2.1 Software Metrics - Concept

Software metrics are one of the most important tools used in software engineering in order to enhance the quality of software. Software metrics can be loosely defined as activities concerned with measurement in software engineering [Fenton and Neil(1999)]. Goodman(1993) has defined software metrics as: The continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products. Software metrics can be used throughout a software development life cycle to assist in estimation, quality control, productivity assessment, and project control [Wakil and Fahmy(2004)]. Figure 2 demonstrates the software metric concept in which measurement-based techniques are applied to processes, products and services to supply useful engineering information. This information is further used to improve the processes, products and services. It also emphasizes that software metrics provide the information needed by engineers for technical decisions [Westfall(2001)].

Figure 2: Software metric – Concept [Westfall(2001)]
Software metrics can be used to aid industrial management in [Chidamber and Kemerer(1991)]:

- Estimating the cost and schedule of future products.
- Evaluating the productivity impacts of new tools and techniques.
- Establishing productivity trends over time.
- Improving Software Quality.
- Forecasting future staffing needs.
- Anticipating and reducing future maintenance requirements.
- Evaluating the object-orientation of an implementation.
- Setting design standards for an organization.

2.2 Measurement, Metric and Software Quality

Fundamental measurement is a means by which numbers can be assigned according to natural laws to represent the property and yet which doesn’t presuppose measurement of any other variables [Torgerson(1958)]. Measurement is the process of empirical and objective assignment of numbers to properties of objects or events of the real world in such a way to describe them [Finkelstein(1982)].

IEEE Standard 1061-1998 [IS] defines a quality factor as “a management-oriented attribute of a software that contributes to its quality”. A metric is a measurement function, and a software quality metric is a “function whose inputs are software data and whose output is a single numeric value that can be interpreted as a degree to which software possesses a given attribute that affects its quality”.

2.2.1 Measurement Scales for Software Metrics

Every bit of software metric data should be collected with a specific goal. The general goal is to use the data in some calculation work related to statistical analysis of the process model. It is important to consider the type of information involved before the collection of data. Six types of measured data have been recognized by statisticians (See Table 1). These are nominal, typological, ordinal, interval, ratio and absolute [MS,Pandian(2003)]. All the six types of measurement scales are described briefly below:

a) Nominal

Measurement on a nominal scale employs semantic expressions to represent objects for the
<table>
<thead>
<tr>
<th>Scale</th>
<th>Description</th>
<th>Appropriate Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linguistic Scales</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nominal</td>
<td>Identification code distinguishing objects</td>
<td>Type, Frequency, Mode, Distribution</td>
</tr>
<tr>
<td>Typological</td>
<td>Categories distinguishing type of objects</td>
<td>Type, Frequency, Mode, Distribution</td>
</tr>
<tr>
<td><strong>Linguistic + Numerical</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ordinal</td>
<td>Rankings ordering objects</td>
<td>Percentile, Median, Rank Order Correlation</td>
</tr>
<tr>
<td><strong>Numerical</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interval</td>
<td>Differences in magnitude, No Comparison possible</td>
<td>Mean, Standard deviation, Correlation - r, Regression, Analysis of variance, Factor analysis</td>
</tr>
<tr>
<td>Ratio</td>
<td>Ratio between values, have a non-arbitrary absolute zero.</td>
<td>Geometric Mean, Harmonic Mean, standard deviation, Correlation - r, Regression, Factor analysis, Coefficient of variation</td>
</tr>
<tr>
<td>Absolute</td>
<td>Comparison with absolute normative value</td>
<td>No restrictions due to the scale type.</td>
</tr>
</tbody>
</table>

**Table 1: Software metrics – Measurement scales [MS, Pandian(2003)]**

purpose of identification (referential value). An example is to assign the ID numbers to the defects found in a code.

b) **Typological**
Measurement on a typological scale identifies types or categories in entities that have been already recognized and named. An example is classifying defects according to a well-defined framework like orthogonal defect classification.

c) **Ordinal**
Measurement on an ordinal scale assesses values in measured entities and rearranging them according to the order of value. Both value and order are expressed using words or symbols. For example, when the defects are measured, their severity levels are described in semantic expressions such as high, medium and low.
d) Interval
Interval Scale marks equal distances from one point to another. It is an arbitrary scale, used for perceiving increments, not ratios. For example, temperature (in degrees centigrade), and calendar time.

e) Ratio
A ratio scale introduces proportion. It is a more potential scale, which permits ratio calculation and equipped with rational zero reference point. For example, while 30 degrees centigrade is not twice as hot as 15 degrees centigrade, 30 degrees absolute (Kelvin) is twice as hot as 15 degrees absolute. Similarly for length, 20 centimeters is twice as long as 10 centimeters.

f) Absolute
Absolute scales are entities that are simply counted. They are unique and unambiguous. They have all the properties of ratio scales. For example, counting lines of code.

2.2.2 Software Metrics and Quality
The true value of product metrics comes from their association with measures of important external quality attributes [ISO]. An external attribute is measured with respect to how the product relates to its environment [Fenton(1991)]. Examples of external attributes are testability, reliability, maintainability etc. Practitioners, whether they are developers, managers, or quality assurance personnel, are always concerned with the external attributes. However, they cannot measure many of the external attributes directly until quite late in a project’s or even a product’s life cycle. Therefore, they can use product metrics as leading indicators of the external attributes that are important to them.

It is worthwhile mentioning here that the external quality attributes get their values from internal quality attributes of the software product. Internal quality attributes are those which can be measured directly in terms of software product and they will remain measurable during and after the creation of the software product. Thus internal quality attributes are direct metrics as they are engaged in direct measurement of software complexity. Internal attributes are of no use if used alone. To provide meaning to these measures, they must be characterized in terms of external quality attributes of a software product such as maintainability, reusability etc (See Figure 3). Software metrics can be used in at least three ways: making system level predictions, early identification of high
risk software components, and the construction of preventative design and programming guidelines. These uses allow an organization to have an early estimate of quality, in order to take timely action to reduce the number of faulty software components. Large effort has been spent by the software engineering research community in designing and developing software metrics for object-oriented systems, and empirically establishing their relationship to important external attributes. The latter process is called the *empirical validation* of a software metric. A metric that has been proved to be empirically valid in a number of different contexts and applications, could be used by the organizations in order to use them to create appropriate prediction models and guidelines customized to their own context.

### 2.3 Types of Software Metrics

For better understanding of software metrics, their classification by some meaningful categories is important. Practitioners and Researchers have described many independent categories for software metrics each demonstrating a unique viewpoint. Even though there are many classifications of software metrics, they were never grouped into one comprehensive grouping [Wakil and Fahmy(2004)]. Wakil and Fahmy(2004) conducted a thorough survey of the categories of metrics available till date and organized them into taxonomy framework. Such taxonomy is expected to help researchers and practitioners to better understand and select appropriate metrics for their application purposes.

---

**Figure 3: Software quality model**
2.3.1 Product, Process, Resources and Hybrid Metrics

Fenton and Pfleeger (1996) classified software metrics into three main categories: product, process and project metrics described below:

- **Process metrics** are used to measure attributes related to the software development life cycle processes. The most significant process attributes are time, cost and effort that could be estimated by Boehm’s constructive cost model (COCOMO) [Boehm et al. (2000)].

- **Product metrics** describe characteristics of the software development life cycle processes outputs such as requirement specifications documents, design diagrams, source code listings and the executable software. For example, requirement specification document’s readability is assessed using Flesch-Kincaid Grade Level index [RF], source code size is measured using Lines of Code metric (LOC) [Conte et al. (1986)].

![Figure 4: Software metrics – Taxonomy [Wakil and Fahmy (2004)]](image-url)
• **Resource metrics** describe the characteristics of available resources. For example, number of developers and their skills, and hardware reliability and performance. These types of metrics are most beneficial for owners and project managers. Resource metrics are also called project metrics.

• **Hybrid metrics** [Abreu(1992)] are combination of product and process metrics. Examples are: N cost per function point, and time to deliver per LOC.

### 2.3.2 Internal and External metrics

Product metrics are divided into internal and external metrics as follows:

• **Internal metrics** measure factors visible only to the software development team. For example, size metrics (e.g. Lines of Code (LOC) [Conte *et al.* (1986)], number of modules [Gilb(1977)] and unique subjects count [Hyatt *et al.* (2004)]) and Complexity metrics (e.g. Cyclomatic Complexity [McCabe(1976)] and Halstead Complexity [Halstead(1977)]).

• **External metrics** measure factors that are visible to the users of a software product. For example, maintainability, reliability, functionality, reusability etc. External metrics are harder to measure as compared to the internal metrics since they are of subjective nature. Also the external metrics are not available till the late stages of the software development life cycle. Hence internal metrics are used to estimate external metrics at the early stages of the development process.

### 2.3.3 Specification, Design and Code Metrics

Internal product metrics could be further categorized according to the software artifact under consideration such as specification documents, design diagrams, and code listings as defined below:

• **Specification metrics** analyze the product requirement specifications and provide an early feedback about the developing software product. For instance, count of weak phrases, lines of text, and unique subjects [Hyatt *et al.* (2004)].

• **Design metrics** are measured at a later stage of the software development process. They can help to refine the software design in order to avoid already known problems with the final product even before program code is written. Examples of design metrics are Henry and Kafura's information flow [Henry and Kafura(1981)], reuse ratio [Xenos *et al.* (2000)] and coupling factor [Abreu(1995)].
- Code metrics are metrics that measure the features of the program code. For example, Lines of Code (LOC) metric [Conte et al. (1986)], Cyclomatic complexity [McCabe (1976)] and Knot metric [Shepperd and Ince (1993)].

2.3.4 Procedural and Object-Oriented metrics

On the basis of programming paradigm they use, design and code metrics can be classified into procedural and object-oriented metrics as described below:

- Procedural metrics measure the properties concerned to software developed in procedural programming languages. They are organized around a concept in which a software has individual procedure or subprogram as the most significant unit. Procedural metrics include Lines Of Code (LOC) [Conte et al. (1986)], Cyclomatic Complexity [McCabe (1976)] etc.

- Object-Oriented Metrics are affected by the use of object-oriented mechanisms such as inheritance, abstraction, encapsulation and polymorphism. Examples of object-oriented metrics include Weighted Methods per Class (WMC), Number of Children (NOC), and Response for a Class (RFC) [Chidamber and Kemerer (1994)].

Object-oriented metrics could be further categorized according to the object-oriented property they measure. For example, there are metrics for measuring size such as number of methods, number of attributes and number of classes. There are metrics for measuring coupling such as direct class coupling, number of dependencies, and coupling factor. There are metrics for measuring inheritance, cohesion, and encapsulation.

2.3.5 Intra-class and Inter-class metrics

There are two sub-categories of object-oriented design metrics: Intra-class and inter-class metrics.

- Intra-class metrics measure characteristics related to one class. Examples are Weighted Methods per Class (WMC), Number of Children (NOC), and Depth of Inheritance Tree (DIT) [Chidamber and Kemerer (1994)].

- Inter class metrics measure features between a set of classes. Examples are Coupling Factor (CF) and Method Hiding Factor (MHF) [Abreu (1995)].
2.3.6 Elementary and Composite metrics

- Elementary metrics are evaluated directly from product artifacts. For example, Number of Methods in a Class and, Complexity of a method.

- Composite metrics are evaluated using mathematical combination of elementary metrics. For example, Weighted Methods per Class (WMC) [Chidamber and Kemerer(1994)].

2.3.7 Static and Dynamic metrics

- Static metrics are collected from the static artifacts of the software such as specification documents, design diagrams and code listings. Examples of static metrics include Lines of Code (LOC) [Conte et al.(1986)], Weighted Methods per Class (WMC) and Coupling Between Objects (CBO) [Chidamber and Kemerer(1994)].

- Dynamic metrics are evaluated using data collected from the actual execution of a software. For example, Dynamic Coupling [Mitchell and Power(2004a)], Dynamic Lack of Cohesion [Mitchell and Power(2004b)].

Figure 4 shows the software metrics taxonomy proposed by Wakil and Fahmy(2004). The shaded blocks in the diagram represent the type of metrics explored in this work. This research deals in internal static and dynamic product metrics for object-oriented systems. These two types of metrics are briefly defined below:

a. Object-Oriented Static Metrics: These are the design metrics for object-oriented software that are based on information gathered from static analysis. Chapter 3 discusses this type of metrics including the related work about them.

b. Object-Oriented Dynamic/Runtime Metrics: These metrics are evaluated from the related information extracted at runtime. Chapter 4 discusses this type of metrics including the related work about them.

2.4 Defining and Validating a Software Metric

2.4.1 Defining a Software Metric

To extract useful information from a metric, every person involved in designing, implementing, collecting data for and utilizing a software metrics must understand its definition and purpose. There are seven steps involved in designing a software metric [Westfall(2001)].
Step 1 - Objective Statement: Software metrics can execute one of the following four tasks.

- Metrics can help in understanding more about software products, processes and services.
- Metrics can be used to evaluate software products, processes and services against established standards and goals.
- Metrics can provide the information needed to control resources and processes used to produce software.
- Metrics can be used to predict attributes of software entities in the future [Humphrey(1989)].

The objective for each metric can be formally defined in terms of one of these functions, the attribute of the entity being measured and the goal for the measurement.

Step 2 - Clear Definitions: The next step is to agree to a standard definition for the entities and their attributes being measured. When the terms like defect, problem report, size and even project are used, users would interpret these words in their own context with meanings that may differ from our intended definition. These interpretation differences increase further when more ambiguous terms like quality, maintainability and reusability are used.

The approach must be to adopt standard definitions within an organization and then apply them consistently. Suggestions from the industry could be used as a foundation to get started. Pick and choose the definitions that match with the organizational objectives or use them as a basis for creating a new definition.

Step 3 - Define the Model: The next step is to derive a model for the metric. In simple terms, the model defines how a metric would be calculated. There are two methods for selecting a model. In many cases there is no need to "re-invent the wheel". There are many software metrics models that have been used successfully by other organizations. These can be found documented in the current literature and in proprietary products that can be purchased. One can utilize these models with little or no adaptation to match a given environment. The second method is to create a new model. The best advice here is to talk to the people who are actually responsible for the product or resource or who are involved in the process. They know about the importance of various factors.
Step 4 - Establish Counting Criteria: The next step in designing a metric is to break the model down into its lowest level metric primitives and define the counting criteria used to measure each primitive. This defines the mapping system for the measurement of each metric primitive.

Step 5 - Decide What’s "Good": The fourth step in designing a metric is defining what’s "Good". Once it is decided what to measure and how to measure it, it has to be decided what to do with the results. Is 10 too few or 100 too many? Should the trend be up or down? Once it is decided "what’s good", it can also be determined whether or not action is needed. If the desired values are being met or exceeded, no corrective action is necessary. Management can either turn its attention elsewhere or establish some maintenance level actions to insure that the value stays at acceptable levels.

Step 6 - Metrics Reporting: The next step is to decide how to report the metric. This includes defining the report format, data extraction and reporting cycle, reporting mechanisms and distribution and availability.

Step 7 - Additional Qualifiers: The final step in designing a metric is determining the additional metric qualifiers. A good metric is a generic metric. That means that the metric is valid for an entire hierarchy of additional extraction qualifiers. The additional qualifiers provide the demographic information needed for these various views of the metric. The main reason that the additional qualifiers need to be defined as part of the metrics design is that they determine the second level of data collection requirements.

2.4.2 Validating a Metric [Fenton(1994)]

A software metric is useful only if it is validated both theoretically and practically. The evaluation of the ‘validity of a metric’ is a complex task as it involves in-depth theoretical analysis and empirical investigation [Kitchenham et al.(1995)]. A metric is theoretically valid if it measures the right attribute and possesses desirable formal (or axiomatic) properties [Fenton and Pfleeger(1996)]. Two approaches for validation have been prescribed and/or practiced in software engineering:

a) **Theoretical validation** [Fenton and Kitchenham(1990)]: A measure is really measuring the attribute it is purporting to measure.

b) **Empirical validation** [Schneidewind(1992)]: A measure is useful in the sense that it is related to other variables in expected ways.
There are also a number of criteria that determines a metric’s practical validity. These are mentioned below [Daskalantoanis(1992)].

- **Robustness**: Responsiveness to modifications.
- **Normativeness**: Availability of a datum.
- **Specificity**: Influence of attributes other than that measured.
- **Prescriptiveness**: Ability to prescribe methods for improvement.
- **Understandability**.

In order to add to the validity of metrics it is necessary to have a formal set of criteria with which to evaluate the proposed metrics [Chidamber and Kemerer(1994)]. Many researchers have proposed properties that they feel a software metric must possess. One such example is Weyuker(1998) who has developed a formal list of desiderata for software metrics, which has been used to validate the metric suite proposed by Chidamber and Kemerer(1994). It defines the following behavioral properties for a metric under practical validation:

**Property 1: Non-Coarseness**
This implies that every class/object cannot have the same value for a metric, otherwise it loses its value for the measurement.

**Property 2: Non-Uniqueness**
This implies that two classes/objects can have the same metric value, that they both can be equally complex.

**Property 3: Permutation is significant**
Given two classes/objects A and B such that A is a permutation of B, i.e. elements of A are simply a different ordering of elements of B, this does not imply that the metric value of A is equal to that of B.

**Property 4: Implementation not function is important.**
Given two class/object designs which perform the same function does not imply that the metric values for each are the same. The implementation details matter when determining the design metric.

**Property 5: Monotonicity**
This implies that the metric value for classes/objects is minimally zero and therefore the combination of two classes/objects can never be less than the metric value of either of the components.
Property 6: Non-equivalence of interaction

Given three classes/objects A, B and C. If the metric value for A is equivalent to metric value for B, this does not imply that the metric value for A+C is equivalent to the metric value for B+C.

Property 7: Interaction increases complexity

The idea behind this property is that when two classes/objects are combined, the interaction can increase the complexity metric value.

2.5 Object-Oriented Design

Structured/Procedural systems are based on operations/functions and data does not depend upon operations in such systems. But data is closely related to operations in real-world problems as it defines the state of a real world object whereas operations define the behavior of that object. Object-Oriented systems use this concept in solving the given problem. As object-oriented paradigm is more closely related to the real world situations, they have significantly replaced the structured paradigm in industry in past two decades.

Object-Oriented Design is defined as [Booch(1994)] Object-oriented design is a method of design encompassing the process of object-oriented decomposing and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

Classes and objects are the basic units of object-oriented design. State and behavior are the main characteristics of any object that are defined by the variables and methods in the design. Objects interact and communicate with each other by sending messages to each other. A class is an object-oriented entity in which state and behavior are bundled together for all the objects that have common behaviors. “A class represents a template for several objects and describe how these objects are structured internally. Objects of the same class have the same definition both for their operation and for their information structure” [Jacobson et al.(1992)]. A class is a blueprint that defines the variables and the methods common to all objects of a certain kind [Campione and Walrath(1996)]. It contains the implementation statements for an object that are hidden from other classes of the software system, thus demonstrating the key object-oriented concepts such as information hiding and abstraction.
Object-oriented design is related to developing an object-oriented module of a software system to serve the identified and analyzed requirements for the system. Various advantages/features of an object-oriented design include:

- improves design quality
- high reusability features
- faster development process
- module-based architecture

Figure 5: Object-oriented methodology [Booch(1994)]

Booch(1994) introduced a methodology, given in Figure 5, which shows the necessary features of an object-oriented design. It comprises of a four-step process to design an object-oriented system.

i. **Identify Classes and Objects:** Key abstractions in the problem space are identified and labeled as potential classes and objects.

ii. **Identify of semantics of Classes and Objects:** The meanings of the previously identified classes and objects are established, including defining the life cycle of each object from creation to destruction.

iii. **Identify of Class-Object Relationship:** Class and object interaction is identified, for example patterns of inheritance and patterns of visibility among objects and classes.
iv. *Specify Class and Object Interfaces and Implement Classes and Objects:* Detailed internal view is constructed. This includes defining methods and their various behaviors.

### 2.5.1 Object-Oriented Internal Design Attributes

There are a number of vital concepts/attributes in object-oriented design that are used primarily to support object-oriented design for measurement tasks.

a) **Coupling**

Coupling is defined as the relationship or interdependency between the modules of a software system. For example, object A is coupled to object B if and only if A sends a message to B. Thus coupling refers to the number of messages passed between objects.

b) **Cohesion**

“Cohesion measures the degree of connectivity among the elements of a single class or object” [Booch(1994)]. Cohesion relates to internal bonding among the various parts of a software design. Cohesion measures encapsulation within an object. It deals with method-data interaction inside an object that makes it internally bonded. A class is said to be cohesive when the methods and variables contained are highly correlated. Cohesion can be used to identify the badly designed classes.

c) **Inheritance**

Inheritance is a mechanism in which one object acquires characteristics from one, or more other objects. “Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship” [Rumbaugh et al.(1991)]. Inheritance occurs at all levels of a class hierarchy. It is used to construct relationships between superclasses and subclasses in various ways.

d) **Encapsulation**

Encapsulation is an indirect measure of data abstraction and information hiding. Encapsulation hides internal specification of an object and show only external interface. *The process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation* [Booch(1994)]. Encapsulation influences metrics by changing the focus of measurement from a single module to a package of data.
e) Information Hiding

Information hiding is the process of hiding all the secrets of an object that do not contribute to its essential characteristics [Booch(1994)]. Public interface and a private implementation of an object are kept distinct. All information about a module should be private to the module unless it is specifically declared public [Meyer(1998)]. Information hiding plays a strong role in such metrics as object coupling and the degree of information hiding.

f) Localization

Localization is the process of gathering and placing things in close physical proximity to each other [Booch(1994)]. It is based on objects in case of object-oriented design. A design plan is totally dependent upon the localization approach, because one function may involve several objects, and one object may provide many functions. Metrics should apply to the class as a complete entity. Even the relationship between functions and classes is not necessarily one-to-one. For that reason, metrics that reflect the manner in which classes collaborate must be capable of accommodating one-to-many and many-to-one relationships [Pressman(1997)].

2.6 Object-Oriented Static Metrics

As the use of object-oriented design techniques in industry increased, the need to accurately measure the quality of object-oriented designs also increased. Software metrics such as function points [Fenton et al.(1996), Jeffery and Stathis(1996)], Lines of Code (LOC) [Conte et al.(1986)], that were already designed and empirically validated for structured or procedural development techniques, were tried upon object-oriented paradigm. However these metrics, that were originally developed for structured/procedural systems, were found lacking when applied to object-oriented software systems by the researchers [Bluemke(2001), Ebert and Morschell(1997)].

Object-oriented static metrics are design metrics that are computed from static analysis data of an object-oriented software in order to estimate the quality of the software. Over the years, many researchers have proposed and developed a number of static metrics for object-oriented software. The first of such metrics suite was CK suite [Chidamber and Kemerer(1994)] that was devised to replace structural/procedural metrics such as Lines of Code (LOC) [Conte et al.(1986)] because of latter’s inadequacies for the analysis of object-oriented software.
Table 2: CK Metrics [Chidamber and Kemerer (1991&1994)]

CK suite comprises of some very useful object-oriented design metrics such as Coupling Between Objects (CBO), Weighted Methods per Class (WMC) (Refer Table 2) that are used to measure various object-oriented attributes like coupling, inheritance and complexity. Other such static metrics were proposed by Li and Henry(1993), Abreu(1994&1995), Lorenz and Kidd(1994), Henderson-Sellers(1996), Briand *et al.*(1997), Benlarbi and Melo(1999), Tang *et al.*(1999), Cartwright and Shepperd(1997). The CK metric suite has still been the most famous and widely referenced metric suite till date [Chidamber and Kemerer(1991&1994), Xu *et al.*(2008)].

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBO</td>
<td>Coupling Between Object Classes (CBO) is the number of other classes with which a class is coupled to. A class is coupled with another class if the methods of one class use the methods or attributes of the other or vice versa. It includes inheritance based coupling (i.e. coupling between classes related via inheritance). A previous variant of CBO excludes inheritance-based coupling.</td>
</tr>
<tr>
<td>RFC</td>
<td>Response For a Class (RFC) is the number of methods in the response set of the class. Response set of a class consists of the set M of methods of the class and the set of methods directly invoked by the methods in M (i.e. the set of methods that can potentially be executed in response to a message received by that class). A variant of RFC excludes methods indirectly invoked by a method in M.</td>
</tr>
<tr>
<td>DIT</td>
<td>Dept of Inheritance Tree (DIT) metric is defined as the length of the longest path from the class to the root in the inheritance hierarchy.</td>
</tr>
<tr>
<td>NOC</td>
<td>Number of Children (NOC) counts the number of classes that inherit from a particular class (i.e. the number of classes in the inheritance tree down from a class).</td>
</tr>
<tr>
<td>WMC</td>
<td>Weighted Methods per Class (WMC) metric counts the number of methods in a class. Although the developers of this metric left the weighting scheme as an implementation decision, but if no weighting scheme is used, WMC becomes the number of methods implemented in a class.</td>
</tr>
<tr>
<td>LCOM</td>
<td>Lack of Cohesion of Methods metric measures the number of pairs of methods in the class that have no attributes in common, minus the number of pairs methods that do. If the difference is negative, the metric value is set to zero.</td>
</tr>
</tbody>
</table>

2.6.1 Coupling Concept and Coupling Metrics - Related work

*Coupling* [Chidamber and Kemerer(1994)]: Two objects are coupled, if and only if, at least one of them acts upon the other [Wand and Weber(1980)]. Let $X = (x, p(x))$ and $Y = (y, p(y))$ be two objects. Here $x, y$ are the names by which the objects $X, Y$ are represented and $p(x), p(y)$ are their respective set of properties. $X$ is said to act upon $Y$ if the history of $Y$ is affected by $X$, where *history* is defined as the chronologically ordered states that a thing traverses in time. Also,
Design and Validation of Dynamic Metrics for Object-Oriented Software Systems

\[
p(x) = \{M_x\} \cup \{I_x\} \\
p(y) = \{M_y\} \cup \{I_y\}
\]

where \(\{M_i\}\) is the set of methods and \(\{I_i\}\) is the set of instance variables of object \(i\). Using the above definition of coupling, any action by \(\{M_x\}\) on \(\{M_y\}\) or \(\{I_y\}\) constitutes coupling, as does any action by \(\{M_y\}\) on \(\{M_x\}\) or \(\{I_x\}\).


In his paper titled “On the criteria to be used in decomposing systems into modules”, Parnas (1972) introduced the criteria of information hiding for use in system decomposition of large scale systems. Parnas however did not use the term “coupling” in his research. This criteria was later used to prepare the design rules for well-formed classes in object-oriented design like low coupling and high cohesion. Stevens, Myers and Constantine first used the term “coupling” in software engineering two years later [Stevens et al. (1974)]. The general concept of coupling was introduced to account for count connections between modules. A module was defined as a subprogram, a procedure or a function. This work formed the basis for many of the famous coupling metrics that followed. Coupling was defined on an ordinal scale (low/high) depending upon the complexity of connection, strength of connection and the complexity of the message involved in that connection. They also classified coupling into data and control coupling.

Coad and Yourdan (1991) classified coupling into following two types as criteria for the design in their work on structural design and object-oriented design.

- **Interaction Coupling**: Coupling that results from the exchange of messages between two classes.
- **Inheritance Coupling**: Interconnection between generalizations and specializations.

They cautioned against the overuse of inheritance.

Berard (1993a) classified coupling into two major categories: necessary and unnecessary coupling. The term “object coupling” was also introduced in this work. He
stated “unnecessary object coupling needlessly decreases the reusability of coupled objects”.

Chidamber and Kemerer (1991 & 1994) introduced the first coupling metric for object-oriented systems. In their metric suite, commonly known as CK metric suite, they defined CBO (Coupling Between Objects) metric as number of non-inheritance related couples with other classes [Chidamber and Kemerer(1991)]. They concluded that high coupling leads to high complexity. They later revised their definition of CBO to include coupling due to inheritance [Chidamber and Kemerer(1991)].

Eder et al. (1994) used the definition of coupling provided by Stevens et al. (1974) and in 1994 classified coupling into 3 major types: Interaction, Component and Inheritance. They classified inheritance coupling on an ordinal scale. They introduced a third major type of coupling called component coupling. In component coupling, each object has a unique identifier (the object identity). An object o may reference another object p using the identifier of object p. This introduces a component relationship between the classes of o and p.

Hitz and Montazeri (1995) introduced two distinct levels of coupling, class-level and object-level coupling. They used definition of object coupling given by Berard (1993) and proposed a method for its estimation. The problem with their object-level coupling work is, that they claim that it measures dynamic coupling from runtime behavior of the system, but practically they end up estimating dynamic coupling from static code analysis.

Briand et al. (1999b) later extended the coupling classification proposed by Eder et al. (1994). Interaction, Inheritance and Component coupling were identified as the three dimensions of coupling. These are shown in Figure 6 and briefly described as follows:

i. **Interaction coupling:** two methods are interaction coupled if
   a) one method invokes the other, or
   b) they communicate via sharing of data.

   Interaction coupling between method m implemented in class c and method m’ implemented in class c’ contributes to interaction coupling between c and c’.

ii. **Component coupling:** two classes c and c’ are component coupled, if c’ is the type of either
   a) an attribute of c, or
   b) an input or output parameter of a method of c, or
   c) a local variable of a method of c, or
   d) an input or output parameter of a method invoked within a method of c.
iii. *Inheritance coupling*: two classes $c$ and $c'$ are inheritance coupled, if one class is an ancestor of the other.

![Diagram of Coupling Types]

**Figure 6: Types/Categories/Dimensions of coupling [Briand et al.(1999b)]**

There has been a lot of research done on structural coupling metrics over the years. Briand et. al. [Briand et al.(1999b)] describes many of such metrics in their work on unified framework for coupling measurement. Some of the famous coupling metrics are Coupling Between Objects (CBO) and CBO1 [Chidamber and Kemerer(1991&1994)], Response For Class (RFC) and RFC∞ [Chidamber and Kemerer(1991&1994)], Efferent Coupling(Ce) [Martin(1994&2003)], Afferent Coupling(Ca) [Martin(1994 & 2003)], Coupling Factor(COF) [Abreu(1995)], Message Passing Coupling(MPC) [Li and Henry(1993)], Data Abstraction Coupling (DAC) and DAC1 [Li and Henry(1993)], Information-flow-based Coupling (ICP) [Lee et al.(1995)], Briand et al.(1997) suite (IFCAIC, ACAIC, OCAIC, FCAEC etc), CP coupling metric for business process models [Irene et al.(2007)]. Most of these metrics are based on method invocations and attribute references. CBO and COF metrics count method invocations and references to both methods and attributes. RFC and MPC metrics count only method invocations.

Briand et al.(1997) metric suite records many types of interactions between object-oriented design attributes such as class-attribute, class method, as well as method-method
interactions. The metrics from this suite also differentiate between import and export coupling as well as other types of relationships like friends, ancestors, descendants etc.

Choi and Lee(2006) came up with a coupling metric and an approach to improving the existing component-based coupling metrics by considering the dependency about the structured relationships and the method call types between classes.

Coupling can also be divided into different types based on various levels of abstraction. These are system-level coupling, package-level coupling, method-level coupling [Singh and Singh(2007)] in addition to class-level coupling and object-level coupling defined in [Hitz and Montazeri(1995)].

Dynamic coupling measures [Mitchell and Power(2004a)] were introduced as the refinement to existing coupling measures due to gaps in addressing polymorphism, dynamic binding, and the presence of unused code by static structural coupling measures. They are one of the most talked about modern forms of coupling measures that are being researched upon these days with huge expectations from software engineering community. Dynamic coupling measures are thus the focus of this research. These measures are covered in the next Chapter along with all the other such dynamic measures.

Another vital type of coupling measures is derived from the evolution of software system in contrast to structural coupling which is evaluated from static program analysis or dynamic coupling which is computed from runtime analysis. These are called evolutionary couplings among parts of the systems which are determined by the common changes or co-changes [Zimmermann et al.(2005)]. Some application-based coupling metrics were also proposed for various real-world software systems. They are coupling metrics for knowledge-based systems [Kramer and Kaindl(2004)] and coupling metrics for aspect-oriented programs [Zhao(2004)] and service-oriented systems [Perepletchikov(2007)]. Recent work on software clustering [Kuhn et al.(2005), Maletic and Marcus(2001)] uses the concept of semantic similarity between elements of the source code. Semantic similarity between elements was further used to introduce another evolving concept called conceptual coupling and the concerned metric called conceptual coupling metric [Poshyvanyk and Marcus(2005&2006)].

Inheritance-based metrics (as shown in Table 3) were being developed alongside. Inheritance has been covered over the years as a separate concept with its own metrics. Impact of inheritance on attributes such as coupling, cohesion and size has also been researched upon.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIT (Depth of Inheritance) [Chidamber and Kemerer(1994)]</td>
<td>The DIT of a class is the length of the longest path from the class to the root in the inheritance hierarchy.</td>
</tr>
<tr>
<td>AID (Average Inheritance Depth of a Class) [Henderson-Sellers(1996)]</td>
<td>AID of a class having no ancestors is zero. For all the other classes, AID of a class is average AID of its parent classes, increased by one.</td>
</tr>
<tr>
<td>CLD (class-to-leaf) depth [Tegarden et al.(1995)]</td>
<td>CLD of a class is the maximum number of levels in the hierarchy that are below the class.</td>
</tr>
<tr>
<td>NOC (Number of Children) [Chidamber and Kemerer(1994)]</td>
<td>Self Explanatory</td>
</tr>
<tr>
<td>MIF (Method Inheritance Factor) [Abreu(1995)]</td>
<td>MIF for a system is the percentage of total number of inherited methods to the total number of methods (including the inherited methods) of all the classes of the system.</td>
</tr>
<tr>
<td>AIF (Attribute Inheritance Factor) [Abreu(1995)]</td>
<td>AIF for a system is the percentage of total number of inherited attributes to the total number of attributes (including the inherited attributes) of all the classes of the system.</td>
</tr>
<tr>
<td>NOP (Number of Parents) [Lake and Cook(1994), Lorenz and Kidd(1994)]</td>
<td>Self Explanatory</td>
</tr>
<tr>
<td>NOD (Number of Descendants) [Lake and Cook(1994), Tegarden et al.(1995)]</td>
<td>Self Explanatory</td>
</tr>
<tr>
<td>NOA (Number of Ancestors) [Tegarden et al.(1995)]</td>
<td>Self Explanatory</td>
</tr>
<tr>
<td>NMO (Number of Methods Overridden) [Lorenz and Kidd(1994)]</td>
<td>Number of methods in a class that override a method inherited from an ancestor class.</td>
</tr>
<tr>
<td>NMI (Number of Methods Inherited) [Lorenz and Kidd(1994)]</td>
<td>Number of methods in a class that the class inherits from its ancestors and does not override.</td>
</tr>
<tr>
<td>NMA (Number of Methods Added) [Lorenz and Kidd(1994)]</td>
<td>The number of new methods in a class, not inherited, not overriding.</td>
</tr>
<tr>
<td>SIX (Specialization Index) [Lorenz and Kidd(1994)]</td>
<td>SIX is NMO*DIT/(NMO+NMA+NMI)</td>
</tr>
</tbody>
</table>

**Table 3: Inheritance Coupling Metrics**

Most metrics for size, coupling and cohesion within the object oriented area ignore inheritance [Beyer et al.(2001)]. Various inheritance-measuring metrics are Number of Children (NOC), Depth of Inheritance Tree (DIT), Number of Parents, Attribute Inheritance Factor (AIF), Method Inheritance Factor (MIF), Number of methods inherited by a subclass, number of methods overridden by a subclass etc. NOC and DIT are a part of the CK metric suite [Chidamber and Kemerer(1994)]. AIF and MIF are from MOOD metric suite [Abreu(1995)].

The effect of inheritance on the various external quality attributes such as fault-proneness, understandability, maintainability etc has been empirically proven [Daly et al.(1996), Basili et al.(1996), Chidamber and Kemerer(1994)]. The use of inheritance has
been proved to reduce the amount of software maintenance necessary and ease the burden of testing [Chidamber and Kemerer(1994)]. Also the reuse of software through inheritance is claimed to produce more maintainable, understandable and reliable software [Basili et al.(1996)]. It is however evident that the industrial acceptability of academic metrics research has been slow. One of the reasons could be a lack of perceived need. The results of such research are not often applied to industrial software [Fenton and Neil(1999)], which makes validation a daunting and difficult task. Some other contradicting results have been a problem as well. For example, the experimental research of Harrison et al. (1998) indicates that a system not using inheritance is better for understandability or maintainability than a system that uses inheritance. However, Daly’s experiment [Daly et al.(1996)] indicates that a system with three levels of inheritance is easier to modify than a system with no inheritance. Rajnish and Bhattacherjee carried out research related to class inheritance metrics [Rajnish and Bhattacherjee(2005,2006a,2006b&2007]. Some famous inheritance-based static metrics are briefly described in the Table 3.

Mitchell and Power(2004a) placed a need for further research to find out the impact of inheritance on coupling at runtime. There are no runtime/dynamic metrics that measure inheritance during execution. It has to be pointed out here that almost all the metrics that are defined to measure size, coupling and cohesion for object-oriented systems, ignore inheritance [Beyer et al.(2001)]. So there are just a limited number of static metrics that measure inheritance.

2.7 Object-Oriented Dynamic/Runtime Metrics

In today’s software-oriented world, the key to success for an object-oriented software system is its regular performance appraisal that further demands certain measurement criteria that can be effectively used to measure the working performance of the system. Software metrics measure different aspects of software complexity and thus play an important role in analyzing and improving software quality [Rosenberg and Hyatt(1995)]. Previous studies have shown that they provide useful information on external quality aspects of software such as its maintainability, reusability, reliability, fault proneness etc derived from important characteristics of object-oriented software such as size, polymorphism, inheritance, coupling and cohesion.

However, static metrics are not capable of predicting the dynamic behavior of a software application. This is because the behavior of a software application is not only
influenced by the complexity, but also by the operational or runtime environment of the source code. A latest research by Yacoub et al. (1999) has indicated that useful information may be obtained from a measure of quantifying the dynamic complexity of software in its operational environment. Thus it is possible to quantify the quality of a software product using a set of dynamic metrics calculated at run-time that evaluate the product’s complexity. These metrics by themselves can provide us with useful information, and can also help to calibrate the information obtained from a static analysis.

2.7.1 Need: Static Vs Dynamic Analysis

Let us differentiate between static and dynamic analysis with the help of a simple C++ program. The program in Figure 4.1 takes an input from the user and calls a function function1 depending upon an if condition. If the value of a simple metric like ‘Number of Functions called’ is evaluated before runtime, it gives the value 1. But if the value is evaluated during runtime, it can be 0 or 1 depending upon the user input. So in this case, the output of a program depends upon the user input that can only be known at runtime. User input is just one of many factors that differentiate static analysis results from dynamic/runtime results.

As the example program in Figure 7 uses a structural/procedural paradigm, it can demonstrate only general method-method or method-attribute coupling. This gap between static and dynamic analysis even widens in case of object-oriented systems due to the presence of concepts such as inheritance and dynamic polymorphism. Figure 8 shows how the actual behavior of an object evaluated at runtime can vary from its expected behavior evaluated from static analysis. In Figure 8(A), an object A1 is expected to call

```c++
function_1()
{
    cout << "You never know before runtime";
}

void main()
{
    int a;
    cin >> a;
    if(a<3)
    {
        function1();
    }
}
```

*Figure 7: A simple C++ program code*
a) Before runtime

b) At runtime

Figure 8: Static vs Dynamic analysis - Object interactions

objects B1, B2, C1 and C2 as evaluated from the static analysis. But during runtime (Figure 8(B)), it only calls B1 and C1, hence showing a variation from its expected behavior shown by the static analysis.

Mitchell and Power(2003a,2003b,2003c,2004a,2004b&2005) conducted a number of studies on the quantification of many runtime object-level coupling metrics for Java-written object-oriented programs. They have also statistically analyzed the differences in the underlying dimensions of coupling captured by static versus dynamic coupling metrics. The results across all the test cases of their work showed that a static and dynamic analysis of a program does not produce equivalent results, suggesting that it may be worthwhile conducting both types of analysis [Mitchell and Power(2003b)]. The reasons for the different results obtained from the static and dynamic analysis may arise from the fact that static metrics are concerned with statically coupled and complex design elements whereas dynamic metrics are concerned with frequently invoked and frequently executed objects [Yacoub et al.(1999)]. Next we explore certain internal design attributes for which static and dynamic analysis produce variable results in an object-oriented environment.

2.7.1.1 Reasons for Variations in Static and Dynamic Analysis Results

The major differences between the static and dynamic analysis results are unveiled during the use of object-oriented internal design characteristics like coupling, cohesion, inheritance and dynamic binding.
i. **Coupling and Cohesion [Chidamber and Kemerer(1991)]** 

Two objects are said to be coupled if and only if at least one of them acts upon the other [Wand and Weber(1980)]. Let $X = (x,p(x))$ and $Y = (y,p(y))$ be two objects. Here $x$ and $y$ are the names by which the objects $X$, $Y$ are represented and $p(x)$, $p(y)$ are their respective set of properties. $X$ is said to act upon $Y$ if the history of $Y$ is affected by $X$, where *history* is defined as the chronologically ordered states that a thing traverses in time. Also,

$$p(x) = \{M_x\} \cup \{I_x\}$$

$$p(y) = \{M_y\} \cup \{I_y\}$$

where $\{M_i\}$ is the set of methods and $\{I_i\}$ is the set of instance variables of object $i$. Using the above definition of coupling, any action by $\{M_x\}$ on $\{M_y\}$ or $\{I_y\}$ constitutes coupling, as does any action by $\{M_y\}$ on $\{M_x\}$ or $\{I_x\}$.

A method being observed by static metrics which is expected to contribute to coupling between two classes may in fact be rarely invoked at runtime. This would result in a much lower actual coupling between classes/objects at runtime [Mitchell and Power(2003c)]. The reason can be a user input dependent code or could be dynamic binding/inheritance which are also the other key reasons for static and dynamic behavioral differences.

Cohesiveness of a class is measured by the degree of similarity [Bunge(1977)] in methods of that class. The *degree of similarity* of methods is a major aspect of object class cohesiveness. Consider a class $A$ having a method set $\{M_i\}$ and an instance variable set $\{I_i\}$. Then, the degree of similarity in a method pair $\{M_1,M_2\}$ having an instance variable pair $\{I_1, I_2\}$ is given by

$$\sigma(M_1, M_2) = \{I_1\} \cap \{I_2\}$$

If an object class has different methods performing different operations on the same set of instance variables, the class is cohesive. Let us consider an example to see the effect of static and dynamic analysis on cohesion.

Consider, for example, a class $A$ having two cohesive methods $a()$ and $b()$. Let $A$ have two non-cohesive methods $c()$ and $d()$. If $a()$ and $b()$ are called more often than $c()$ and $d()$ before runtime, the class $A$ becomes cohesive or more cohesive whereas if $c()$, $d()$ are called more often than $a()$, $b()$ during runtime, class $A$ becomes non-cohesive or
lesser cohesive. Thus at runtime, a statically cohesive class may not exhibit cohesive behavior. That is, the methods that detract from cohesive behavior might far outweigh those that contribute to cohesion when weighed by the number of times they are invoked.

ii. Object lifetime variations
Some classes have state-based behavior. That is, we can distinguish between the various stages in the lifetime of their objects: perhaps an initialization stage, followed by a period of activity, followed by a finalization phase [Mitchell and Power(2003c)]. It seems reasonable that some objects may exhibit high degrees of cohesion and low coupling during initialization, but exhibit low cohesion and high coupling during their active phase. Static metrics fail to quantify or identify this behavior, whereas this information should be available from the runtime metrics.

iii. Dynamic Binding
Many object-oriented languages support runtime (/dynamic) binding of messages being passed between objects [Li and henry(1993)]. Also called "late binding," it is the linking of a method or object at runtime based on conditions at that moment. That is during dynamic binding, a run-time selection of methods is performed via lookup (bound) at run-time (dynamically). This is often desired and even required in many applications including databases, distributed programming and user interaction (e.g. GUIs). The concept is a superset of runtime polymorphism. In dynamic binding, the actual methods called can only be known late during execution and hence static analysis fails to predict such method bindings.

iv. Inheritance
One of the most influential features of object-oriented paradigm is the concept of inheritance which allows one class to be defined in terms of another. A derived class is a class defined in terms of another class. A derived class is a class defined in the context of some other class with possible substitutions and additions of methods and members. An object oriented program will contain atleast one class hierarchy, consisting of a base class and any classes which are directly or indirectly derived from the base class [Li and henry(1993)]. A program can behave differently at runtime as that expected from static analysis. This is because there is no way of finding out before
runtime whether an invoked method of a class, actually belongs to that class or is inherited from one of its base classes.

For example, Degree of Dynamic Coupling between two classes A and B is a count of the amount of times class A accesses methods or instance variables from a class B as a percentage of total number of methods or instance variables accessed by A at runtime. Let C be the base class of B. If an object of A accesses a method of B the value of the metric will increase. Now there can be a situation where during runtime the method of B accessed by A’s object is actually inherited from C. As this cannot be known before runtime, the metric value for classes A and B will be incremented, thus making a wrong assessment during static analysis.

This work provides a pack of runtime inheritance related coupling metrics that can capture situations like the one presented in above mentioned example. Following section walks through the research work done in the field of dynamic/runtime object-oriented metrics.

### 2.7.2 Dynamic Object-Oriented Metrics – Concept

#### 2.7.2.1 Definition

Dynamic Metrics can be defined as the metrics used to measure the internal quality attributes of object-oriented software systems by working on the information gathered from the dynamic (runtime) behavioral analysis of software. Dynamic metrics are thus accurately evaluated only at late development stage.

#### 2.7.2.2 Desired Properties

Dynamic metrics must have a number of characteristics to be useful in measuring the performance of a software application. The following is a non-exhaustive list of such characteristics [Dufour et al. (2003)].

- **Unambiguous:** An ambiguous definitions lead to unusable metrics. So, dynamic metrics must be unambiguous in nature. For instance, the most widely used metric for program size is ’lines of code’ (LOC). LOC is sufficient to give a ball park measure of program size. However, without further specification it is virtually useless to compare two programs. Are comments and blank lines counted? How do you compare two programs from different languages? Within a given language, the LOC of a pretty-printed version of a program with comments and blank lines
removed would give an unambiguous measurement that can be used to compare two programs.

- **Dynamic:** The metric should measure an aspect of a program that can only be obtained by actually running the program. In compiler optimization papers, static metrics are often given because they are easier to obtain. They tend to relate to the cost of a particular optimization technique (e.g. the number of virtual call sites for a de-virtualization technique), whereas dynamic metrics relate to the relevance of a technique (e.g. the proportion of dynamically executed monomorphic call sites). While dynamic metrics usually require more work than static measurements, the resulting numbers will be more meaningful since they will not change by adding unexecuted code to the program. Dead code should not influence the measurement. We will refer to instructions that are executed at least once as *instructions touched*, or live code.

- **Robust:** A small change in behavior should cause a correspondingly small change in the resulting metric. Dynamic measurements are heavily influenced by program behavior. Where static numbers may have reduced relevance because non-executed code influences the numbers, dynamic metrics may have reduced relevance because the measured program execution may not reflect common behavior. Unfortunately, one simply cannot guarantee that a program’s input is representative. However, one can take care to define metrics that are robust with respect to program behavior.

- **Discriminating:** A large change in behavior should cause a correspondingly large change in resulting metric. Dynamic metrics must reflect changes in program behavior.

- **Machine-independent:** Metric values should not change if the measurement takes place on a different platform (including virtual machine implementation).

### 2.7.2.3 Types

From a relatively small number of dynamic metrics available today, it has been observed that most of these metrics fall in one of the four categories depending upon the type of value stored by the metrics. These are four main categories are [Dufour et al.(2003)] shown in Table 4.
Design and Validation of Dynamic Metrics for Object-Oriented Software Systems

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Metric that is represented by a single value. e.g. Average, Maximum etc.</td>
</tr>
<tr>
<td>Percentile</td>
<td>Metrics that define the distribution of contributions from system entities. e.g. X% of characteristic A accounts for Y% of B, rather than count(A) and count(B), for example.</td>
</tr>
<tr>
<td>Bins</td>
<td>Metrics representing classes of behaviors of special interest. Bins work in absolute or relative value ranges like x=1 &amp; x&gt;1. e.g Categorizing runtime safety checks according to the type.</td>
</tr>
<tr>
<td>Continuous</td>
<td>Metrics representing time series data of dynamic behavior collected at a given time interval or state. e.g. a line graph representing value metric for objects of a class over a given time interval.</td>
</tr>
</tbody>
</table>

Table 4: Types of dynamic metrics

2.7.3 Related Research

Schikuta(1993) was the first to propose a dynamic approach to measure coupling of software systems. He defined static measures as those that are derived from the source code. It was concluded that the conventionally used measurement systems were static oriented and provide the incomplete dynamic behavior of the system. This is because static systems give only measures for syntactical program analysis and thus have limited ability. They also insisted that the dynamic measures should always be used in combination with their static counterparts. Overall view of the programming module in question can only be given using dynamic components.

Briand et al.(1997) studied a vast amount of available literature on coupling in object-oriented systems and concluded that all the metrics at that time measured coupling at the class level from static code. No measures of runtime object-level coupling had been proposed till that time.

Paques and Delcambre(1999) presented an approach to evaluate the dynamic coupling at analysis phase. They proposed Dynamic Clustering Mechanism (DCM) that works by tracking hot spots for dynamic coupling at analysis phase. DCM constructs clusters of classes that are highly dynamically coupled. They stated “Dynamic coupling is based on the frequency with which classes interact at runtime”. Class interaction frequencies are computed by propagating use case frequencies to sequence diagrams. Interactions inside the cluster and outside the cluster are computed for each cluster.

Yacoub et al.(1999) defined a set of object-level dynamic coupling metrics (listed in Table 2.9) designed to evaluate the change-proneness of a design. The metrics were
applied at an early development phase to determine design quality. They used executable object-oriented design models to model the application to be tested. The metrics were evaluated for a number of different execution scenarios, and they extended the scenarios to have an application scope.

Arisholm (2002) proposed dynamic Import Coupling (IC) and Export Coupling (EC) measures quantifying message communication among objects at runtime. These measures are further defined at object-level and class-level with each metric based upon either number of messages sent/received or methods invoked or classes accessed. They studied the relationship of these metrics with the change proneness of a system. These metrics are briefly described in Table 6. They found that the dynamic coupling measurement did capture additional properties to the static coupling metrics and were good predictors of the change proneness of a class.

Dufour et al. (2003) defined five families of dynamic metrics for Java-based software systems which characterized a program’s runtime behavior in terms of size and control structure, data structures, polymorphism, memory use, and concurrency and synchronization. These metrics were designed with the goals of being unambiguous, dynamic, robust, discriminative, and machine-independent, and are meant to quantitatively characterize a benchmark in ways relevant to compiler and runtime developers. Based on values contained, four types of dynamic metric were also introduced as value metrics, percentile metrics, bins metrics and continuous metrics. These metrics are briefly defined in Table 11.

Gupta and Rao (2001) compared statically calculated metrics against a program execution based approach of measuring the levels of module cohesion. The results showed that the static approach significantly overestimated the levels of cohesion present in the software under test.

Thompson (2002) worked to show that the dynamic behavior of an object is related to its static structure according to its state space complexity. He proposed a new metric called Relative Dynamic Complexity Metric for Object (RDCMO) that represents the degree of dynamic state-space behavior of an object. It considers the number of state traversals an object would undergo relative to its modular requirements. RDCMO is defined in Table 10.

Mitchell and Power (2003a, 2003b, 2003c, 2004a, 2004b & 2005) conducted a number of studies on the quantification of many runtime class-level coupling metrics (listed in Table 5) for object-oriented (Java) programs. They have also statistically analyzed the differences
in the underlying dimensions of coupling captured by static versus dynamic coupling metrics using CK metrics [Chidamber and Kemerer(1994)] like CBO and LCOM. The results indicated that the run-time metrics did capture different properties than the static metrics alone.

Zaidman and Demeyer(2004) exposed the usage of dynamic coupling metrics in analyzing the event traces of large scale industrial applications. They also devised a new class-level dynamic coupling metric called Class Request For Service (CQFS), described in Table 7, to measure software performance at runtime.

Hassoun et al.(2004&2005) studied object-level coupling as it evolves during program execution and proposed a dynamic coupling measure that takes object interactions into consideration. They also proposed a dynamic metric DCM (Dynamic Coupling Metric) illustrated in Table 8. The metric can be used to compare systems built on meta-level architectures with systems having no reflective features yet, at the same time, exhibiting the same interface.


Beszedes et al.(2007) introduced the measure of dynamic function coupling (DFC) between two functions or methods, which they used to define a more precise way of computing impact sets on function level with a scalable rate of recall.

Yuying et al.(2007) in their work on fan-in and fan-out metrics [Henry and Kafura(1981)] introduced a set of dynamic metrics meant to discover knowledge about software systems thus facilitating program comprehension.

Quynh and Thang(2009) proposed a dynamic coupling metric suite for service-oriented systems that evaluates service’s quality according to its ability of coupling. They used the coupling metrics to measure the maintainability, reliability, testability, and reusability of services. Another dynamic coupling metric was proposed by Choi and Lee(2006&2007) for embedded systems using service-oriented approach. Rothlisberger et al.(2009) emphasized the need of integrating dynamic metrics to IDE’s like Eclipse to help maintain the object-oriented applications written in languages like java. They also introduced an Eclipse plugin for the purpose.

Chapter 1 lists the conclusions and future research recommendations for dynamic object-oriented metrics drawn by the research community over the past decade. Research till date has suggested various runtime or dynamic metrics. Tables 5–11 list all the major dynamic metrics along with their descriptions.
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime or Dynamic Coupling between Objects (CBO) for a class A.</td>
<td>Number of couples with other classes at runtime</td>
</tr>
<tr>
<td>Degree of Dynamic Coupling between a class A and class B</td>
<td>Number of 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>
</tr>
<tr>
<td>Degree of Dynamic Coupling within a given set of classes</td>
<td>Number of times methods or instance variables are accessed outside each class as a percentage of the total number of methods or instance variables accessed by these classes.</td>
</tr>
<tr>
<td>Runtime Simple Lack of Cohesion Metric ($R_{LCOM}$)</td>
<td>Number of pairs of method in the class that have no instance variables in common, minus the number of pairs of methods that have common instance variables at runtime.</td>
</tr>
<tr>
<td>Runtime Call-Weighted Lack of Cohesion Metric ($RW_{LCOM}$)</td>
<td>Sum of number of accesses to instance variables from the number of pairs of method in the class that have no instance variables in common, minus sum of the number of accesses by the number of pairs of methods that have common instance variables a runtime.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import Coupling Metrics</td>
<td></td>
</tr>
<tr>
<td>IC_CD</td>
<td>Number of messages sent from a class A during a given scenario.</td>
</tr>
<tr>
<td>IC_CM</td>
<td>Number of different methods used by a class A to send all the messages in a given scenario.</td>
</tr>
<tr>
<td>IC_CC</td>
<td>Number of distinct classes to which a class A has sent messages in a given scenario.</td>
</tr>
<tr>
<td>IC_OD</td>
<td>Number of messages sent from an object A during a given scenario.</td>
</tr>
<tr>
<td>IC_OM</td>
<td>Number of different methods used by an object A to send all the messages in a given scenario.</td>
</tr>
<tr>
<td>IC_OC</td>
<td>Number of distinct classes to which an object A has sent messages in a given scenario.</td>
</tr>
<tr>
<td>Export Coupling Metrics</td>
<td></td>
</tr>
<tr>
<td>EC_CD</td>
<td>Number of messages received by a class A during a given scenario.</td>
</tr>
<tr>
<td>EC_CM</td>
<td>Number of different methods used by a class A to receive all the messages sent to it in a given scenario.</td>
</tr>
<tr>
<td>EC_CC</td>
<td>Number of distinct classes from which a class A has received messages in a given scenario.</td>
</tr>
<tr>
<td>EC_OD</td>
<td>Number of messages received by an object A during a given scenario.</td>
</tr>
<tr>
<td>EC_OM</td>
<td>Number of different methods used by an object A to receive all the messages sent to it in a given scenario.</td>
</tr>
<tr>
<td>EC_OC</td>
<td>Number of distinct classes from which an object A has received messages in a given scenario.</td>
</tr>
</tbody>
</table>

**Table 6: Arisholm (2002) metric suite**
Table 7: Zaidman and Demeyer(2004) metric suite

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Request For Service (CQFS)</td>
<td>Number of messages sent by a class to the other classes or their objects during the execution of a program.</td>
</tr>
</tbody>
</table>

Dynamic Coupling Metric (DCM) (for an object $P$)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Coupling Metric (DCM) (for a Complete System)</td>
<td>Sums the DCMs of all the objects in the system for a given time period $\Delta t$.</td>
</tr>
</tbody>
</table>

Table 8: Hassoun et al.(2004&2005) metric suite

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Export Object Coupling (EOC)</td>
<td>$EOC_x(o_i \text{ w.r.t. } o_j)$: Percentage of the number of messages sent from object $o_i$ to object $o_j$ with respect to the total number of messages exchanged during the execution of the scenario (a sequence of interactions between objects stimulated by input data or events).</td>
</tr>
<tr>
<td>Import Object Coupling (IOC)</td>
<td>$IOC_x(o_i \text{ w.r.t. } o_j)$: Percentage of the number of messages received by object $o_i$ and was sent by object $o_j$ with respect to the total number of messages exchanged during the execution of the scenario.</td>
</tr>
<tr>
<td>Object Request for Service (OQRS)</td>
<td>Percentage of the total number of messages sent by an object $o_i$ to all other objects.</td>
</tr>
<tr>
<td>Object Response for Service (OPFS)</td>
<td>Percentage of the total number of messages sent to an object $o_i$ by all other objects in the application during the execution of a scenario.</td>
</tr>
<tr>
<td>Operational Complexity Metric (OCPX)</td>
<td>Sum of dynamic cyclomatic complexities of execution path for an object $o_i$ taken over all the scenarios.</td>
</tr>
</tbody>
</table>

Table 9: Yacoub et al.(1999) metric suite

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Dynamic Complexity Metric for Objects (RDCMO)</td>
<td>Ratio of relevant state space traversals to total state space transitions divided by the total number of Relative Requirements (RR’s) that directly affect the object.</td>
</tr>
</tbody>
</table>

Table 10: Thompson(2002) metric suite
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size.appLoadedClasses.value</td>
<td>Number of application-specific classes loaded.</td>
</tr>
<tr>
<td>size.appLoad.value</td>
<td>Number of bytecode instructions loaded in application specific classes.</td>
</tr>
<tr>
<td>size.appRun.value</td>
<td>Number of bytecode instructions touched.</td>
</tr>
<tr>
<td>size.appHot.value</td>
<td>Number of bytecode instructions responsible for 90% of execution.</td>
</tr>
<tr>
<td>structure.controlDensity.value</td>
<td>Total number of control bytecodes touched divided by the total number of bytecodes touched.</td>
</tr>
<tr>
<td>structure.changingControlDensity.value</td>
<td>Total number of control bytecodes that change direction at least once divided by the total number of bytecodes touched.</td>
</tr>
<tr>
<td>structure.changingControlRate.value</td>
<td>Number of changes in direction divided by the number of control instruction executions.</td>
</tr>
<tr>
<td>data.[app]arrayDensity.value</td>
<td>Average number of array access operations per kbc of executed code.</td>
</tr>
<tr>
<td>data.[app]floatDensity.value</td>
<td>Number of floating-point operations per kbs of executed code.</td>
</tr>
<tr>
<td>pointer.[app]RefFieldAccessdensity.value</td>
<td>Average number of field access operations that reach an object field.</td>
</tr>
<tr>
<td>pointer.pointsToCount.value</td>
<td>Measures the average number of distinct objects referenced by each object reference and the average number of object references directed at each object respectively.</td>
</tr>
<tr>
<td>polymorphism.[app]CallSites.value</td>
<td>Total number of different call sites executed.</td>
</tr>
<tr>
<td>polymorphism.appInvokeDensity.value</td>
<td>Number of invokeVirtual and invokeInterface calls per kbc executed.</td>
</tr>
<tr>
<td>polymorphism.appReceiverArity.bin</td>
<td>Percentage of all calls that occur from a call site with one, two and more than two different receiver types.</td>
</tr>
<tr>
<td>polymorphism.appReceiverCacheMissRate.value</td>
<td>Percentage how often a call site switches between receiver types.</td>
</tr>
<tr>
<td>polymorphism.appTargetArity.bin</td>
<td>Percentage of all call sites that have one, two and more than two different target methods.</td>
</tr>
<tr>
<td>memory.byte[App]AllocationDensity.value</td>
<td>Measures the number of bytes allocated per kbc executed.</td>
</tr>
<tr>
<td>memory.object[App]AllocationDensity.value</td>
<td>Similar to the previous metric, but reports the number of objects allocated per kbc executed.</td>
</tr>
<tr>
<td>memory.[app]objectSize.bin</td>
<td>Object size distributions can be represented using this bin metric, where each bin contains the percentage of all objects allocated corresponding to the sizes associated with each bin.</td>
</tr>
<tr>
<td>concurrency.threadDensity.value</td>
<td>Maximum number of threads simultaneously in the ACTIVE or RUNNABLE states.</td>
</tr>
<tr>
<td>concurrency.lockDensity.value</td>
<td>Average number of lock operations per kbc.</td>
</tr>
</tbody>
</table>

Table 11: Dufour et al. (2003) metric suite
Chapter Summary

This chapter covered the thorough survey conducted over the related work done in the field of static and dynamic object-oriented software metrics. In the initial phase, the chapter briefs about the basics of the software metric concept and its relation to the quality attributes. Software metric basics also include an overview of various types of metrics evolved over the last few decades. It is followed by the methodology to define and validate a software metric. As this research is based upon object-oriented metrics, object-oriented concept is described next including various object-oriented design concepts such as coupling, cohesion, inheritance etc. Object-oriented concepts are followed by the work related to static object-oriented metrics. The reasons behind the evolution of dynamic metrics were then studied along with the comparison of static and dynamic metric analysis. The desired features of dynamic metrics along with the types of dynamic metrics are discussed next. Chapter ends with an illustration of all the latest dynamic object-oriented metric suites known as per literature conducted for this research. The next chapter introduces the new dynamic object-oriented metric suite proposed in this research.