carefully designed and verified because their code is shared by all the classes that are deeper in the hierarchy. NSUB metric (number of subclasses, Bucci et al (1998)) counts all the subclasses until leaves are reached and, thus, it can be regarded as a more precise measure of class relevance in the system.

Therefore, these metrics can be useful for identifying critical classes, and are also strongly correlated with maintenance costs as demonstrated in Li et al (1993). Other metrics for assessing system structure can be: NRC, number of root classes; mean value of NM; mean value of NA; mean value of CC, etc. In general, complexity metrics (like CC, WMC, etc.) are unsuitable for evaluating comprehensibility of the system under assessment (for reuse and/or maintenance). For example, a metric that produces a general view of class understandability is CCGI (class cognitive index; Fioravanti et al (1998)), which is defined as follows:

\[
CCGI = \frac{ECD}{CC}
\]

\[
= \frac{CACI + CACL + CMICI + CMICL}{CACI + CACL + CMICI + CMICL + CI + CL}
\]

where ECD is the external class description and is defined as the sum of terms related to class definition of CC metric. With CCGI it is possible to identify classes with low understandability and select classes that can be used as a “black box”. These considerations arise from the definition of the class itself
that measures an index showing how much the class is understandable by looking only at its external interface.

A high value for CCGI means that ECD is very detailed with respect to the class complexity. For example, if a class presents several small methods in its definition, then it is more understandable than a class that, having the same total complexity or size, presents a lower number of members.

In object-oriented systems assessment, it is very important to take into account all the typical relationships that characterize OOP. This can be performed by using metrics such as CBO, coupling between objects. Several other coupling metrics have been reviewed and compared in Briand et al (1998, 1999). Even metrics CC and CCGI contain some terms related to the coupling among object, i.e., CMICI and CMICL. In that case, the coupling is partially considered by means of the parameter complexity of method calling.

In order to evaluate an objective quality profile of the system under assessment, a specific set of metrics is necessary and, therefore, the number of data that have to be managed by the system manager or reviewer may become very large. Therefore, a procedure for the automatic or semi-automatic identification of degenerative conditions according to OOP, quality and company reference profile is mandatory.

Three decision criteria are used to define and classify the run-time coupling measures. Firstly, a distinction is made as to whether the entity of measurement is the object or the class. Run-time object-level coupling quantifies the level of dependencies in between objects in a system. Run-time class-level coupling quantifies the level of dependencies in between the classes that implement the methods or variables of the caller object and the receiver object. The class of the object sending or receiving a message may be
different from the class implementing the corresponding method due to the impact of inheritance.

Secondly, the direction of coupling for a class or object is taken into account. This allows for the fact that in a coupling relationship a class may act as a client or a server, that is, it may access methods or instance variables from another class (import coupling) or it may have its own methods or instance variables used (export coupling). Finally the strength of the coupling relationship is assessed, that is the amount of association in between the classes. To do this it is possible to count either:

1. The number of distinct classes that a method in a given class uses or is used by.
2. The number of distinct methods invoked by each method in each class.
3. The total number of dynamic messages sent or received from one class or other classes.

The following are metrics for evaluating class-level coupling:

- **IC_CC**: This determines the number of distinct classes accessed by a class at run-time.
- **IC_CM**: This determines the number of distinct methods accessed by a class at run-time.
- **IC_CD**: This determines the number of dynamic messages accessed by a class at run-time.
- **EC_CC**: This determines the number of distinct classes that are accessed by other classes at run-time.
- EC_CM: This determines the number of distinct methods that are accessed by other classes at run-time.
- EC_CD: This determines the number of dynamic messages that are accessed by other classes at run-time.

Li et al (1993) defines the Message Passing Coupling (MPC) as the count of the number of send statements that is found in methods of one class to other classes. Chidamber et al (1994) introduces the Response for Class (RFC) as a measure of the number of methods that can potentially be executed in response to a message received by an object of that class. Chidamber et al (1994) defines Coupling between Object Classes (CBO) as “the count of the number of classes to which it is coupled” and further elaborates in the definition as “two classes are coupled when methods of one class use methods or instance variables defined by one class use methods or instance variables defined by the other class”. These measures are not a dynamic measure of coupling because it does not count the number of invocations during execution, but these count the number of methods and variables invoked.

4.2.5 Object Level Metrics

To evaluate object-level coupling it was deemed necessary to define just one metric. Since we want to examine the behavior of objects at run-time we require a measure that is based on a class rather than a method-level. Further, it was deemed necessary to evaluate only coupling at the import level, as we are interested in examining how classes use other classes at the object-level rather than how they are used by other classes, therefore export coupling for this measure was not evaluated.
The following is a measure for evaluating object-level coupling:

- **IC OC**: Import, Object-Level, Number of Distinct Classes: This measure will be some function of the static CBO measure, as this measure determines the classes that can be theoretically accessed at run-time. This is a coarse-grained measure which will assess class-class coupling at the object-level.

Yacoup et al (1999) defines Dynamic Couplings between Objects, i.e. EOC (Export Object Coupling), IOC (Import Object Coupling), OQFS (Object Request for Service) and OPFS (Object Response for Service). EOC (Export Object Coupling), the export coupling for object $O_i$ with respect to 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 the scenario $X$. IOC (Import Object Coupling), the import coupling for object $O_i$ with respect to object $O_j$, is defined as the percentage of the number of messages received by object $O_i$ and is sent by object $O_j$ with respect to the total number of messages exchanged during the execution of the scenario $X$.

OQFS (Object Request for Service) is defined as the percentage of the total number of messages sent by the object $O_i$ to all other objects in the design. OPFS (Object Response for Service) is defined as the percentage of the total number of messages sent to the object $O_i$ from all other objects in the application during the execution of a specific scenario $X$. This coupling metric is not a dynamic measure of coupling because it does not count the number of invocations during execution, but it counts the number of methods and variables invoked. Arisholm et al (2004) defines Dynamic Couplings between Objects and Classes in each level. Dynamic Couplings are defined as
the total number of Dynamic Messages (sent from one object to other objects, within the scope considered), the number of Distinct Method Invocations (invoked by each method in each object) and the number of Distinct Classes (that a method in a given object uses or is used by).

Another dimension to measure dynamic coupling is initiated by Hassoun, where they target to measure the influence of one object on others over a period of time, instead of import/export coupling concept. To propose a dynamic coupling metric DCM for measuring object level coupling for systems built on meta-level architectures, this metric is derived from the study that is coupling in between two objects that can be defined in terms of time during which one object influences the other. Two objects are said to be coupled if either one of them could influence the history of the other. The history of an object is defined as the sequence of its states in time.

4.3 UNIFIED REPRESENTATION OF OO COUPLING MEASURES

A large number of Object-Oriented coupling measures exist in the literature. Informal definitions of terminologies and metrics in coupling analysis bring about ambiguities in interpreting their meaning, thus the coupling computation is made difficult. This also makes it difficult to understand how different coupling measures relate to one another. By using standardized terminologies and formalism, we can express coupling measures in a consistent and unambiguous manner. Considering how hard it is to determine how such measures relate to one another and for which application they can be used, Briand et al. provides a unified framework for OO coupling measurement. This framework comes with a standardized terminology and formalism so that measures can be expressed in a consistent and operational manner Briand et al (1999).
Briand et al (1999) investigates the properties of couplings and proposes five mathematical properties. The motivation behind defining mathematical properties is that a measure must be supported by some underlying theory of the internal quality attribute that measures. The five proposed coupling properties are defined as follows. Let Coupling be a candidate measure for coupling of a class or an object-oriented system.

Relationships capture the connections in between classes of the respective coupling measure is focused on. As the coupling measure can measure import or export coupling (or both), \( \text{Outer} R(c) \) will denote the relevant set of relationships from class \( c \) or both. Let \( \text{Inter} R(c) = \bigcup_{c \in C} \text{Outer} R(c) \) be the set of interclass relationships insystem \( C \). The five coupling properties are:

1. **Non negativity**
   The coupling \([\text{of a class c of an object } - \text{oriented system C}]\) is nonnegative:
   \[ \text{Coupling}(c) \geq 0 \mid \text{Coupling}(C) \geq 0 \]

2. **Null value**
   The coupling \([\text{of a class c of an object } - \text{oriented system C}]\) is null if \([\text{Outer} R(c) | \text{Inter} R(C)]\) is empty:
   \[ \text{Outer} R(c) = \emptyset \Rightarrow \text{Coupling}(c) = 0 \mid \text{Inter} R(C) = \emptyset \Rightarrow \text{Coupling}(C) = 0 \]

3. **Monotonicity:**
   Let \( C \) be an object-oriented system and \( c \in C \) be a class in \( C \). Class \( c \) is modified to form a new class \( c' \) which is identical to \( c \) except that \( \text{Outer} R(c) \subseteq \text{Outer} R(c') \). Let \( C' \) be the object-oriented
system which is identical to $C$ except that class $c$ is replaced by class $c'$. Then,

$$[Coupling(c) \leq Coupling(c') | Coupling(C) \leq Coupling(C')]$$

4. Merging of classes:

Let $C$ be an object-oriented system, and $c_1, c_2 \in C$ be two classes in $C$. Let $c'$ be the class which is the union of $c_1$ and $c_2$. Let $C'$ be an object-oriented system which is identical to $C$ except that classes $c_1$ and $c_2$ are replaced by $c'$. Then

$$[Coupling(c_1) + Coupling(c_2) \geq Coupling(c') | Coupling(C) \geq Coupling(C')]$$

5. Merging of unconnected classes

Let $C$ be an object-oriented system, and $c_1, c_2 \in C$ be two classes in $C$. Let $c'$ be the class which is the union of $c_1$ and $c_2$. Let $C'$ be an object-oriented system which is identical to $C$ except that classes $c_1$ and $c_2$ are replaced by $c'$. If no relationships exist between classes $c_1$ and $c_2$ in $C$, then

$$[Coupling(c_1) + Coupling(c_2) = Coupling(c') | Coupling(C) = Coupling(C')]$$

4.4 STATIC VS. DYNAMIC SOFTWARE METRICS

In this section, we discuss benefits of dynamic metrics in comparison to their static counter-parts. Static measures are obviously simpler to collect because there is no need to run the software. Moreover, to obtain dynamic metrics, code or simulation models of the software system are needed, which are available very late in the software development lifecycle. Static metrics are widely used because they are easier to obtain, especially at the early stages of software development. However, the potential benefits of
dynamic metrics collected by executing the program outweigh the complexity and cost of measuring them.

Static metrics satisfy the purpose of judging the quantity attributes like size and complexity of the software artifacts. But they are less precise than dynamic metrics in measurement of the quality attributes of software such as reliability, testability, as static metrics are evaluated only by means of static inspection of the software artifacts. The quality of software systems is more dependent on the runtime behavior than the potential characteristics implied by the static analysis of the software system. Dynamic metrics are computed on the basis of the data collected during actual execution of the system, and thus directly reflect the quality attributes (performance, error rates etc.) of the software in its operational mode. Moreover, static metrics deal with the structural aspects of a software system, whereas runtime metrics also deal with the behavioral aspects of the system. For example, according to the results of a controlled experiment conducted by Briand et al., static coupling measures may be insufficient to explain discrepancies in changeability for object-oriented designs. Moreover, static metrics are somewhat constrained in their ability to deal with inheritance, polymorphism and dynamic binding issues since the runtime types of field access and method invocation are not known. However, dynamic metrics are capable to deal with such issues. The major differences between static and dynamic metrics are listed in Table 4.1.
Table 4.1. Comparison between Static and Dynamic metrics

<table>
<thead>
<tr>
<th>Static Metrics</th>
<th>Dynamic Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpler to collect</td>
<td>Difficult to obtain</td>
</tr>
<tr>
<td>Available at the early stage of software development</td>
<td>Accessible very late in software development life cycle</td>
</tr>
<tr>
<td>Less accurate than dynamic metrics in measuring qualitative attributes of software</td>
<td>Suitable for measuring quantitative as well as qualitative attributes of software</td>
</tr>
<tr>
<td>Deal with the structural aspects of the software system</td>
<td>Deal with the behavioral aspects of the system also</td>
</tr>
<tr>
<td>Inefficient to deal with dead code and OO features such as inheritance, polymorphism and dynamic binding</td>
<td>Dynamic metrics are capable to deal with all object-oriented features and dead code</td>
</tr>
<tr>
<td>Less precise than dynamic metrics for the real-life systems</td>
<td>More precise than static metrics for the real-life systems</td>
</tr>
</tbody>
</table>

Major benefit of using dynamic metrics in software engineering is their ability to more precisely measure the internal attributes of software like coupling, complexity etc., which have direct impact on quality factors of software such as reliability, testability, reusability, maintainability, performance, error-rates. In subsequent sections, different dynamic metrics proposed in literature till date are discussed and presented into different categories depending on their types.

4.5 DYNAMIC COUPLING METRICS

Dynamic metrics are the class of software metrics that capture the dynamic behavior of the software system and are usually obtained from the execution traces of the code or from the executable models. Dynamic
coupling metrics are used to measure actual coupling taking place between a pair of objects or classes at runtime in a software system. Dynamic coupling metrics are measured at object level and can be aggregated to class or system level. Moreover, dynamic coupling measures can be defined at different stages of software development lifecycle like design-time or coding-time. In subsequent sub-sections, we discuss and then compare different types of dynamic coupling metrics proposed in literature.

### 4.5.1 EOC and IOC Metrics

Yacoub et al (1999) propose object level dynamic coupling measures, Export Object Coupling (EOC) and Import Object Coupling (IOC) based on executable object-oriented design models and these models are generated using Real-Time Object Oriented Modeling (ROOM) language. The design models used to collect the coupling measures are a kind of sequence diagrams that allow execution simulation. The EOC or IOC count the number of messages sent between two distinct objects \( o_i \) and \( o_j \) in a given ROOM sequence diagram \( x \) (in opposite directions), divided by the total number of messages exchanged in \( x \). Thus, the result of each metric is the percentage that reflects the “intensity” of interactions between two objects in a particular direction related to the total number of object interaction in \( x \). These metrics are defined within a scenario scope, i.e., measurements are calculated for parts of the design model that are activated during the execution of a specific scenario triggered by an input stimulus. Then, these metrics can be extended to have an application scope, i.e., for all scenarios.

### 4.5.2 Arisholm Metrics Suite

The concept of import and export coupling given by Yacoub et al (1999) is extended by Arisholm to take into consideration by the direction as well as class level, where he proposes 12 dynamic coupling measures for
object-oriented software divided along three orthogonal dimensions: direction, mapping and strength. Out of these 12 metrics, six metrics are defined at object level and other six are defined at class level. Each dynamic coupling metric name (e.g., IC OC) starts with either IC or EC to distinguish between import coupling and export coupling based on direction of the coupling. Import coupling counts the messages sent from an object or class, whereas export coupling counts the messages received by an object or class. The next letter indicates the mapping level (‘O’ for Object and ‘C’ for Class).

Mapping level here defines the granularity level at which coupling is being measured by the concerned metric. The last letter in the name of a particular metric denotes the strength of coupling as shown in Table 3. Here, strength of coupling measures the amount of association between the two objects. The amount of association between the objects may be quantified at three levels of granularity: 1) Dynamic messages (D): the number of times each message is sent from one object to another; 2) Distinct method invocations (M): the number of distinct method invocations between two objects; 3) Distinct classes (C): the number of distinct classes involved in association between the objects. The twelve metrics proposed by Arisholm are defined in Table 4.2.

Arisholm et al extends the work done by Arisholm by formally defining the dynamic coupling measures in an operational form and validating them theoretically as well as empirically.
Table 4.2. Dynamic Coupling Metrics by Arisholm

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IC_OD</strong></td>
<td>Import Coupling (IC) Object (O) Dynamic messages (D)</td>
<td>This measure counts the total no. of messages sent from one object to other objects</td>
</tr>
<tr>
<td><strong>IC_OM</strong></td>
<td>Distinct methods (M)</td>
<td>This measure counts the no. of distinct methods invoked from one object to other objects</td>
</tr>
<tr>
<td><strong>IC_OC</strong></td>
<td>Distinct classes (C)</td>
<td>This measure counts the no. of distinct server classes used by the methods of the given object</td>
</tr>
<tr>
<td><strong>IC_CD</strong></td>
<td>Class (C) Dynamic messages (D)</td>
<td>This measure counts the total no. of messages sent by all methods in all objects of a class.</td>
</tr>
<tr>
<td><strong>IC_CM</strong></td>
<td>Distinct methods (M)</td>
<td>This measure counts the no. of distinct methods invoked by all methods in all the objects of a class.</td>
</tr>
<tr>
<td><strong>IC_CC</strong></td>
<td>Distinct classes (C)</td>
<td>This measure counts the no. of distinct server classes used by all methods in all the objects of a class.</td>
</tr>
<tr>
<td><strong>EC_OD</strong></td>
<td>Export Coupling (EC) Object (O) Dynamic messages (D)</td>
<td>This measure counts the total no. of messages received by one object from other objects</td>
</tr>
<tr>
<td><strong>EC_OM</strong></td>
<td>Distinct methods (M)</td>
<td>This measure counts the no. of distinct methods received by an object.</td>
</tr>
<tr>
<td><strong>EC_OC</strong></td>
<td>Distinct classes (C)</td>
<td>This measure counts the no. of distinct client classes that in a given object are being used.</td>
</tr>
<tr>
<td><strong>EC_CD</strong></td>
<td>Class (C) Dynamic messages (D)</td>
<td>This measure counts the total no. of messages received by all methods of all objects of a class.</td>
</tr>
<tr>
<td><strong>EC_CM</strong></td>
<td>Distinct methods (M)</td>
<td>This measure counts the no. of distinct methods received by all methods of all objects of a class.</td>
</tr>
<tr>
<td><strong>EC_CC</strong></td>
<td>Distinct classes (C)</td>
<td>This measure counts the no. of distinct client classes that in all objects of a given class are being used.</td>
</tr>
</tbody>
</table>

4.5.3 Mitchell and Power Metrics Suite

As discussed above, Arisholm et al measures the amount of import and export coupling between objects at different levels, whereas Mitchell and Power propose to measure degree of import and export coupling between objects. Mitchell and Power define a set of dynamic coupling metrics as given in Table 4.3. The first three metrics in Table 4 are defined on the basis of
static coupling metric, CBO. They are designed to apply to an application at runtime and provide a means to evaluate class level coupling.

**Table 4.3 Dynamic Coupling Metrics by Mitchell and Power**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic CBO for a class</td>
<td>This metric is a direct translation of the C&amp;K CBO metric, except it is defined at runtime.</td>
<td>No. couples of a class with other classes at runtime</td>
</tr>
<tr>
<td>Degree of dynamic coupling</td>
<td>No. times a class A accesses methods or instances variables from a class B as a percentage of the total number of methods or instance variables accessed by A.</td>
<td>$\frac{\text{No. of times class } A \text{ accesses methods or instance variables from class } B \text{ at runtime}}{\text{Total no. of accesses to any method or instance variables}} \times 100$</td>
</tr>
<tr>
<td>within a given set of classes</td>
<td>This metric is an extension of above metric, to indicate the level of dynamic coupling occurring within a given set of classes.</td>
<td>$\frac{\text{Sum of no accesses to methods or instance variables outside each class}}{\text{Sum of total no of accesses from these classes}} \times 100$</td>
</tr>
<tr>
<td>$R_i$</td>
<td>Runtime import coupling between objects</td>
<td>No. classes from which a given class accesses methods or instance variables at runtime</td>
</tr>
<tr>
<td>$R_{E}$</td>
<td>Runtime export coupling between objects</td>
<td>No. classes which access methods or instance variables from a given class at runtime</td>
</tr>
<tr>
<td>$R_{DI}$</td>
<td>Runtime import degree of coupling</td>
<td>$\frac{\text{No accesses to class } B \text{ makes}}{\text{Total no accesses}}$</td>
</tr>
<tr>
<td>$R_{DE}$</td>
<td>Runtime export degree of coupling</td>
<td>$\frac{\text{No accesses made from class } A}{\text{Total no accesses}}$</td>
</tr>
</tbody>
</table>

The next four metrics ($R_{I1}, R_{E1}, R_{DI1}, R_{DE1}$) in Table 4.3 are defined to evaluate object level dynamic coupling. The metrics $R_{I1}$ and $R_{DE1}$ are an improvement over metrics $R_i$ and $R_{DE}$ as they are normalized and may be more useful in comparing classes of different sizes.

Mitchell et al (2006) examines the relationship between static and dynamic coupling metrics in the context of the influence of instruction coverage. The main measures used in this study are the static coupling metric
(CBO), six dynamic metrics (IC_CD, IC_CM, IC_CC, EC_CD, EC_CM, EC_CC) proposed by Arisholm et al and instruction coverage measure ($\iota_c$). The instruction coverage measure $\iota_c$ corresponds to the Java bytecode instructions. The results of study indicate strong influence of coverage measures on the correlation between static and dynamic metrics.

4.5.4 Dynamic Coupling Metric

This section summarizes the set of dynamic coupling metrics of object level using characteristics between classes.

[Definition 1] Classes in a System

A system consists of finite classes. If the system is referred to as $S$ and the involved classes in the system $S$ are referred to as $C_i (i = 1 ... l)$, then the system $S$ is defined as follows:

$$S = \{C_1, C_2, \ldots , C_l\}$$

[Definition 2] Methods of the Class

The methods of each class $M(C_j)(j = 1 ... k)$ are defined as follows:

$$M(C_j) = \{m_{j1}, m_{j2}, \ldots , m_{jn}\}$$

[Definition 3] Method Calls between Classes

Interactions between classes depend on method calls between classes in object level. If methods $m' \in M(C_g)$, $m \in M(C_y)$ exist for objects of different classes $C_g (1 \leq g \leq m)$, $C_y (1 \leq y \leq m)$ and the method $m'$ calls $m$, it is defined as $(m', m)$.
[Definition 4] Method Calls by Method Types between classes in object level

Interactions between classes in object level depend on method calls between classes. In case of different classes $C_g(1 \leq g \leq m), C_y(1 \leq y \leq m)$ interactions between objects of class $C_g$ and $C_y$ by method calls are defined as follows:

\[
Calling(C_g, C_y) = \{ m \in M(C_y) \mid \exists \text{some } g \neq y \}
\]

\[
\exists m' \in M(C_g) \text{ s.t. } (m', m)
\]

\[
Called(C_g, C_y) = \{ m' \in M(C_g) \mid \exists \text{some } g \neq y \}
\]

\[
\exists m \in M(C_y) \text{ s.t. } (m, m')
\]

Interactions between objects of classes according to the two method call types such as “Write” and “Read” are defined as follows:

a. In case that the objects of class $C_g$ and class $C_y$ send a message to write data each other:

\[
Calling(C_g, C_y) = :W(C_g, C_y)
\]

\[
Called(C_g, C_y) = :W(C_g, C_y)
\]

b. In case that the objects of class $C_g$ and class $C_y$ send a message to read data each other.

\[
Calling(C_g, C_y) = :R(C_g, C_y)
\]

\[
Called(C_g, C_y) = :R(C_g, C_y)
\]
[Definition 5] Weights by Method Call Types between Classes

Since the degree of dependency between classes depends on method call types, weights according to the method call types should be given. Thus, weights \( W \) according to method call types between classes are defined as follows:

a. The weight \( (W) \) in case that the objects of class \( C_g \) and class \( C_y \) send a message to write:

\[
W\left(W\left(C_g, C_y\right)\right) = W(W(C_g, C_y)) := W_w
\]

b. The weight \( (W) \) in case that the objects of class \( C_g \) and class \( C_y \) send a message to read:

\[
W\left(R\left(C_g, C_y\right)\right) = W(R(C_g, C_y)) := W_r
\]

We can see that the connectivity strength of dynamic dependency relationship between objects of classes has weakened as per the following ranking: Write \( > \) Read. Therefore, different weights are assigned to the message call types in the order of \( W_w > W_r \).

[Definition 6] Number of Method Calls by Method Types

In case of different classes \( C_g, 1 \leq g \leq m \), \( C_y, 1 \leq y \leq m \), Number of method calls by method types between objects of Class \( C_g \) and \( C_y \) is defined as follows:

a) Number of method call for writing between classes in object level

\[
|W(C_g, C_y)| + |W(C_g, C_y)| := nwc
\]
b) Number of method call for reading between classes in object level

\[
\left| R(C_i, C_y) \right| + \left| R(C_i, C_y) \right| := nrc
\]

[Definition 7] Maximum Number of Available Method Calls by Method Types

Maximum number of method calls by each method type is defined as follows:

a. Maximum number of available method call for writing between classes in object level

\[
\max \left( \left| W(C_i, C_y) \right| + \left| W(C_i, C_y) \right| \right) := mnwc
\]

b. Maximum number of available method call for reading between classes in object level

\[
\max \left( \left| R(C_i, C_y) \right| + \left| R(C_i, C_y) \right| \right) := mnrc
\]

[Definition 8] The Dynamic Coupling between Classes in Object Level (DCCOL)

\[
DCCOL(C_i, C_j) = W_w \times \frac{nwc}{mnwc} + W_r \frac{nrc}{mnrc}
\]

where:

- \( C_i, C_j \): Classes that relationship exist and \( C_i \neq C_j \)
- \( W_w(\text{write}), W_r(\text{read}) \): Weights about types of method call

\[
W_w(0.8) + W_r(0.2) = 1.0
\]
4.5.5 Modified Import Coupling Metric

Zaidman et al (2004, 2006 & 2006) propose a variation of import coupling and use the same for the purpose of program comprehension. Authors consider the following properties of a coupling metric in order to be useful for the purpose of program comprehension:

- Since, software engineers try to comprehend the software at the class-level only. While selecting dynamic coupling measures, only those metrics which are defined at class-level need to be considered.
- All classes external to the actual project (e.g., library classes), have no direct influence on the program comprehension process.
- Only those classes that have a prominent role within the system’s architecture need to be considered and such classes are expected to give orders to other classes, i.e., tell them what to do and what to give in return. As such, these classes are expected to request the services of other classes. This suggests that direction of coupling needs to be taken into consideration is the “import coupling” for the purpose of program comprehension.

Out of twelve metrics proposed by Arisholm et al (2002), two metrics IC_CM and IC _C adhere to the criteria set out as above, namely: working at the class-level and measuring import coupling. Authors also use a variation of IC_CC metric and refer it as IC_CC’. This metric differs from IC_CC metric in the sense that IC_CC metric is targeted more towards finding the number of class collaborations, while IC_CC’ retrieves the number of method-collaborations.
Consider the following example Chhabra et al (2010): A class having one method calls two distinct methods of second class and one method of third class. IC_CC is calculated as two (number of distinct classes called) and is calculated as three (number of distinct methods called). These dynamic coupling measures allow us to identify the most need-to-be-understood classes in a system. Detecting these classes very early in the program comprehension process allows the end user to pay attention towards these classes and start exploring the software system directly from there. Authors experiment with various dynamic coupling metrics and also compare direct and indirect coupling solutions. To simulate the indirect coupling, they use the HITS web-mining algorithm. Their experiments show that taking indirect coupling into account delivers better results for program comprehension.

4.6 COMPARISON OF DYNAMIC COUPLING MEASURES

In above subsections, a number of different metrics have been presented for measuring dynamic coupling. These metrics are defined at different stages of software development and are useful for different purposes. First type of metrics, EOC and IOC are defined at design-time and are derived from dynamic design models (models depicting execution scenarios). EOC and IOC metrics can affect many of the quality attributes such as maintainability, understandability, reusability and error-propagation. Objects having higher values of EOC or IOC would be more critical to changes due to maintenance and are more likely to export or import these maintenance changes to other objects.

Moreover, objects having higher values of EOC or IOC metrics are harder to understand since their dynamic behavior tightly depends on each other. Objects with higher EOC or IOC are less reusable because they strongly depend on each other and are more likely to be used together. Further, objects with high EOC or IOC are more likely to be a source of error
propagation since errors are more likely to propagate from the faulty source to
the destination object as a result of the frequent messages exchanged between
them.

However, these metrics do not give a precise depiction of the actual runtime situation as they are calculated during the early design stage of a program. Further, EOC and IOC metrics do not comply with the coupling properties for object-oriented software systems described in the axiomatic framework given by Briand et al. (1999) and these metrics do not account for inheritance and polymorphism. Second type, Arisholm’s dynamic coupling metrics are defined at actual run-time and quantify the flow of messages between objects at runtime.

These metrics adhere to the theoretical framework for coupling measures proposed for object-oriented software systems. It has been shown that these metrics are complementary to simple size measures and static coupling measures. Moreover, authors show that dynamic coupling measures capture different properties than static coupling measures, though some degree of correlation exists between them. Authors also demonstrate that dynamic export coupling measures are good indicators of change proneness of software systems. However, Arisholm et al (2002) define and study dynamic coupling measures as stand-alone metrics and do not consider the effect of code coverage measures on the proposed measures in detail.

Third type, Mitchell and Power metrics are an extension of static CBO coupling metric as defined in Table 4.4. The Dynamic CBO and Degree of Dynamic Coupling between classes’ metrics are proposed to quantify the external complexity of a class at runtime. Degree of Dynamic Coupling within a given set of classes metric is defined to determine external complexity within a group of classes. These metrics are directly related to testability and maintainability. The greater the level of coupling present, the
more rigorous testing needs to be done and the greater the Dynamic CBO is for a class, maintenance is more difficult as the class will be more sensitive to changes in other classes with which it is coupled. Authors [Mitchell & Power, 2004] show that the runtime metrics, $R_t, R_p, R_{hl}$ and $R_c$ capture different properties than the static metrics although some degree of correlation does exist. Authors in Mitchell Power et. al.(2006) successfully demonstrate that dynamic coupling metrics might be better interpreted in the context of coverage measures, rather than as stand-alone software metrics and for this purpose, they use dynamic coupling metrics proposed by Arisholm et al (2002).

Fourth type, DCM is a dynamic coupling measure for systems built on meta-level architectures and is calculated during the actual program execution. The DCM metric can be used to predict the runtime complexity of the system. The value of DCM metric in an object-oriented system has relation with quality of the system in terms of software maintenance. It may help system engineers to decide on the appropriate software components to be used in production and maintenance phase. Classes with high object couplings need more attention and consequently induce higher maintenance cost. Thus, these types of classes should be assigned to more experienced developers.

Further, DC Metric can be used to compare systems’ coupling at runtime and can also be used as a means of comparing runtime coupling of a system at different stages of its development. Knowledge of amount of coupling at runtime can also be helpful in making decisions on re-engineering and re-factoring. Zaidman et al (2006) uses two existing dynamic coupling metrics; IC_CM and IC_CC defined by Arisholm et al (2002) and proposes a metric IC_CC', which is a variation of IC_CC metric. These dynamic coupling measures are quite useful in identification of key classes for comprehension process in a software system. Authors’ work clearly indicates that dynamic coupling metrics and dynamic analysis with its goal-oriented
strategy, pay dividends when used for program comprehension purposes. The different types of dynamic coupling metrics as discussed above have been found to be indicators of external quality attributes as given in Table 4.4.

Table 4.4 Relations of Dynamic Coupling Metrics with Quality Attributes

<table>
<thead>
<tr>
<th>Metric</th>
<th>Quality Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOC &amp; IOC Metrics</td>
<td>Maintainability, Understandability, Reusability &amp; Error-propagation</td>
</tr>
<tr>
<td>Arisholm Metrics</td>
<td>Change proneness</td>
</tr>
<tr>
<td>Mitchell &amp; Power Metrics suite</td>
<td>External complexity, testability &amp; maintainability</td>
</tr>
<tr>
<td>DCM Metric</td>
<td>Maintainability</td>
</tr>
<tr>
<td>Zaidman et al Metrics</td>
<td>Program comprehension</td>
</tr>
</tbody>
</table>