Chapter 3

PROPOSED DYNAMIC METRIC SUITE

The history of static analysis of object-oriented systems is much more aged than that of dynamic analysis and so is true for the respective metrics representing the two types of analysis. With the arrival of dynamic/runtime metrics, a little more than a decade ago, arrived another debate in the world of software metrics. The debate was about the existence of the static metrics once dynamic metrics strengthen their stance. But as the dynamic metrics evolved, it was evident that they can’t replace static metrics because of their own limitations [Dufour et al.(2003b)]. It was also evident that both types of analysis must work together to propel the system towards maximum efficiency. As soon as the usefulness of dynamic analysis and inturn dynamic metrics was known, researchers followed it with a number of dynamic metric suites. The dynamic metric suites introduced by researchers like Mitchell et al.(2003a,2003b,2003c,2004a,2004b&2005), Arisholm(2002) etc can be divided by definition into following two categories:

a) Dynamic metrics that are defined to be the runtime versions of their static counterparts i.e. using the same definition as a static metric but evaluated at runtime. The researchers compare the static and dynamic analysis results for the same metric and try to figure out any useful correlation. e.g. Dynamic CBO for a class [Mitchell and Power(2004a)], Runtime Lack of Cohesion (LCOM) [Mitchell and Power(2004b)] etc.

b) Dynamic Metrics that are not exactly the runtime versions of some already known static metrics. Such dynamic metrics have some additional information added to their definitions that make them more useful than the static metric from which they were originated. This additional information could be some static analysis information (e.g. number of classes) or dynamic information. Examples of such metrics are Dynamic Complexity Metric (DCM) [Yacoub et al.(1999)], Degree of Dynamic Coupling between two classes [Mitchell and Power(2004a)], Class Request For Service (CQFS) [Zaidman and Demeyer(2004)] etc.

In spite of a decade long work on dynamic analysis, just a few metric suites have been devised for analyzing the dynamic behavior of object-oriented software systems. Knowing the usefulness of the dynamic analysis for software performance evaluation, these metrics are not enough as compared to the huge accumulation of static metrics over
the past few decades. Thus it is important to find out new metrics for dynamic analysis especially those that work on the key runtime features like inheritance, dynamic binding etc. Looking for more runtime metrics is also vital to keep a sufficient balance between the available static and dynamic metrics, because of the expected usefulness of their combination. Some of the important conclusions and future recommendations drawn from the latest research on dynamic metrics were listed in Chapter 1. Working on these conclusions and future recommendations, we have proposed a new dynamic metric suite for object-oriented systems.

3.1 Selecting Appropriate Object-Oriented Attributes

It was very important to carefully select the object-oriented internal attributes that would be measured by the proposed dynamic metrics. We have targeted coupling and inheritance attributes for the following reasons:

a) **Coupling:** Coupling has been found to be one of the best indicators of the external quality of object-oriented software by estimating the external attributes like maintainability, reusability, fault-tolerance etc. Thus no wonder most of the research done on dynamic metrics has also been on the coupling attribute of object-oriented systems [Mitchell and Power(2004a), Zaidman and Demeyer(2004), Arisholm(2002)]. Thus there is a solid base available to be worked upon for dynamic coupling metrics. It can act as an appropriate attribute to compare static to dynamic analysis as well as to empirically validate the dynamic metrics.

Interaction-based coupling was introduced by Coad and Yourdon(1991) as a type of coupling that results from the exchange of messages between two classes. The term ‘coupling’ means ‘interaction-based coupling’ when viewed from a static context. But it is not so when viewed from a dynamic/runtime context. As coupling is affected by the object-oriented features like inheritance and dynamic binding at runtime, the interaction-based coupling is bound to show the impact of such features in the dynamic results. Thus from a dynamic context, the type of coupling must be viewed as interaction-based coupling reflecting the effects of such dynamic features (i.e. inheritance, dynamic binding etc.) in its evaluation. We can call this combination *dynamic coupling*.

b) **Inheritance:** Researchers over the past decade have recommended an investigation into the impact of inheritance on dynamic coupling metrics [Mitchell and
Chapter 3 – Proposed Dynamic Metric Suite

Power(2004a)]. Chapter 2, section 2.7.1 covered the related work on effects of inheritance on internal quality attributes such as coupling, as well as on the external quality attributes such as maintainability etc. We need to devise such metrics that can measure coupling occurring due to inheritance at runtime and analyze its impact on the overall dynamic coupling. Figure 9 shows the difference between dynamic coupling (not using inheritance) and dynamic coupling (using inheritance) between two classes. Here classes c1 and c2 call a method md() of class d. Class c1 calls the method md() directly from class d without using subclass d1 of d, thus demonstrating dynamic coupling without inheritance, whereas class c2 calls md() via subclass d1 using inheritance demonstrating dynamic coupling using inheritance.

Dynamic coupling via inheritance can be used to find out the reasons behind the variations in the static and dynamic coupling results. We call such metrics as the *dynamic inheritance metrics*. No such dynamic metrics exist till date as per this study.

![Dynamic Coupling and Dynamic Inheritance](image)

**Figure 9: Dynamic Coupling and Dynamic Inheritance**

Inheritance-based coupling was introduced by Coad and Yourdon(1991). It was defined as a type of coupling that exists if one class is an ancestor of another class i.e. interconnections between generalizations and specializations. “Inheritance-based coupling” only considers the parent-child relationships among classes and that too from a static context. It does not include actual coupling caused using inheritance relationships at runtime which is the main concern of this research. Thus this work will not use the term
inheritance-based coupling to define any of the inheritance related dynamic metric, rather the term dynamic inheritance seems to be more appropriate.

Thus we would have two sets of metrics, one measuring dynamic coupling and the other measuring dynamic inheritance.

### 3.2 Proposed Dynamic Metrics for Object-Oriented Systems

The proposed set of metrics is divided into two broad categories i.e. dynamic coupling metrics and dynamic inheritance metrics. Each metric is explained with the help of the following:

- **Definition**: Defines a metric.
- **Literary Basis**: Already known metrics/measures (if any) on which the metric is based upon.
- **Need**: Explains the need of the metric in evaluating the behavior of object-oriented systems.
- **Metric Effect**: Lists the effect of metric on various external quality attributes. We have selected some of the most common external attributes for this study: maintainability, reusability, testability, portability etc.

Although the metrics have been defined using the famous Goal-Question-Metric (GQM)[Basili(1992)] approach, metrics are explained here starting with their definition and then mentioning their need. This is for a better understanding of metrics.

#### 3.2.1 Dynamic Coupling Metrics

**a) Dynamic Afferent Coupling (DCa)**

**Definition**

DCa is the percentage of number of classes accessing the methods of a class at runtime to the total number of classes. It can be formulated as:

\[
DCa(C) = \frac{M}{N} \times 100
\]  

*(Equation 1)*

where,

- \(M\) = Number of classes invoking the methods of class \(C\) at runtime.
- \(N\) = Total number of classes

**Literary Basis**

Afferent Coupling (Ca) by Martin(1994) and Export Coupling (EC) by Arisholm(2002).
Chapter 3 – Proposed Dynamic Metric Suite

**Need**

Ca metric counts the number of classes accessing the methods of a class evaluated from the static code. This count can vary at runtime as there may be some method calls that are counted by Ca but are not executed at runtime. DCa takes account of this problem as it is calculated at runtime only.

Another dynamic metric based on number of classes sending method calls at runtime is class-level Export Coupling (EC) at class granularity by Arisholm(2002).

DCa uses EC as it is a percentage metric that finds the number of classes accessing the methods of a class with respect to the total number of classes. Thus it gives an additional dimension to the EC count. Moreover, evaluating the count in percentage form helps in comparing the dynamic count with the static count. So whenever DCa is evaluated, it is evaluated in combination with the static Ca (SCa) metric so that the dynamic analysis results could be compared with the static analysis results in order to analyze the impact of dynamic object-oriented features like inheritance, polymorphism etc. Such a comparison can be summarized in the form of percentage variation in the metric value at runtime e.g. DCa is P% of SCa, to record a specific percentage drop or rise in the metric value at runtime.

Import Object Coupling (IOC) by Yacoub et.al. (1999) also calculates coupling at runtime but it finds it at object-level and it is a call-weighed (based on number of calls sent to an object) metric.

**Metric Effect**

Following is the metric impact on various external quality attributes:

- **Maintainability** is inversely related to DCa. This is because any maintenance action on a class will propagate the action to all the classes depending on that class. Thus all the classes depending upon a class have to undergo a change, every time a class changes. Thus more the number of classes accessing a particular class, more the value of DCa and hence, more the amount of maintenance effort required.

- **Testability** is also inversely related to DCa. To test every dependent class of a given class, the given class will have to be tested. Thus more the number of dependent classes, more the testing effort.

- **Reusability** is directly related to DCa. A high number of classes accessing a given class increase its reuse.
• *Portability* is inversely related to DCa. A higher value of DCa would mean high number of classes dependent on the given class and hence, making the class less portable.

• *Replaceability*: Classes showing a high dynamic afferent coupling could be highly inversely related to replaceability, since high number of classes depend on them. Hence, replaceability decreases with increasing DCa.

• *Security*: Classes showing a high dynamic afferent coupling may be inversely related to security, since they can be influenced in many ways from the other dependent classes. Security decreases with increasing DCa.

• *Reliability* is inversely related to DCa as reliability is directly related to maintainability and testability.

• *Fault-tolerance*: Classes showing a high dynamic afferent coupling could be inversely related to fault-tolerance, since a local fault might be propagated to the dependent classes. Fault-tolerance decreases with increasing DCa.

### b) Dynamic Key Server Class (DKSC)

**Definition**

DKSC is the percentage of number of calls sent to a class at runtime to the total number of static calls sent to all the classes. It is formulated as:

\[
\text{DKSC}(c_i) = \frac{\sum_{j=1}^{n-1} M(c_j,c_i) | c_j,c_i \in C, c_i \neq c_j} {MT_r} \times 100
\]  

(Equation 2)

\(M(c_j,c_i)\) = Number of messages sent from class \(c_j\) to \(c_i\).

\(C\) = Set of classes in the system.

\(MT_r\) = Total number of messages received from all the classes before runtime.

\(n\) = Total number of classes.

**Literary Basis**

This metric uses class-level EC metric (at message granularity) [Arisholm(2002)] to get the number of calls for each class.

**Need**

EC metric at dynamic message granularity level counts the number of calls sent to a class at runtime. DKSC utilizes EC to find its percentage over the number of static calls sent to
all the classes. This helps in comparing the dynamic count to the static count by taking the percentage over the total number of static calls in both the cases. DKSC adds another dimension to the EC metric by finding out the most coupled server class called ‘key server class’ that in turn shows the importance of maintenance of the class. If the class is overloaded with calls at runtime, it might be considered for redesigning. It may well happen that the static key server class does not come out to be the key server class at runtime because of the differences in behavior at runtime. So it is important to find the key server class at runtime. Also, EC metric is not yet extensively validated so it will only help if new dimensions are added to the metric at this point.

Import Object Coupling (IOC) by Yacoub et al. (1999) also calculates call-weighed coupling at runtime but it finds it at object-level whereas DKSC works at class-level.

**Metric Effect**

A high value indicates that the system is largely dependent upon the concerned class. The class with the highest value is called ‘Key Server Class’ for the system. A key server class has high complexity and high coupling. It has high reusability. A low value indicates that the methods of the concerned class are least demanded at runtime. This metric is used to find the Dynamic Key Class (DKC) (Most Active Class) of the system. Following is the metric impact on various external quality attributes:

- **Maintainability** may be inversely related to DKSC if the calls sent from the dependent client classes were well distributed among those classes. If DKSC is high but most of the calls are sent from selected number of classes, maintainability would be higher. Thus DKSC is not a good indicator of maintainability.
- **Testability** may be inversely related to DKSC if the calls sent from the dependent client classes were well distributed among those classes. In a case where DKSC is high but most of the calls are sent from selected number of classes, the class becomes more testable as only those selected number of classes would be dependent on the given class for testing. Thus DKSC is not a good indicator of testability.
- **Reusability** is directly related to DKSC. More the number of calls sent to a class higher the reuse.
- **Portability** may be inversely related to DKSC if the calls sent from the dependent client classes were well distributed among those classes. In a case where DKSC is
high but most of the calls are being sent from selected number of classes, the class becomes more portable. Thus DKSC is not a good indicator of portability.

- **Replaceability** may be negatively influenced by attributes assessed with DKSC if the calls sent were well distributed among the client/source classes. In a case where DKSC is high but most of the calls are being sent from selected number of classes, the class becomes more replaceable. Thus DKSC is not a good indicator of replaceability.

- **Security**: Classes showing a high DKSC than the other classes may be inversely related to security, since they are repeated accessed. Security decreases with increasing DKSC.

- **Reliability** is not dependent upon DKSC because of its independence from maintainability.

- **Fault-tolerance** may be negatively influenced by attributes assessed with DKSC if the calls sent from the dependent client classes were well distributed among those classes. In this way a fault in the server class would propagate to all the dependent client classes. In a case where DKSC is high, but most of the calls are being sent from selected number of client classes, the fault-tolerance would be higher because of limited fault propagation. Thus DKSC is not a good indicator of fault-tolerance.

Thus DKSC is not a good predictor of many of the external quality attributes. But a combination of DKSC and DCa can be a good indicator for maintainability, testability, portability etc i.e. to know the distribution of DKSC over the number of classes in DCa. Such a method can predict the quality factors related inversely to coupling. This combination of DKSC and DCa can also trace the client class that sends the maximum number of calls to the server class, hence making the highest contribution in strengthening coupling and in turn making the class less maintainable, less testable and less portable.

c) **Dynamic Key Client Class (DKCC)**

**Definition**

DKCC is the percentage of number of calls sent by a class at runtime to the total number of static calls sent by all the classes. It is defined as:

\[
DKCC(c_i) = \frac{\sum_{j=1}^{n-1} M(c_i, c_j) | c_i, c_j \in C, c_i \neq c_j}{MT_i} \times 100 \quad (Equation 3)
\]

\(M(c_i, c_j)\) = Number of messages sent from class \(c_i\) to \(c_j\).
C = Set of classes in the system.
MT_s = Total number of messages sent to all the classes before runtime.
n = Total number of classes.

Literary Basis
It uses class-level Import Coupling (IC) at message granularity level [Arisholm(2002)].

Need
IC metric at dynamic message granularity level counts the number of calls sent to a class at runtime. DKCC improves it to find this count’s percentage over the number of static calls sent by all the classes. This helps in comparing the dynamic count to the static count by taking the percentage over the total number of static calls in both the cases. DKCC adds another dimension to the IC metric by finding out the most coupled client class called key client class that in turn shows the importance of maintenance of the class. If the class is too much dependent on other classes, it might be considered for redesigning. It may well happen that the static client class does not come out to be the key client class at runtime because of the differences in behavior at runtime. So it is important to find the key client class at runtime. Also, IC metric is not yet extensively validated so it will help to add a new dimension to the metric at this point.

Metric Effect
The Key Client Class has high dynamic complexity and high dynamic coupling. It makes maximum number of method-level outgoing couples but doesn’t necessarily make the maximum number of class couples. This Metric is used to find the Dynamic Key Class (Most Active Class) of the system. Following is the metric impact on various external quality attributes:

- **Maintainability** may be negatively influenced by attributes assessed with DKCC as the client class with a high DKCC would be more sensitive to changes in its server classes. But if most of the calls are sent to a limited number of server classes, the maintainability may improve. Thus DKCC is not reliable maintainability predictor.

- **Testability** may be inversely related to DKCC if the calls sent were well distributed among the server classes. If a client class is dependent upon a high number of server classes, it is tough to test as it will inturn require server classes to be tested. In a case where DKCC is high but most of the calls are sent to selected number of
server classes, the client class becomes relatively easier to test. Thus DKCC is not a good indicator of testability.

- **Reusability** could be inversely related to DKCC. The greater the value of DKCC for a class, the greater the number of times the class needs other classes at runtime. This will discourage the reuse of this class, as it will not be independent in nature. But if most of the calls are sent to a limited number of server classes, the reusability of the client increases. Thus reusability cannot be predicted accurately with DKCC.

- **Portability** may be inversely related to DKCC if the calls sent were well distributed among the server classes. In a case where DKCC is high but most of the calls are sent to selected number of server classes, the client class becomes relatively more portable. Thus DKCC is not a good indicator of portability.

- **Replaceability** may be inversely related to DKCC if the calls sent were well distributed among the server classes. In a case where DKCC is high but most of the calls are sent to a selected number of server classes, the client class becomes relatively more replaceable. Thus DKCC is not a good indicator of replaceability.

- **Security** is not influenced by DKCC.

- **Reliability** is not dependent upon DKCC because of it doesn’t influence maintainability.

- **Fault-tolerance** may be negatively influenced by attributes assessed with DKCC if the calls sent were well distributed among the receiving server classes. In this way a faults in a faulty server classes would propagate to the client class. In a case where DKSC is high, but most of the calls are sent to selected number of server classes, the fault-tolerance would be higher because of limited fault propagation from server class to client class. Thus DKSC is not a good indicator of fault-tolerance.

Thus DKCC is not a good predictor of many of the external quality attributes. But a combination of DKCC and DCBO [Mitchell and Power(2004a)] can be a good indicator for maintainability, testability, portability etc i.e. to know the distribution of DKCC over the number of classes in DCBO. Such a method can predict the quality factors related inversely to coupling. This combination can also trace the strongest client-server coupling among all such pairs with a given server class, hence making a maximum contribution to strengthening coupling, and in turn making the class less maintainable, less testable, less reusable, less portable and less fault tolerant.
d) Dynamic Key Class (DKC)

Definition
DKC is the most active class in the system at runtime. DKC value for a class is the percentage of sum of calls sent out from the class and calls received by the class at runtime over the total number of static calls sent and received by all the classes.

\[
DKC(c_i) = \frac{\sum_{j \neq i} (M(c_j, c_i) + M(c_i, c_j)) \mid c_i, c_j \in C, c_i \neq c_j}{MT_{\text{sys}}} \times 100
\]  
(Equation 4)

M(c_i, c_j) = Number of messages sent to class c_i to c_j.
C = Set of classes in the system.
MT_{sys} = Sum of number of messages sent to and number of messages received from all the classes before runtime.

n = Total number of classes.

Literary Basis
This metric uses both IC and EC metrics at dynamic messages granularity level [Arisholm(2002)].

Need
DKC is a measure to calculate the two-way coupling of a class at runtime. No such dynamic two-way coupling metric is known till date. IC and EC metrics are value metrics whereas DKC is a percentage metric designed to compare the static and dynamic coupling. DKC can help to find out the most used or most active class at runtime. Again the static and dynamic values can vary showing the actual key class of the application.

DKC can further be used to extract more useful information about the system (i.e. at system-level). One such information is the percentage of number of active classes in the whole system (comprising of a set of classes). We use it to propose an extension of DKC metric named as Percentage Active Classes (PAC).

Percentage Active Classes (PAC)
PAC is the percentage of number of classes sending or receiving at least one method call from/to another class at runtime to the total number of classes.

\[
PAC(\text{System}) = \frac{P_{DKC>0}}{N} \times 100
\]  
(Equation 5)
Design and Validation of Dynamic Metrics for Object-Oriented Software Systems

\[ P_{DKC>0} = \text{Number of classes sending/receiving atleast one method call to/from another class (i.e. DKC>0) at runtime.} \]

\[ N = \text{Number of classes.} \]

**Metric Effect**

The Dynamic Key Class has high complexity, high coupling, low maintainability. It is the most active (coupled) class at runtime. A high DKC value indicates that a class is overloaded with calls (incoming and outgoing) at runtime. As DKC is a call-weighed metric, its dynamic results may vary a lot from their static counterparts for each class because of the variation in number of executed calls at runtime due to the effects of user input, inheritance and dynamic binding. Following is the metric impact on various external quality attributes:

- **Maintainability** may be negatively influenced by attributes assessed with DKC as the client class with a high DKC would have the high number of couples with other classes. But if most of the calls are sent to a limited number of server classes, the maintainability may improve. Thus DKC is not reliable maintainability predictor.

- **Testability** may be inversely related to DKC if the couples formed by the class use high number of unique classes. If a class is dependent upon a high number of server classes and, high number of client classes depends on that class, it is tough to test. In a case where DKC is high but most of the calls are sent to/received from a limited number of server/client classes, the client class becomes relatively easier to test. Thus DKC is not a good indicator of testability.

- **Reusability** factor increases with an increase of DKSC count whereas it decreases with an increase of DKCC count. Thus reusability must be a factor of a positive or a negative difference between the DKCC and DKSC that combine to evaluate the respective DKC value. Thus a class having a high DKC value can result in a high or low reusability depending upon the positive or negative differences between its DKCC and DKSC values.

- **Portability** may be inversely related to DKC if the couples formed by the class use high number of unique classes. If a class is dependent upon a high number of server classes and, high number of client classes depends on that class, it is tough to test. In a case where DKC is high but most of the calls are sent to/received from a limited number of server/client classes, the client class becomes relatively more portable. Thus DKC is not a good indicator of portability.
Thus DKC is not a good predictor of many of the external quality attributes. But a combination of DKC, DCBO and DCa i.e. knowing the distribution of DKC over the number of classes in DCBO and DCa, can be a good indicator for maintainability, testability, portability etc.

3.2.2 Dynamic Inheritance Metrics

a) Dynamic Percentage Inheritance Coupling (DPIC)

Definition

DPIC is the percentage of number of calls sent to a super class via all its subclasses to the sum of direct and indirect calls at runtime. It measures two types of inheritance for a class, single and hierarchical inheritance.

\[
\text{DPIC}(C) = \frac{\sum_{i=1}^{\text{TSb}} N_i(C)}{N_d(C) + \sum_{i=4}^{\text{TSb}} N_i(C)} \times 100
\]

(Equation 6)

TSb = Total number of subclasses of a class C.

\(N_d(C)\) = Number of calls sent to methods inherited by \(i^{th}\) subclass of class C at runtime.

\(N_d(C)\) = Number of direct calls sent to a superclass.

Need

There has been a need for the metrics that measure the amount of inheritance used in a software application. Mitchell and Power (2004a) showed the need of exploring the impact of inheritance on coupling at runtime. Thus there is a requirement for a set of metrics that can measure the contribution of dynamic inheritance coupling to the overall dynamic class coupling. DPIC finds the amount of coupling due to hierarchical inheritance out of the total coupling for a super class at runtime.

Metric Effect

DPIC gives a measure of amount of inheritance at runtime evaluated from a super class. A high DPIC value will indicate a high level of inheritance in a program. It will also indicate that super classes are being accessed largely through subclasses at runtime. Higher DPIC value will possibly mean a higher difference between static and DCa metric values (mostly an increase in metric value from static to dynamic analysis).
A 100% value will indicate no direct usage of the super class at runtime and therefore is fully dependent upon its subclasses to act. Subclasses in this case would act as an external interface for the super class.

On the other hand, a 0% value would indicate that the super class perhaps is needed to be redesigned. It also indicates that the super class need not be extended as its subclasses are not using its methods and variables at runtime.

The dynamic and static values of the metric can vary as there can be an unpredictable difference between the static method calls and the actual method calls. Following is the metric impact on various external quality attributes:

- **Maintainability** seems to be inversely related to DPIC. Any modifications to a superclass would not only affect the classes accessing the methods of the class via inheritance at runtime, but also its subclasses that are used to access those methods. If a high DPIC value is caused by just a limited number of classes, the maintainability would improve. Inheritance based systems are less maintainable than the non-inheritance based systems.

- **Testability** may be inversely related to DPIC as testing a class dependent on the given class would require it to be tested as well along with its subclasses that are used to build the communication, making the class tough to test. Lesser the classes involved in sending the method calls, easier the testing. Inheritance based systems are less testable than the non-inheritance based systems.

- **Reusability** would be high for a high DPIC class as inheritance boosts reuse.

- **Portability** seems to be inversely related to DPIC as higher the DPIC value higher the number of classes accessing it via its subclasses. Thus not only the classes sending method calls but also the subclasses they are using to send those calls, are dependent upon the superclass, and hence the superclass becomes less portable. Inheritance based systems are less maintainable than the non-inheritance based systems.

**b) Dynamic Method Inheritance Factor (DMIF)**

**Definition**

DMIF is the percentage of number of inherited methods of a subclass used at runtime to total number of methods of the class. It measures single and multiple inheritance for a class at runtime.
DMIF(C) = \frac{\sum_{i=1}^{TSp} M_{\text{inu}}(S_i(C))}{M_d(C) + \sum_{i=1}^{TSp} M_{\text{in}}(S_i(C))} \times 100 \quad (Equation 7)

TSp = \text{Number of super classes of a class C.}
S_i(C) = \text{ith superclass of C}
M_{\text{inu}}(S_i(C)) = \text{Number of inherited methods, from } i^{\text{th}} \text{ superclass of C, used at runtime.}
M_{\text{in}}(S_i(C)) = \text{Number of methods inherited from its } i^{\text{th}} \text{ superclass by C.}
M_d(C) = \text{Total number of methods that can be invoked in association with C excluding inherited methods.}

This definition of DMIF can be extended for the whole system as:

DMIF(System) = \frac{\sum_{j=1}^{TC} \sum_{i=1}^{TSp} M_{\text{inu}}(S_j(C_j))}{\sum_{j=1}^{TC} \left\{ M_d(C_j) + \sum_{i=1}^{TSp} M_{\text{in}}(S_i(C_j)) \right\}} \times 100 \quad (Equation 8)

Literary Basis
This metric uses the MOOD set’s Method Inheritance Factor (MIF) metric [Abreu(1995)].

Need
DPIC measures the amount of inheritance calculated from a superclass (single/hierarchical inheritance). As the concept always involves the two entities super and subclass, it is worth finding a way to measure inheritance factor of subclass that actually utilizes the inheritance concept (single/multiple inheritance). DMIF can be used to measure the percentage utilization of inheritance by a subclass taken over the total number of methods that could be accessed using that class at runtime.

MIF is a static metric that evaluates the percentage of inherited methods to the total number of methods of a subclass. DMIF modifies MIF to find out the percentage of inherited methods that are actually invoked at runtime instead of just computing the percentage of total number of inherited methods. MIF is a system-level inheritance measure whereas DMIF is defined for both class-level and system-level inheritance measurement.
Metric Effect

A 0% value would indicate that none of the inherited methods were used at runtime and hence there is no inheritance supported by the class. Thus system module containing the class may require redesigning.

A 100% value would mean that there are no direct method calls for the class at runtime i.e. all the calls are for inherited methods at runtime and hence, the class shows a high degree of inheritance. The value of DMIF should always be less than or equal to corresponding MIF value.

If the value of DMIF (System) is low, it indicates an overall low amount of inheritance supported by the system at runtime.

- **Maintainability** may be inversely related to DMIF. A high DMIF value would mean that out of the total methods that can be invoked in reference to the subclass, used inherited methods have a high percentage, and hence more dependency on the superclasses. Thus any modifications in the superclasses would require modifications in the subclass also, making it tough to maintain.

- **Testability** may also be inversely related to DMIF metric. This is because a subclass with a high DMIF value has a good chance of accessing the inherited methods of higher number of superclasses at runtime. Hence to test the subclass, all those superclasses would need to be tested, decreasing its testability.

- **Reusability** may decrease with increasing DMIF. More the use of inherited methods at runtime, stronger the inheritance between the subclass and its superclasses, more the dependence on superclasses and, hence lower the reusability of the class.

- **Portability** may decrease with an increase of DMIF as a high DMIF value would mean more inheritance-based couples with superclasses i.e. more dependency on superclasses and, hence low portability.

c) **Dynamic Method Inheritance Factor – Variation1 (DMIF1)**

**Definition**

DMIF1 is the percentage of number of inherited methods of a subclass used at runtime to total number of inherited methods of the class. It can be useful in measuring single and multiple inheritances for a class at runtime.
Chapter 3 – Proposed Dynamic Metric Suite

\[
DMIF1(C) = \frac{\sum_{i=1}^{TSp} M_{inu}(S_i(C))}{\sum_{i=1}^{TSp} M_{in}(S_i(C))} \times 100 \quad (Equation 9)
\]

\(TSp\) = Number of super classes of a class \(C\).
\(S_i(C)\) = \(i^{th}\) superclass of \(C\).
\(M_{inu}(S_i(C))\) = Number of inherited methods, from \(i^{th}\) superclass of \(C\), used at runtime.
\(M_{in}(S_i(C))\) = Number of methods inherited from its \(i^{th}\) superclass by \(C\).

This definition of DMIF can be extended for the whole system as:

\[
DMIF1(System) = \frac{\sum_{j=1}^{TC} \sum_{i=1}^{TSp} M_{inu}(S_i(C_j))}{\sum_{j=1}^{TC} \sum_{i=1}^{TSp} M_{in}(S_i(C_j))} \times 100 \quad (Equation 10)
\]

**Literary Basis**

This metric uses the MOOD set’s Method Inheritance Factor (MIF) metric [Abreu(1995)].

**Need**

DPIC measures the amount of inheritance calculated from a superclass (single/hierarchical inheritance). As the concept always involves the two entities super and subclass, it is worth finding a way to measure inheritance factor of subclass that actually utilizes the inheritance concept (single/multiple inheritance).

MIF metric measures the percentage of inherited methods to the total number of methods of a subclass. As the dynamic metrics provide the actual behavior of an object class at runtime, MIF evaluated at runtime will definitely add useful information to the static MIF. DMIF1 is derived from MIF metric as it also depends upon the number of inherited methods of the subclass but only those used at runtime. The DMIF1 can be highly beneficial as it finds the percentage of number of used inherited methods to the total number of inherited methods (subclass methods and inherited methods). This makes DMIF1 a combination of a dynamic value (numerator) taken over a static value (denominator). Thus definition also helps in comparing the static and dynamic MIF values.

**Metric Effect**

This metric shows the actual effect of inheritance on a subclass at runtime compared to that before runtime. A lower value indicates that actual inheritance is lower at runtime. As the values increases the inheritance effect on the subclass also increases.
A 100% value will indicate the full usage of inherited methods at runtime. It in turn indicates a healthy use of inheritance feature at execution time. It further shows the strong coupling between the super class and the sub class.

On the other hand, a 0% value would indicate that the sub class perhaps is needed to be redesigned. It also indicates that the sub class need not extend the super class as it is not accessing any of the inherited methods at runtime.

- **Maintainability** is expected to decrease with an increasing DMIF1. This is because a high DMIF1 indicates higher number of inherited methods used out of the total number of inherited methods. Any change in the any of the superclasses would require modification in the subclass as well as the client (/source) classes of method calls. Thus the concerned subclass will be strongly coupled to its superclasses, as well as the client classes that are the sources of those method calls (if in case subclass is not the client class itself), the overall coupling increases, hence maintainability of the class falls.

- **Testability** may be low for a high DMIF1 value. As more number of inherited methods are invoked from the client class (via the concerned subclass) being tested, more testing effort would needed to be spent on server subclass and the server superclasses involved.

- **Reusability** is expected to be inversely related to DMIF1. Higher the use of inherited methods at runtime, stronger the inheritance between the subclass and its superclasses. Thus the subclass has lower independent reusability.

- **Portability** could be inversely related to DMIF1 because of the strong inheritance coupling between subclass and its superclasses.

d) **Dynamic Method Inheritance Factor - Variation2 (DMIF2)**

**Definition**

DMIF2 is the percentage of number of inherited methods of a sub class invoked at runtime to the total number of methods (including the inherited methods) of a subclass invoked at runtime. It measures single and multiple inheritance for a class at runtime.

\[
DMIF2(C) = \frac{\sum_{i=1}^{TSp} M_{\text{inv}}(S_i(C))}{M_{\text{dn}}(C) + \sum_{i=1}^{TSp} M_{\text{inv}}(S_i(C))} \times 100
\]

(Equation 11)

TSp = Number of super classes of a class C.
$S_i(C) = \text{ith superclass of } C$

$M_{\text{inu}}(S_i(C)) = \text{Number of inherited methods, from } i^{\text{th}} \text{ superclass of } C, \text{ used at runtime.}$

$M_{\text{du}}(C) = \text{Number of direct methods of } C, \text{ used/invoked at runtime.}$

This definition of DMIF can be extended for the whole system as:

$$\text{DMIF2(System)} = \frac{\sum_{j=1}^{TC} \sum_{i=1}^{TSp} M_{\text{inu}}(S_i(C_j))}{\sum_{j=1}^{TC} \left( M_{\text{du}}(C_j) + \sum_{i=1}^{TSp} M_{\text{inu}}(S_i(C_j)) \right)} \times 100 \quad (\text{Equation 12})$$

**Literary Basis**

This metric is the second variation of Method Inheritance Factor (MIF) metric which is the part of the famous MOOD metric suite given by Abreu(1995).

**Need**

DMIF1 has a dynamic value in the numerator and a static value in denominator. DMIF2 finds the percentage of number of inherited methods used at runtime to the total number of used methods of a subclass (including inherited methods). Thus the denominator of DMIF2 would be less than or equal to that of DMIF1 i.e. DMIF2 $\geq$ DMIF1. Thus DMIF2 is a pure runtime metric whereas DMIF1 takes a static value into account.

**Metric Effect**

It gives a measure of amount of inheritance at runtime. A high DMIF2 value will indicate a high level of inheritance effect possessed by a subclass. It will indicate a high amount of indirect accesses to the super class.

A 100% value will mean that the subclass does not use any of its own methods at runtime and hence should be considered for redesigning. This would also infer that in this situation DMIF1 will be equal to DMIF2. A low value indicates a weak linkage between the class and its super class at runtime. A 0% value indicates that the class can do without extending the super class.

The dynamic and static values of the metric can vary as there can be an unpredictable difference between the static method calls and the actual method calls.

- *Maintainability* may be inversely related to DMIF2. A high DMIF2 value would mean that out of the total methods used at runtime, inherited methods have a high percentage, and hence more dependency on the superclasses. Thus any
modifications in the superclasses would require modifications in the subclass also, making it tough to maintain.

- **Testability** may decrease with an increase in DMIF2 value. This is because a subclass with a high DMIF2 value would depend upon more number of superclasses at runtime. Hence to test the subclass, all those superclasses would need to be tested, decreasing its testability.

- **Reusability** is expected decrease with increasing DMIF2. More the use of inherited methods at runtime, stronger the inheritance between the subclass and its superclasses and hence lower the reusability of the class.

- **Portability** may be inversely related to DMIF2 as a high DMIF2 value would mean more inheritance-based couples with superclasses and hence low portability.

e) **Dynamic Effective Number of Children (DENOC)**

**Definition**

DENOC is the percentage of number of immediate specializations used to send method calls to the class at runtime to the total number of immediate specializations. The types of inheritance covered are single and hierarchical.

\[
\text{DENOC}(C) = \frac{N_R(C)}{N(C)} \times 100
\]

(Equation 13)

N(C) = Number of immediate specializations of class C.
N_R(C) = Number of immediate specializations of C whose inherited methods are invoked at runtime.

**Literary Basis**

This metric is a modification to NOC (Number of Children) metric given by Chidamber and Kemerer(1994).

**Need**

NOC metric is a static metric that calculates the number of immediate specializations of a class. DENOC finds the actual number of immediate specializations used at runtime and finds its percentage over their NOC count. Thus this metric adds more information to NOC metric. DENOC excludes the specializations that are not used at runtime from the definition of NOC and thus finds the correct percentage of useful immediate
specializations. It is important to state that NOC makes a part of DENOC’s definition thus it adds the static information to DENOC.

**Metric Effect**

It gives an idea of the potential influence a class has on the design. A 0% value would mean no utilization of inherited methods by the subclasses, hence indicating a need for system/class redesign.

A 100% value indicates that all the subclasses of a class are used at runtime. Hence all the subclasses utilize inheritance coupling at runtime.

It gives the actual number of children used at runtime that can be less than or equal to the total number of children of a particular class. This deduces that the above factors like reuse etc. could be lower for dynamic metric values than static metric values.

- **Maintainability** may be inversely related to DENOC. A high value would indicate a high percentage of active subclasses at runtime i.e. classes depending upon the methods inherited from the superclass. So any modifications to the superclass would affect the design of subclasses. Thus it is tough to maintain.

- **Testability** is expected to decrease with an increasing DENOC. If a class has a large number of children and all are being used at runtime then it may require more intense testing of the methods of the class.

- **Reusability** may have a direct relation with DENOC as inheritance promotes the reuse of subclass methods. Thus a class with a high DENOC value would have high reusability.

- **Portability** may be inversely related to DENOC as a class with a high DENOC value would mean more number of subclasses used to send messages to a class, hence making the class more linked to its subclasses at runtime.

### 3.2.3 Additional Metrics

**a) Dynamic Percentage Constructor Calls Received (DPCCR)**

**Definition**

DPCCR is the percentage of number of calls received by the constructors of a class to the total calls received by all the methods of a class (including constructor methods) at runtime.
Design and Validation of Dynamic Metrics for Object-Oriented Software Systems

\[ \text{DPCCR}(C) = \frac{N_c}{N_m + N_c} \times 100 \]  
*(Equation 14)*

\( N_c \) = Number of calls received by the constructors of class C.
\( N_m \) = Number of calls received by all methods of class C excluding constructors.

**Need**

DPCCR measures the amount of *automatic initialization* as well as usage of objects of classes at runtime by evaluating the extent of constructor calls among all the calls received by the class at runtime. DPCCR helps in finding out the amount of code that is automatically executed at the time of object creation.

**Metric Effect**

This metric can be very useful in finding out the extent to which various classes are automatically initialized during runtime. So it finds the amount of job that is automatically executed on creation of an object at runtime.

A high value would suggest that the class demonstrates a highly automated nature and serves fewer method calls apart from the constructor calls at runtime. This also indicates that the objects of the class may have a shorter life time. This may also indicate a high percentage of constructors among the methods of the class. If this value is high for a class having a low percentage of constructors, this may again point to the underuse of methods by live objects of the class at runtime.

**b) Method Request For Service (MQFS)**

**Definition**

MQFS is the number of calls sent by a method to other methods.

\[
\text{MQFS} (m_k) = \sum_{j=1}^{P} \sum_{l=1}^{N} | M(m_k(c_x), m_l(o_j)) | o_j \rightarrow c_y \land o_j \in O \land c_x, c_y \in C \land c_x \neq c_y |
\]

*(Equation 15)*

Here,

\( M(m_k(c_x), m_l(o_j)) \) = set of messages sent from \( k^{th} \) method \( m_k \) of class \( c_x \) to \( l^{th} \) method \( m_l \) of \( j^{th} \) object \( o_j \) of class \( c_y \).

Because MQFS is defined at *method-level*, it considers all the methods including those exhibiting polymorphism. But it eliminates duplicate method calls from one particular method \( m_k \) to another method \( m_l \).
Chapter 3 – Proposed Dynamic Metric Suite

**Literary Basis**

The metric works on the same lines as Class Request For Service (CQFS) [Zaidman and Demeyer(2004)] at method level.

**Need**

Object Request For Service (OQFS) metric [Yacoub et al.(1999)] counts the total number of (unique) messages that object $o_i$ has sent during the program run. Thus OQFS can be defined in the form of EOC metric [Yacoub et al.(1999)] as:

$$\text{OQFS}(o_i) = \sum_{j=1}^{p} \text{EOC}(o_i, o_j)$$  \hspace{1cm} (Equation 16)

It was also concluded that in the case of program understanding considering the class-level coupling instead of object-level coupling is more intuitive for the end user and because of the direct relation between the results from the heuristic and the actual source code. This idea resulted into Class Request For Service (CQFS) metric defined as:

$$\text{CQFS}(c_k) = \sum_{j=1}^{p} \{M(m_k(o_i), m_l(o_j))\} | o_j \rightarrow c_x \land o_j \rightarrow c_y \land o_i, o_j \in O \land c_x, c_y \in C \land c_x \neq c_y$$  \hspace{1cm} (Equation 17)

where,

$M(m_k(o_i), m_l(o_j)) = \text{set of messages sent from } k^{th} \text{ method } (m_k) \text{ of } i^{th} \text{ object } (o_i) \text{ of class } c_x \text{ to } l^{th} \text{ method } (m_l) \text{ of } j^{th} \text{ object } (o_j) \text{ of class } c_y$.

CQFS counts all the methods that a class calls during runtime. MQFS metric goes deeper into analyzing the sources of request by finding the methods called by each method outside its own class. So it adds more detail to CQFS metric as it finds CQFS at method-level.

**Metric Effect**

It demonstrates the extent to which a method contributes towards the efferent coupling [Martin(1994)] of a class. Also a high value indicates more coupled method and class in turn.

A high value indicates a high inter-method and inter-class coupling. A class carrying methods having high MQFS values has a high probability of having a high CQFS value and in turn high outgoing coupling.
c) Most Active Method (MAM)

**Definition**
It is defined as the sum of number of messages sent to a method and number of messages sent by that method to other methods at runtime.

**Literary Basis**
It is based on (Dynamic Key Class) DKC metric at method-level.

**Need**
DKC metric is a class-level metric that finds the *most active class* at runtime by counting the number of times a class acts as a client or a server. MAM goes deeper into this idea by finding the *most active method* in the system at runtime using the same concept. As method is always the key functioning module of the system, it is also the ultimate source or the destination of a method call. There are no dynamic method-level metrics known till date to the knowledge of this research.

**Metric Effect**
Most Active Method gives the method that is active for the most number of calls during runtime for the whole system. It gives the centrifugal point in the system where the most of

<table>
<thead>
<tr>
<th>Metric</th>
<th>Maintainability</th>
<th>Testability</th>
<th>Reusability</th>
<th>Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Coupling Metrics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCa</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>DKSC</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>DKCC</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>DKC</td>
<td>Low</td>
<td>Low</td>
<td>Low/High</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Inheritance Metrics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPIC</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>DMIF</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>DMIF1</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>DMIF2</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>DENOC</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Additional Metrics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPCCR</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>MQFS</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>MAM</td>
<td>Low</td>
<td>Low</td>
<td>Low/High</td>
<td>Low</td>
</tr>
</tbody>
</table>

Table 12: Metric effect on external quality attributes
the processing is being done.

Further, it may or may not be that the most active method belongs to the most active class i.e. Dynamic Key Class before runtime or at runtime. As the method calls received by the dynamic key class may be distributed widely among all the methods thus reducing the chances of letting a method become the most active method.

Table 12 summarizes the effects of all the proposed dynamic metrics on four major external quality attributes of maintainability, testability, reusability and portability. The values are interpreted in ordinal form. All the quality attribute values are shown in accordance with corresponding high metric values.

Chapter Summary

This chapter introduces all the proposed dynamic object-oriented metrics proposed in this research work. The reasons behind selecting the coupling and inheritance object-oriented attributes are discussed. It is followed by the metric definition template used to define the new metrics. New metric suite is divided into two parts: dynamic coupling and dynamic inheritance metrics. Each metric is introduced with the help of a definition, literary basis of the metric and the metric’s effect on the external quality attributes like maintenance, testability, portability and reusability. There are four dynamic class-level coupling metrics and five dynamic class-level inheritance metrics proposed in this work. There are three additional dynamic metrics that contain two method-level dynamic coupling metrics and one constructor-based automatic initialization metric. The chapter ends with the introduction of three additional proposed metrics. The next chapter explains the methodology followed for statistically validating these metrics.