



NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

**National University of Ireland, Maynooth**  
MAYNOOTH, CO. KILDARE, IRELAND.

DEPARTMENT OF COMPUTER SCIENCE  
TECHNICAL REPORT SERIES

# **NUIM-CS-TR-2003-08**

Run-time Cohesion Metrics for the Analysis of Java Programs - preliminary  
results from the SPEC and Grande suites

Aine Mitchell and James Power

# Run-time Cohesion Metrics for the Analysis of Java Programs

- *preliminary results from the SPEC and Grande suites*

Aine Mitchell and James F. Power

Department of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland

{ainem,jpower}@cs.may.ie

## ABSTRACT

Software metrics measure different aspects of software complexity and therefore play an important role in analyzing and improving software quality. Given the importance of object-oriented design techniques, a large number of object-oriented metrics for statically evaluating a design have been proposed. Cohesion is such a measure that evaluates the internal complexity of a design. A large body of research has gone into investigating how this complexity measure characterizes the external quality attributes of a design, for example its maintainability or reusability. However static measures only capture certain underlying dimensions of cohesion. Other dependencies regarding the dynamic behaviour of a program can only be inferred from run-time information. The quality of a software product will therefore be influenced by its operational environment as well as the source code complexity. Consequently measures that assess the run-time behaviour may aid in the analysis of software quality.

This paper describes new dynamic class level cohesion metrics suitable for the run-time evaluation of a program. It characterizes the ability of these metrics to evaluate the external quality attributes of a design. These dynamic cohesion metrics are then applied to assess the quality of Java programs from the Java Grande Forum Benchmark Suite and the SPECjvm98 Benchmarks. An investigation is also conducted to see if the results bear any relation to those obtained from a static analysis.

## 1. INTRODUCTION

Recent years have seen the increasing use of the object-oriented paradigm in software development. The use of object-oriented software development techniques introduces new elements to software complexity both in the software development process and in the final product.

Software metrics measure different aspects of software complexity and therefore play an important role in analyzing

and improving software quality [1, 4]. They provide useful information on external quality aspects of software such as maintainability, reusability and reliability, and provide a means of estimating the effort needed for testing.

Traditional metrics for measuring software such as Lines of Code (LOC) have been found to be inadequate for the analysis of object-oriented software [10]. In recent years many researchers and practitioners have proposed a number of static code metrics for object-oriented software, for example, the suite of metrics proposed by Chidamber and Kemerer [6, 5]. These code metrics quantify different aspects of the complexity of the source code. However, the ability of such static metrics to accurately predict the dynamic behaviour of an application is as yet unproven.

Static metrics alone may be insufficient in evaluating the dynamic behaviour of an application at runtime, as its behaviour will be influenced by the operational environment as well as the complexity of the source code. Research has indicated that useful information may be obtained from a measure of quantifying the dynamic complexity of software in its operational environment [18]. For this reason this paper defines dynamic metrics that quantify the run-time complexity of object-oriented software.

Cohesion has been proposed as one of the fundamental qualitative measures of the goodness of a software design or implementation [16]. Cohesion is defined as the degree to which the tasks performed by a single module are functionally related. In other words it measures how closely the local methods of a class are related to the local instance variables. Cohesion is said to quantify the internal complexity and this provides a means of evaluating the internal quality of a class. It is thought to be good software design practice to design classes to be as cohesive as possible in an object-oriented program. In a highly cohesive class the elements of the class, which are its methods and instance variables, will be related to the performance of a single function. The benefit of having software that consists of a set of highly cohesive classes is reflected in the effects on the external quality of the design. A set of highly cohesive classes should be easier to maintain, develop and reuse and should also be substantially less error prone.

Briand et. al. have carried out a thorough survey of the literature regarding cohesion [3]. They concluded that no

Technical report  
NUIM-CS-TR2003-08

Department of Computer Science  
National University of Ireland  
Maynooth, Co. Kildare, Ireland.

measures for dynamically determining the cohesiveness of a class are currently available.

Java is a general-purpose object-oriented programming language developed by Sun Microsystems [11]. As it is being increasingly used as a development language for new software products [15], it is important to have a means of evaluating the quality of such products.

The remainder of this paper is organized as follows. Section 2 describes previous work that has been conducted on cohesion metrics. Section 3 outlines a number of new dynamic cohesion metrics. Section 4 describes how the measures are collected. Section 5 outlines the results from this study. Section 6 concludes the paper and discusses possible future work.

## 2. RELATED WORK

In this section there is an overview of a static measure of cohesion defined by Chidamber and Kemerer and a brief summary of the current research in dynamic cohesion metrics.

### 2.1 Static Cohesion Metrics

Chidamber and Kemerer defined a static cohesion metric for object-oriented applications known as Lack of Cohesion in Methods (LCOM). They then related the metric to the maintenance, testing and understandability of a design.

The LCOM measure was originally defined as “the degree of similarity of methods” [5]. For a given class  $C$  with a number of methods,  $m_1, m_2, \dots, m_n$ . Let  $\{I_i\}$  be the set of instance variables accessed by the method  $m_i$ . As there are  $n$  methods there will be  $n$  such sets, one set per method. The LCOM metric is then determined by counting the number of disjoint sets formed by the intersection of the  $n$  sets.

However this was found to be quite ambiguous and the pair later redefined their metric [6]. Two values are defined,  $P$  and  $Q$ .  $P$  is the number of pairs of methods whose intersection is equaled to the null set, that is, pairs of methods that do not have any instance variables in common.  $Q$  is the number of pairs of methods whose intersection is not equaled to the null set, that is methods that access one or more of the same instance variables. LCOM is then equaled to  $|P| - |Q|$ , if  $|P| > |Q|$ . Otherwise LCOM is assigned a value of zero.

They stated that the higher the degree of similarity of methods, the greater the cohesiveness of methods and the higher the degree of encapsulation of the class. If a class lacked cohesion this implied that the classes should possibly be divided into sub-classes. The LCOM metric was deemed useful for identifying flaws in the design of classes. The rationale is that a structure that lacks cohesion will be more complex, thereby increasing the likelihood of errors in its development process.

### 2.2 Current Research in Dynamic Cohesion Metrics

Briand *et.al.* [3] carried out an extensive survey of the current available cohesion literature in object-oriented systems

and concluded that all the current metrics measured cohesion at the class level (static analysis). No measures of object level cohesion had been proposed (dynamic analysis). They suggested that the reason for this is the obstacle of determining the degree of cohesion within individual objects. They proposed that a way of evaluating these would be to find some method of instrumenting the source code to log all occurrences of object instantiations, deletions, method invocations, and direct reference to attributes while the system is executing. Even though no methods of measuring cohesion at runtime for object-oriented systems had been proposed, a number of researchers were found to be investigating applying dynamic metrics at other stages of the software life-cycle [7].

A study was conducted by Gupta and Rao comparing a program execution based approach of measuring the levels of module cohesion present in legacy software, with a static based method [12]. The results from this study showed that the static based approach significantly overestimated the levels of cohesion present in the software tested, therefore indicating that a dynamic measurement would prove useful.

## 3. DEFINITION OF DYNAMIC COHESION METRICS

In this section a number of cohesion metrics are defined which are based on the static cohesion metric outlined by Chidamber and Kemerer [5, 6]. They are designed to be applied to an application at runtime and they provide a means to evaluate cohesion at the class level. The primary development language for these metrics was Java. However, they were not designed to be application specific and should be suitable for use with any object-oriented language.

The metrics are outlined using the following template:

**Definition:** A description of the metric.

**Theoretical basis:** This is an outline on the background upon which the metric is based.

**Impact:** This is an informal discussion the impact the metric has on the quality attributes of a design, for example its maintainability, reusability, error proneness, error propagation or understandability. This discussion is based on intuition rather than theoretical proofs.

### 3.1 Dynamic Cohesion Metrics

#### METRIC 1. Dynamic Simple LCOM

*Definition:*

The Dynamic Simple LCOM for a class  $A$  is the number of pairs of methods in the class that have no instance variables in common,  $|P|$ , minus the number of pairs of methods that have common instance variables,  $|Q|$  at runtime. However, if  $|P| < |Q|$ , Dynamic Simple LCOM is zero. If none of the

methods in a class display any instance behaviour, that is they do not use any instance variables, they have no similarity, and the Dynamic Simple LCOM will be zero.

For a class  $A$ , with methods  $M_1, M_2, \dots, M_n$ , let  $\{I_i\}$  be the set of instance variables used by method  $M_i$  at runtime. There are  $n$  such sets  $\{I_1\}, \dots, \{I_n\}$ .

Then let:

$$\begin{aligned} P &= \{(I_i, I_j) | I_i \cap I_j = \phi\} \\ \text{and } Q &= \{(I_i, I_j) | I_i \cap I_j \neq \phi\} \\ \text{where } P &= \phi, \text{ if } \{I_1\}, \dots, \{I_n\} = \phi \end{aligned}$$

We then define:

<b>Dynamic Simple LCOM</b>	$=$	$\begin{cases}  P  -  Q , \text{ if }  P  >  Q  \\ 0, \text{ otherwise} \end{cases}$
----------------------------	-----	--------------------------------------------------------------------------------------

*Theoretical basis:*

This metric is a translation of the LCOM measure proposed by Chidamber and Kemerer, to make it suitable for use in runtime analysis. It is a measure of the similarity of the methods and instance variables in a class at runtime. Like the Chidamber and Kemerer measure it is an inverse cohesion measure. A low value of LCOM for a class indicates that this class is cohesive. As negative values are not permitted the best possible value obtainable is zero. The higher the value of LCOM the less cohesive the class.

*Impact:*

If a class A has a high Dynamic Simple LCOM than a class B, this may indicate that class A is more complex than class B, as low cohesion increases complexity. Class A may therefore be more difficult to maintain, be susceptible to a greater number of errors and be more difficult to reuse in other applications.

One of the main objectives of object-orientated programming is to design classes that encapsulate their objects. A low Dynamic Simple LCOM value indicated the methods of a class are cohesive and this will promote the encapsulation of it objects.

A high value may indicate flaws in the design of a class and may indicate the class may need to be split into a number of smaller sub-classes.

A high Dynamic Simple LCOM value for a class indicated that it behaviour will be more difficult to predict as the class will not be functioning to perform a single function.

## METRIC 2. Dynamic Call-Weighted LCOM

*Definition:*

The Dynamic Call-Weighted LCOM is the sum of the number of accesses to instance variables from the number of pairs of methods in the class that have no common instance variables,  $\sum P$ , minus the sum of the number of accesses by the number of pairs of methods that have common instance variables,  $\sum Q$  at runtime. If  $\sum P < \sum Q$ , Dynamic Call-Weighted LCOM is zero. If no methods access any instance variables, P is equalled to null.

For a class  $A$ , with methods  $M_1, M_2, \dots, M_n$ , let  $\{I_i\}$  be the set of instance variables used by method  $M_i$  at runtime, as before.

Then let  $N_i$  be the number of times method  $M_i$  dynamically accesses instance variables from the set  $\{I_i\}$ . There are  $n$  such sets  $\{I_1\}, \dots, \{I_n\}$ .

Then let

$$\begin{aligned} P &= \{(N_i + N_j) | I_i \cap I_j = \phi\} \\ \text{and } Q &= \{(N_i + N_j) | I_i \cap I_j \neq \phi\} \\ \text{where } P &= \phi, \text{ if } \{I_1\}, \dots, \{I_n\} = \phi \end{aligned}$$

We then define:

<b>Dynamic Call-Weighted LCOM</b>	$=$	$\begin{cases} \sum P - \sum Q, \text{ if } \sum P > \sum Q \\ 0, \text{ otherwise} \end{cases}$
-----------------------------------	-----	--------------------------------------------------------------------------------------------------

*Theoretical basis:*

This measure is an extension of the Dynamic Simple LCOM, modified to take into account the number of accesses to instance variables by the methods contained within a class. The objective of this study was to determine how dynamically cohesive a class was. If two classes A and B achieved an equivalent value for dynamic Simple LCOM, but the methods of class A accessed their instance variables four times as often as those in class B, class A could be deemed to be more cohesive than class B as its methods and instance variables are more closely linked at runtime. This metric attempts to measure cohesion on a call-weighted basis.

*Impact:*

Identical to Dynamic Simple LCOM viewpoints.

## 4. EXPERIMENTAL PLATFORM

### 4.1 Execution Environment for Dynamic Metrics: The JPDA

In order to study the dynamic behaviour of Java programs at runtime it was necessary to obtain a runtime profile of the program under consideration. This was accomplished using the Java Platform Debug Architecture (JPDA). This is a multi-tiered debugging architecture contained within Sun

Microsystems j2sdk1.4.0.01. It consists of two interfaces, the Java Virtual Machine Debug Interface (JVMDI) and the Java Debug Interface (JDI) and a protocol the Java Debug Wire Protocol (JDWP). It also has two software components which tie them together - the back-end and the front-end as illustrated by Figure 1 .

The JVMDI is a programming interface implemented by the virtual machine and it provides a method of inspecting the state and controlling the execution of programs running in the Java Virtual Machine. JVMDI is a two-way interface. The JVMDI client can be notified of interesting occurrences through events. The JVMDI can query and control the application through many different functions, either in response to events or independent of them. JVMDI clients run in the same virtual machine as the application being debugged and access JVMDI through a native interface. JVMDI is the lowest layer within the Java Platform Debugger Architecture.

The JDWP defines the format of information and requests transferred between the application being profiled and the front end. The front end implements the high level JDI.

The JDI defines information and requests at the user code level. The JDI provides introspective access to a running virtual machine's state, Class, Array, Interface, and primitive types, and instances of those types. The demo/jpda directory of j2sdk1.4.0.01 contains source code for a basic program profiler that utilizes the JPDA. The trace program was modified to obtain the information necessary for this analysis.

## 4.2 Benchmarking

An important technique used in the evaluation of object systems is benchmarking. A benchmark is a black-box test, even if the source code is available [15]. In theory a benchmark consists of two elements:

- The structure of the persistent data.
- The behaviour of an application accessing and manipulating the data.

The process of using a benchmark to assess a particular object system involves executing or simulating the behaviour of the application while collecting data reflecting its performance [14]. A number of different Java benchmarks are available, the ones used for this study were the Java Grande Forum Benchmark Suite (JGFBS) [13] and specJVM98 [17].

### 4.2.1 The Java Grande Forum Benchmark Suite

A Grande application is one that uses large amounts of processing, I/O, network bandwidth or memory. The purpose of the Java Grande Forum Benchmark Suite is to act as a control for measuring and comparing alternative Java execution environments.

The benchmark suite is divided into three sections. Section one consists of a set of micro-benchmarks that measure the performance of low-level operations for example, arithmetic and math library operations, method calls and casting.

Application	Performance Attribute Measured
<i>JGFEuler</i>	This solves the time-dependent Euler equations for flow in a channel with a "bump" on one of the walls using a fourth order Runge-Kutta method.
<i>JGFMolDyn</i>	This is a translation of a Fortran program designed to model the interaction of molecular particles under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions.
<i>JGFMonteCarlo</i>	This is a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset.
<i>JGFRayTracer</i>	This measures the performance of a 3D raytracer rendering a scene containing 64 spheres.
<i>JGFSearch</i>	This solves a game of connect-4 on a 6 x 7 board using an alpha-beta pruning technique. Memory and integer intensive.

**Table 1: Applications in section three of JGFBS**

Section two consists of Kernel applications; these are short codes that carry out specific operations frequently used in Grande applications. Sections one and two are not discussed in this paper as they are considered to small to be representative of real-world programs.

Section three is made up of large-scale applications. These are real Grande codes useful for demonstrating the potential of Java for tackling real problems. A complete analysis of these programs, illustrated in Table 1, was conducted using the Java Platform Debug Architecture. The programs could be executed in two different sizes *SizeA* and *SizeB*.

### 4.2.2 The SPECjvm98 Benchmark Suite

The SPECjvm98 benchmark suite is also used to study the architectural implications of a Java runtime environment. The benchmark suite consists of eight Java programs which represent different classes of Java applications as illustrated by Table 2.

These programs were run at the command line prompt and do not include graphics, AWT (graphical interfaces), or networking. The programs could also be run with a 1%, 10% or 100% size execution by specifying a problem size s1, s10 or s100 at the command line.

## 4.3 Dynamic Cohesion Analysis

The dynamic analysis was conducted as described above. The Java programs from the JGFBS were compiled into their corresponding class file representation using Sun's javac compiler, from version 1.4.0.01 of the Java 2 SDK. The SPECjvm98 benchmarks were obtained in bytecode format. Each of the class files were executed and a dynamic profile of the program was obtained using the Java Debug Architecture which recorded all occurrences of object instantiations,

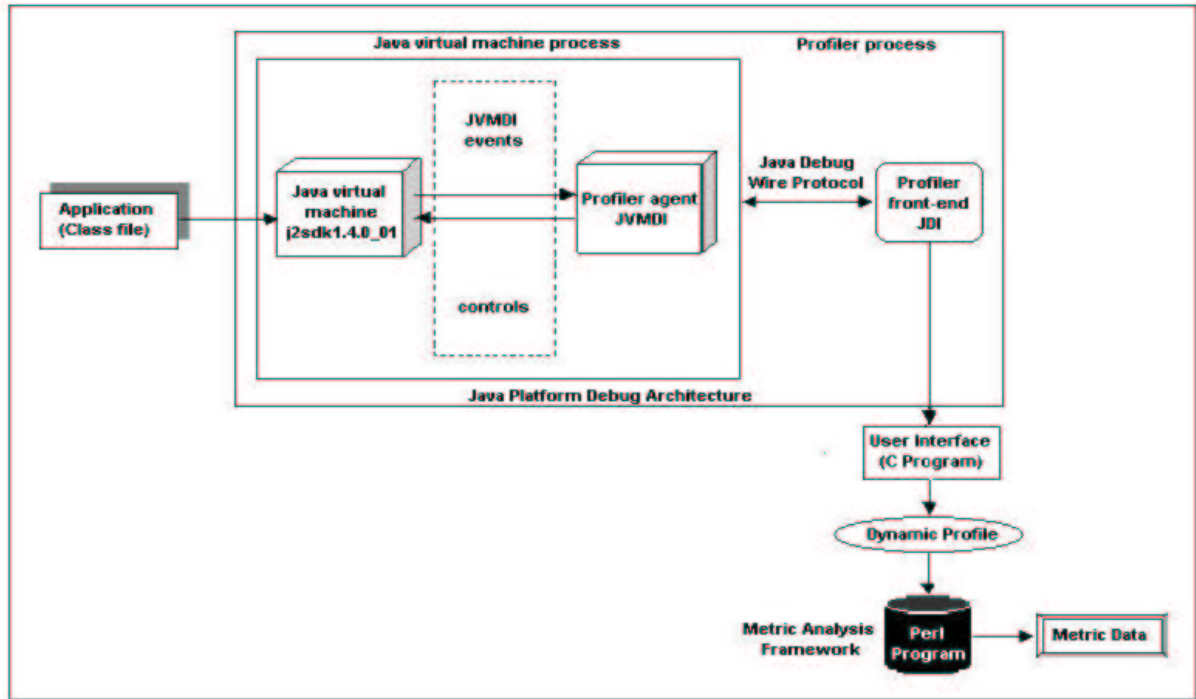


Figure 1: The Java Platform Debug Architecture and the Dynamic Analysis Framework

Application	Description
<i>.201_compress</i>	A popular modified Lempel-Ziv method (LZW) compression program.
<i>.202_jess</i>	JESS is the Java Expert Shell System and is based on NASA's popular CLIPS rule-based expert shell system.
<i>.205_raytrace</i>	This is a raytracer that works on a scene depicting a dinosaur.
<i>.209_db</i>	Data management software written by IBM.
<i>.213_javac</i>	This is the Sun Microsystems Java compiler from the JDK 1.0.2.
<i>.222_mpegaudio</i>	This is an application that decompresses audio files that conform to the ISO MPEG Layer-3 audio specification.
<i>.227_mtrt</i>	This is a variant of <i>.205_raytrace</i> . This is a dual-threaded program that ray traces an image.
<i>.228_jack</i>	A Java parser generator from Sun Microsystems that is based on the Purdue Compiler Construction Tool Set (PCCTS). This is an early version of what is now called JavaCC.

Table 2: Description of the SPECjvm98 Benchmarks

method calls and instance variable accesses.

The following information was obtained from the dynamic profile of the program:

- The total number of methods called and instance variables accessed by a class A was recorded.
- A list was compiled of each method in a class and the instance variables accessed from its own class, this forms a set for this class.
- The total number of sets was determined.

From this information the following metrics were evaluated as previously defined.

- The **Dynamic Simple LCOM**.
- The **Dynamic Call-Weighted LCOM**.

#### 4.4 Static Analysis

A static analysis of the benchmarks of the JGFBS and SPEC suites was also performed as outlined in this section. The compiled representation of a Java program, the class file format, represents a class as a stream of bytes. This is in a binary format that is readily readable by a computer but unintelligible to human beings. It was therefore necessary to convert the class files into a human readable format. A

Static Metrics	Dynamic Metrics
Static LCOM	Dynamic Simple LCOM
	Dynamic Call-Weighted LCOM

**Table 3: Cohesion Metrics Used in this study**

disassembler called Gnoo [9] was used for this purpose. Gnoo disassembled the class files into an Oolong source file, by creating a .j file from the class source file. The .j file will be written in the Oolong language, which is an assembly language for the Java Virtual Machine. This file will be nearly equivalent to the class file format but it will be suitable for human interpretation.

#### 4.4.1 Static Cohesion Metrics

The following information was obtained from the Oolong source file:

- The total amount of methods called and instance variables used by class A was noted.
- A list of each method in the class and the instance variables accessed from its own class, this forms a set for this class.
- The total number of sets was determined.

Utilizing this information the following metrics were calculated. The static cohesion metric is calculated from the above information using the procedure outlined by Chidamber and Kemerer in their definition of the Lack Of Cohesion in methods (LCOM) metric. This metric is defined as a count of the numbers of pairs of methods that do not have instance variables in common minus the count of number of pairs of instance variable that do. For the purpose of this study the metric is called **Static LCOM**.

## 5. EXPERIMENTAL RESULTS

### 5.1 Static LCOM

A static analysis of the Lack of Cohesion in Methods (LCOM) was performed for each of the classes in the benchmark under evaluation. The measure used is the LCOM metric outlined by Chidamber and Kemerer in their 1994 paper [6]. The Static LCOM is proposed to give a measure of how closely the methods contained within a class are related to the local instance variables, thereby giving a measure of the degree to which the internal processing elements of class contribute towards performing a unified function. There are a number of benefits of having highly cohesive classes present in a program. It will encourage the encapsulation of its objects. An application with highly cohesive modules will result in the increased reliability and understandability of the code. Classes that exhibit low cohesion are likely to be more complex and this will increase the probability of errors during the development process. A measure of cohesion may also aid in the identification of flaws in the design of a class [5].

The Static LCOM metric is an inverse cohesion measure. Therefore the lower the Static LCOM value obtained, the

more highly cohesive the class is considered to be. As negative values are not permitted the best possible value obtainable is zero. If a situation arises where a class has only one method that accesses instance variables from the same class, (a single set is formed for the class), the class is automatically assigned a Static LCOM value of zero. If no methods in the class under consideration access any instance variables from the same class, the Static LCOM value is again zero.

All of the programs from the JGFBS and the SPECjvm98 benchmark suites were evaluated. Due to the large volume of results collected only the static and dynamic results from JGFModyn from the JGFBS, \_201\_compress and \_209\_db from the SPECjvm98 suite are illustrated in this paper.

#### 5.1.1 JGFBS

Table 4 illustrates the findings of the static analysis for JGFModyn. Each of the classes achieved maximal cohesion, that is Static LCOM values of 0.

#### 5.1.2 SPECjvm98

Table 6 shows the results for the \_209\_db program. All of the classes were again found to be maximally cohesive.

The \_201\_compress benchmark results are shown in Table 8. Except for the Comp\_Base class, which had a Static LCOM of 1, all of the other 10 non-API classes had Static LCOM values of 0.

### 5.2 Dynamic LCOM

As was observed during the analysis of dynamic coupling, a greater number of classes are involved in the execution of an application than is evident from a simple static evaluation. It was possible to obtain LCOM values not only for the application class under consideration but also for those classes that play a role during its execution. These classes can be categorized as those belonging to Java's Application Program Interface(API). As there was a great volume of API classes involved in the execution of each application, only the results for the non-API were selected for further study.

The Dynamic LCOM metric was evaluated on two levels. The Simple LCOM method parallels the Static LCOM metric, while the Call-Weighted LCOM method took the number of method invocations into account. Again the lower the value obtained, the more cohesive the class was deemed to be. A class exhibiting an LCOM value of zero was again presumed maximally cohesive.

#### 5.2.1 JGFBS

All of the programs from section three of the JGFBS were evaluated. Because of space requirements only the results for the non-API classes in the moldyn package of JGFModyn are illustrated in Table 5, as this is a typical example of the kinds of results obtained for each benchmark.

As values of zero were obtained for each of the classes in this package this would suggest that they all exhibit maximal cohesion.

To aid in the cohesion analysis graphs were constructed for each class. Each class is represented by a square, its methods

are depicted by circles and a line joins any two methods that share common instance variables as illustrated by Figure 4 for JGFMolDyn. Similar graphs were also constructed to visualize the static results for all the classes analyzed. Even though all three classes were deemed to be maximally cohesive according to the dynamic LCOM measures, the graphs suggest that the class `md` may not be as cohesive as other two classes as all of its components are not connected. The inadequacies with these metrics is discussed further in section 5.3.

### 5.2.2 SPECjvm98

The dynamic LCOM results for two of the benchmarks `_209_db` and `_201_compress` are illustrated by Tables 7 and 9 respectively.

The dynamic results deemed all the classes in `_201_compress` to be maximally cohesive. Graphs were constructed for these classes as illustrated by Figure 8. Comparing the graphical representation of the Static LCOM results in Figure 7, it can be seen that the actual runtime situation differs from the information obtainable from a static analysis.

All of the classes in the `_209_db` package achieved dynamic LCOM values of zero except for the Database which had a Dynamic Simple LCOM of 12. However the Dynamic Call-Weighted LCOM value was again 0.

## 5.3 Inadequacies with Cohesion Metrics

As with the static definition of the LCOM metric there are a number of problems that are unaccounted for with the current definition of the dynamic cohesion metrics. Many of these arise from the fact that it is modeled from the original definition as proposed by Chidamber and Kemerer [6].

### 5.3.1 Lack of Discriminating Power

If there is only a single method in a class accessing instance variables in the same class, only one set is formed for that class. The Dynamic LCOM value for that class is thus set to zero. Also if  $|P| < |Q|$ , the number of pairs of methods having no common instance variables is less than the number of pairs of methods having common instance variables, LCOM is again set to zero. A previous study has shown that this results in a large number of classes with a LCOM of zero [1]. It can be said that the LCOM metric has little discriminating power between these classes.

Predominantly a value of zero was obtained for the classes in the 13 programs evaluated in this study for the Dynamic Simple LCOM and the Dynamic Call-Weighted LCOM metrics. Looking at the dynamic results for the `_201_compress` benchmark in Table 9 as an example. The two dynamic metrics make no attempt to distinguish between classes where only one set is formed for the class like `Code.Table` and classes with a number of sets, for example `Compressor`.

### 5.3.2 Inclusion of Access Methods

The role of an access method is typically to provide read or write access to an instance variable belonging to a class. Usually these methods will deal with a single instance variable. The result of this may be the presence of many pairs of such methods that do not have any instance variables in

common. This is a disadvantage when calculating metrics which involve counting pairs of methods that use common instance variables. Their presence may artificially decrease the value of the cohesion measure.

The cohesion value may also be artificially decreased by the presence of access methods if other methods of the class use the access method to access the instance variable instead of directly referencing it. This will result in a reduction in the number of references to that instance variable.

### 5.3.3 Inclusion of Constructors

The role of a constructor is to assign initial values to the instance variables of a class. Typically all of the instance variables in a class will be accessed by these constructor methods. The inclusion of such methods in the analysis will result in an artificial increase in cohesion, as many pairs of methods will exist that have instance variables in common.

### 5.3.4 Impact of Inheritance

The original definition of cohesion by Chidamber and Kemerer made no attempt to deal with methods and instance variables that may be inherited by a class. The Dynamic LCOM measures defined here do not account for inheritance either. This approach is deemed cohesion at the class level, that is, it deals with the "relationships between the elements of a class". The elements of a class are all of its non-inherited methods and instance variables.

A number of alternatives have been proposed to deal with methods and instance variables a class has inherited. They can be omitted or included in the analysis [3]. There is also the option proposed by Bieman and Kang [2] to include inherited instance variables but exclude inherited methods, but they provide no explanation as to why this would be a better alternative. Alternatively inherited methods could be included but inherited instance variables excluded. Briand et al stated that this option makes little sense.

Eder et al [8] distinguished between different levels of cohesion; class cohesion, which is considered here, and inheritance cohesion. Inheritance cohesion takes all inherited methods and instance variables into account as well as the non-inherited. It is therefore an extension of class cohesion that takes inheritance into account. Inheritance cohesion involves analyzing the extent to which a class represents a single semantic concept while including inherited methods and instance variables.

### 5.3.5 Distinction Between Directly and Indirectly Connected Pairs of Methods

The distinction between directly and indirectly connected pairs of methods is not addressed. If two distinct methods  $m_1$  and  $m_2$  of a class  $C$ , both access an instance variable of class  $C$ , they are said to be similar methods. They are also deemed to be directly connected to one another. If another method  $m_3$  in class  $C$  is similar to method  $m_2$  it is said to be directly connected to  $m_2$  but indirectly connected to  $m_1$ .

This metric counts direct connections only. The disadvantage of this is that it has been proposed that indirect connections appear to give a better indication for when to break up

a class [3]. Consider the following example, there is a class  $C$  with six methods  $m_1$  to  $m_6$  as illustrated in Figure 2, and a class  $D$  with methods  $m_7$  to  $m_{12}$ . Any pair of methods, which are connected by a line, are directly connected, for example  $m_1$  and  $m_2$ . Any pair of methods which are not directly connected by a line but can be reached through a path from directly connected methods are said to be indirectly connected, for example  $m_1$  and  $m_3$ .

As this metric counts the number of direct connections between methods, the same cohesion value will be obtained for both classes, as they both contain six methods and have five pairs of similar methods.

However, there is a significant difference between classes  $C$  and  $D$ . In class  $C$  every method is connected to every other method either directly or indirectly. In class  $D$  methods  $m_{11}$  and  $m_{12}$  are directly connected to one another but not to any other methods in the class. This may illustrate that these methods should not be encapsulated together in the same class, which is why indirect connections are useful when defining criteria for when to split up a class.

To obtain maximum cohesion for a class when considering direct connections only, every component of the class needs to be directly connected to every other component. According to Briand et al this appears to be an unrealistic requirement.

## 6. CONCLUSION

In this study a number of dynamic class level cohesion metrics have been proposed to assess the external quality of an object-oriented design at runtime. It is thought that measures that quantify the complexity of a design can be accurate predictors of design quality [10]. A design can be evaluated in terms of both its internal and external complexity and previous research has shown that static cohesion metrics provide a good indication of the internal complexity of a design [19]. For this reason a number of dynamic cohesion metrics were proposed which may provide an important supplement to existing static metrics.

Two complementary dynamic cohesion metrics were defined. The Dynamic Simple LCOM and Dynamic Call-Weighted LCOM were defined to quantify the internal complexity of a class at runtime.

These metrics were applied to a number of case studies involving the Java programs from section three of the Java Grande Forum Benchmark Suite and the SpecJVM98 Benchmark Suite. A static analysis of the benchmarks was also performed utilizing the LCOM metric proposed by Chidamber and Kemerer to determine the static cohesiveness of the class under analysis.

The results of this study indicate that both static and dynamic metrics can give different indications of the levels of cohesion present in a class. 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 executing object" [18].

This study has also shown how these dynamic class level metrics could be used to evaluate external quality aspects of a design by measuring the actual run-time properties of a class.

There is also some evidence to suggest that some sort of relationship may exist between the information obtained from a static and dynamic analysis. It is reasonable to postulate that there may be a correlation between the two, as static metrics evaluate the quality of a class at the code level whereas dynamic metrics quantify the situation when these classes are executed at run-time.

## Future Work

This study has mainly involved benchmark suites such as SPEC and Grande. Future work will involve:

- Widening this collection of programs to include more common "real world" Java applications. There are various technical problems to be solved here in terms of running the programs in a documented, repeatable manner, and in generating and processing the profiling information.
- Empirically validating the proposed dynamic cohesion class level metrics and their correlation with the external quality attributes of a design.
- Further investigating the correlation between static and dynamic cohesion metrics.
- Developing a set of dynamic object level cohesion metrics to measure the levels of cohesion in individual objects at runtime.

## Acknowledgments

This work is funded by the Embark initiative, operated by the Irish Research Council for Science, Engineering and Technology (IRCSET).

## 7. REFERENCES

- [1] Basili, V.R., Briand, L.C. and Melo W.L., "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, Vol. 22, no. 10, pp. 751-761, October 1996.
- [2] Bieman, J.M. and Kang, B.K., "Cohesion and Reuse in an Object-Oriented System," *Proc. ACM Symp. Software Reusability (SSR'94)*, pp. 295-262, 1995.
- [3] Briand, L.C., Daly, J.W. and Wust, J.K., "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.: An Int'l J.*, Vol. 3, no. 1 pp. 65-117, 1998.
- [4] Briand, L.C., "Empirical Investigations of Quality Factors in Object-Oriented Software," *Empirical Studies of Software Engineering*, Ottawa, Canada, March 4-5, 1999.
- [5] Chidamber, S.R. and Kemerer, C.F., "Towards a Metrics Suite for Object-Oriented Design," *Proc. Conference on Object-Oriented Programming:*

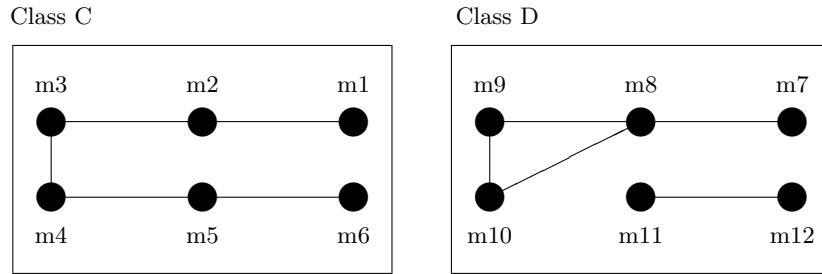


Figure 2: Illustration of the problem with not addressing the distinction between directly and indirectly connected pairs of methods in cohesion calculations

Static Cohesion: JGFMolDyn				
Class	P	Q	Static LCOM	No. Sets
md	1	5	0	4
particle	2	13	0	6
random	1	2	0	3

Table 4: Static Cohesion Results for Non-API Classes in moldyn Package Involved in Execution of JGFMolDyn from JGFBS

Dynamic Cohesion: JGFMolDyn							
Class	P	Q	Dynamic Simple LCOM	P	Q	Dynamic Call-Weighted LCOM	No. Sets
md	3	3	0	1,309,220	2,618,344	0	4
particle	1	9	0	4,072,647	18,437,309	0	5
random	1	2	0	15,493	15,499	0	3

Table 5: Dynamic Cohesion Results for Non-API Classes in moldyn Package of JGFMolDyn from JGFBS

Static Cohesion: _209_db				
Class	P	Q	Static LCOM	No. Sets
Database	120	156	0	24
Entry	0	3	0	3
Main	0	1	0	2

Table 6: Static Cohesion Results for the Non-API Classes in spec.benchmarks.\_209\_db Package Involved in Execution of \_209\_db from SPECjvm98 Benchmark Suite

Dynamic Cohesion: _209_db							
Class	P	Q	Dynamic Simple LCOM	P	Q	Dynamic Call-Weighted LCOM	No. Sets
Database	20	8	12	95,394	125,330	0	8
Entry	-	-	0	-	-	0	1
Main	-	-	0	-	-	0	1

Table 7: Dynamic Cohesion Results for Non-API Classes in spec.benchmarks.\_209\_db Package for Full Size Execution (S100) of \_209\_db from SPECjvm98 Benchmark Suite

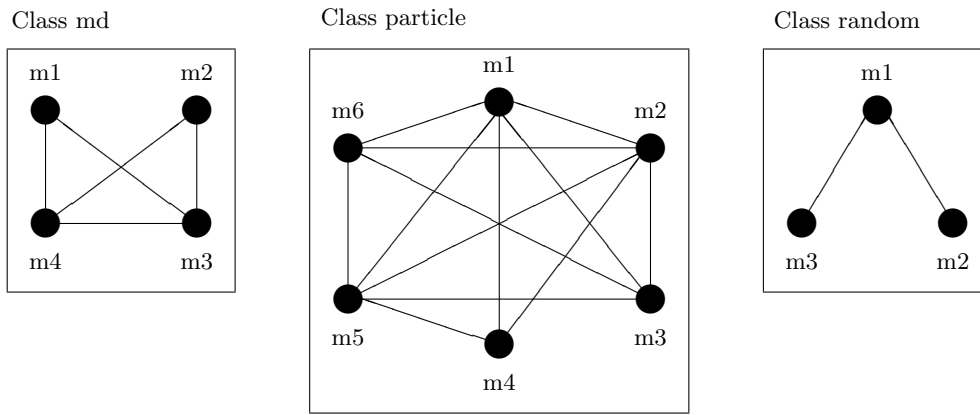


Figure 3: Diagrammatic Representation of Static LCOM in Non-API Classes involved in execution of JGF-MolDyn

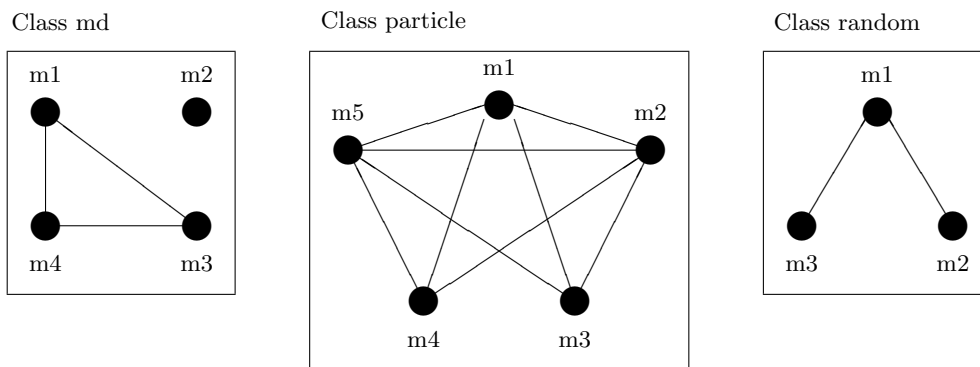


Figure 4: Diagrammatic Representation of Dynamic LCOM for Non-API Classes in execution of JGF-MolDyn

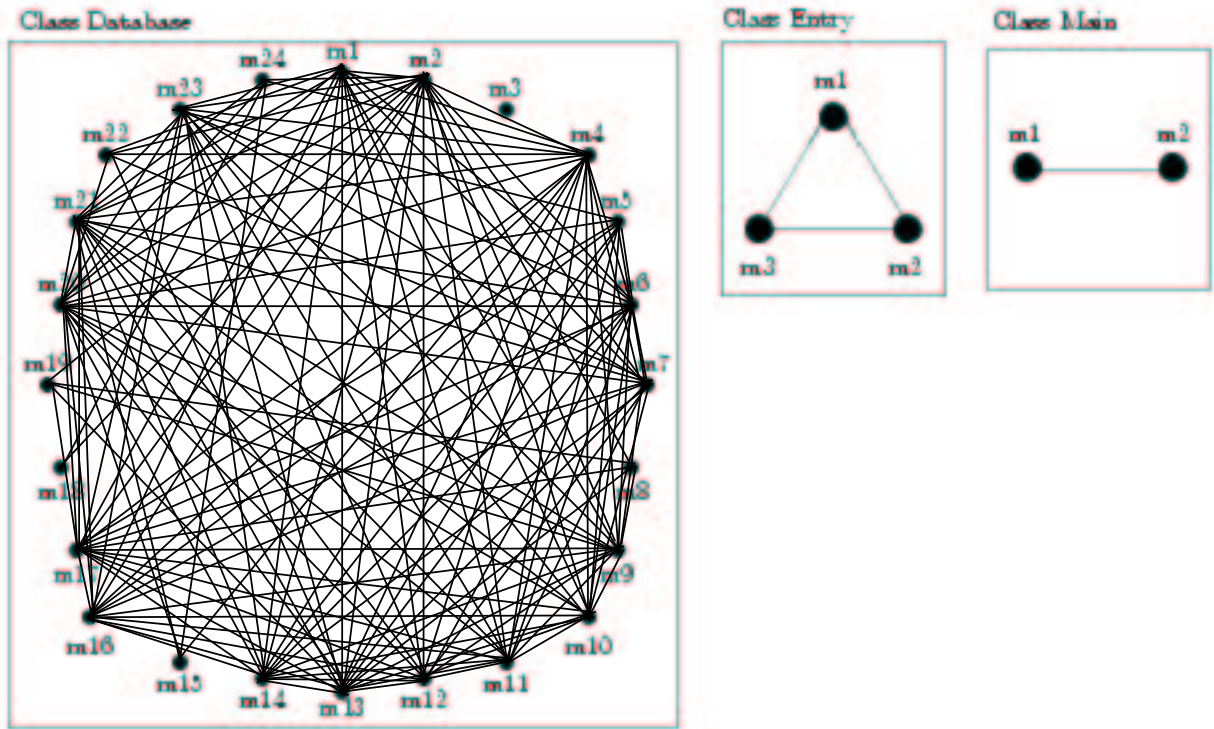


Figure 5: Diagrammatic Representation of Static LCOM for Non-API Classes involved in execution of `_209_db`

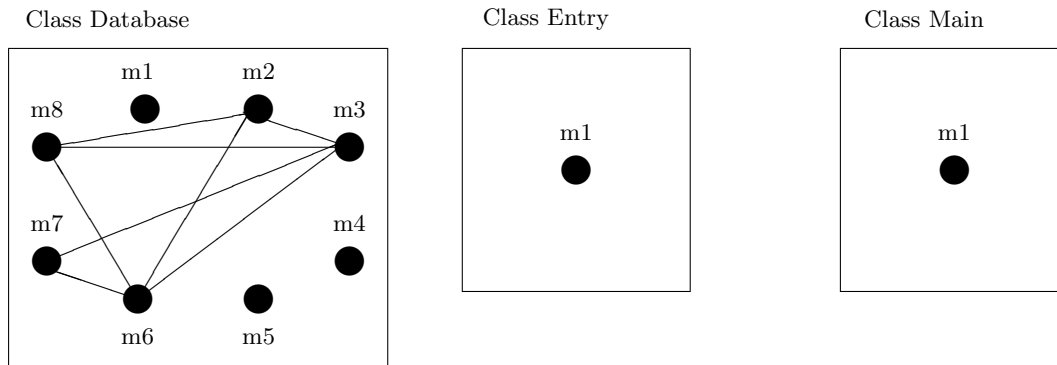


Figure 6: Diagrammatic Representation of Dynamic LCOM for Non-API Classes in execution of `_209_db`

Static Cohesion: <code>_201.compress</code>				
Class	P	Q	Static LCOM	No. Sets
Code_Table	0	6	0	4
Comp_Base	1	0	1	2
Compress	-	-	0	1
Compressor	1	5	0	4
Compressor\$Hash_Table	2	8	0	5
Decompressor	1	2	0	3
Decompressor\$De_Stack	0	6	0	4
Decompressor\$Suffix_Table	0	6	0	4
Harness	0	1	0	2
Input_Buffer	0	3	0	3
Output_Buffer	0	6	0	4

**Table 8: Static Cohesion Results for Non-API Classes in `spec.benchmarks._201.compress` Package Involved in Execution of `_201.compress` from SPECjvm98 Benchmark Suite.**

Dynamic Cohesion: <code>_201.compress</code>							
Class	P	Q	Dynamic Simple LCOM	P	Q	Dynamic Call-Weighted LCOM	No. Sets
Code_Table	-	-	0	-	-	0	1
Comp_Base	-	-	0	-	-	0	1
Compress	-	-	0	-	-	0	1
Compressor	3	3	0	5,259,167	10,518,250	0	4
Compressor\$Hash_Table	-	-	0	-	-	0	1
Decompressor	0	1	0	0	123,987	0	2
Decompressor\$De_Stack	-	-	0	-	-	0	1
Decompressor\$Suffix_Table	-	-	0	-	-	0	1
Harness	-	-	0	-	-	0	1
Input_Buffer	0	3	0	0	213,373,394	0	3
Output_Buffer	0	3	0	0	9,169,256	0	3

**Table 9: Dynamic Cohesion Results for Non-API Classes in `spec.benchmarks._201.compress` Package for Full Size Execution (S100) of `_209.db` from SPECjvm98 Benchmark Suite**

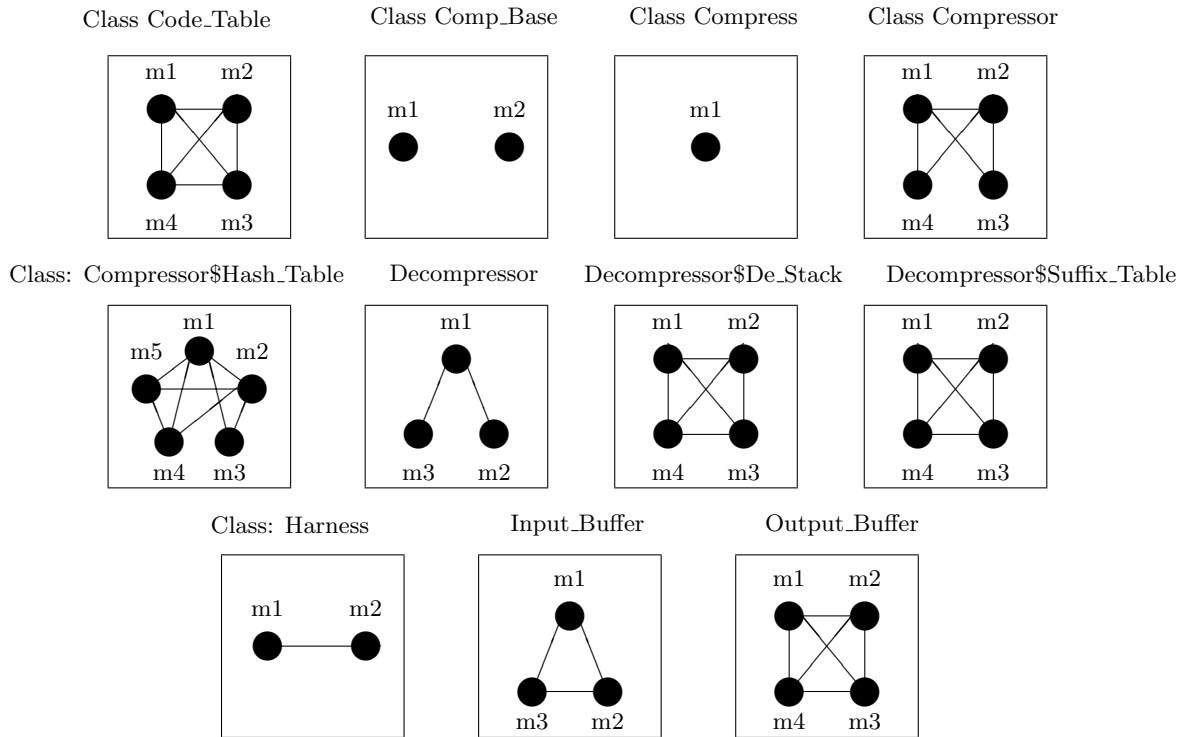


Figure 7: Diagrammatic Representation of Static LCOM for Non-API Classes in execution of `_201_compress` from SPECjvm98 Benchmark Suite

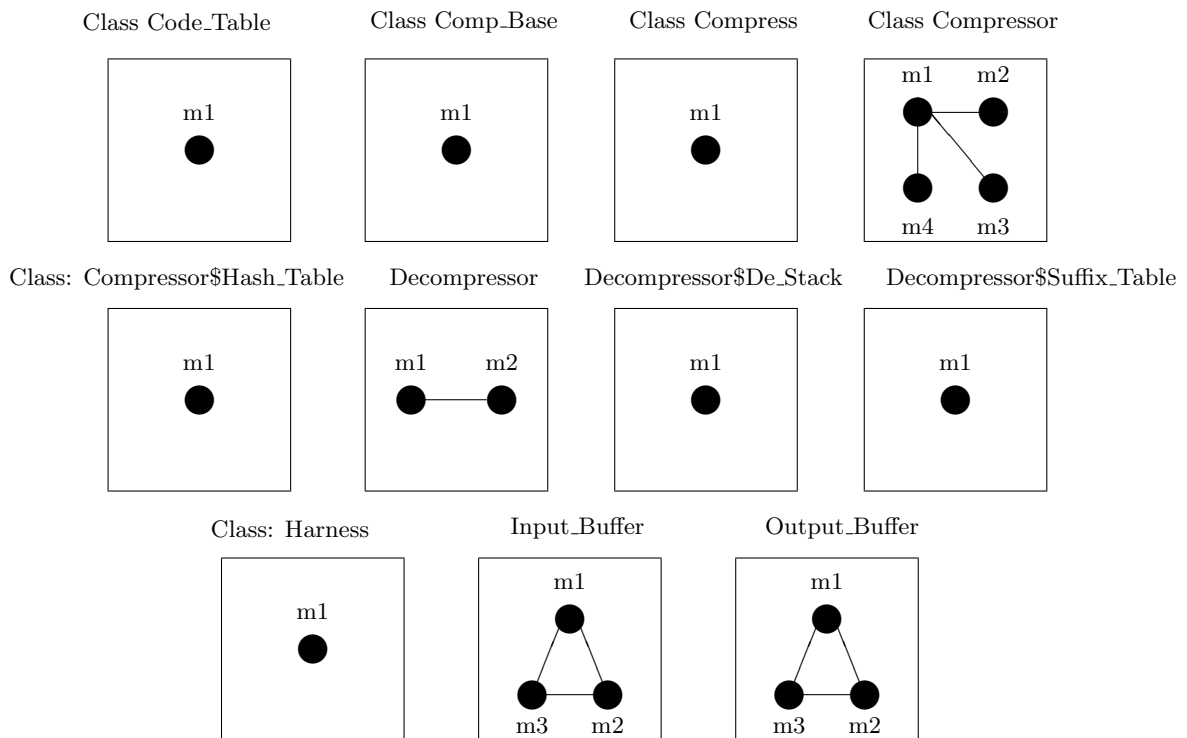


Figure 8: Diagrammatic Representation of Dynamic LCOM for Non-API Classes in execution of `_201_compress` from SPECjvm98 Benchmark Suite

*Systems, Languages and Applications* , (OOPSLA'91), SIGPLAN Notices, Vol. 26, no. 11, pp. 197–211, 1991.

- [6] Chidamber, S.R. and Kemerer, C.F., “A Metrics Suite for Object-Oriented Design,” *IEEE Transactions on Software Engineering* , Vol. 20, no. 6, pp. 467–493, June 1994.
- [7] Cleland-Hunang J., Chang C.K., Kim H. and Balakrishnan A. “A Requirements-Based Dynamic Metric in Object-Oriented Systems” *IEEE Proceedings on 5<sup>th</sup> International Symposium on Requirements Engineering* , pp. 212–219, 2001.
- [8] Eder J., Kappel G. and Schrefl M. “Coupling and Cohesion in Object-Oriented Systems” *Technical Report* ,University of Klagenfurt, 1994.
- [9] Engel, J., “Programming for the Java Virtual Machine,” Addison-Wesley, 1999.
- [10] Fenton, N.E. and Neil M., “Software Metrics: Successes Failures and New Directions,” *The Journal of Systems and Software* , Vol. 47, pp. 149–157, 1999.
- [11] Gosling, J., Joy, B. and Steele, G. “The Java Language Specification,” Addison-Wesley, Reading, MA, 1996.
- [12] Gupta, N. and Rao, P. “Program Execution Based Module Cohesion Measurement,” *16th International Conference on Automated on Software Engineering (ASE '01)* , San Diego, USA, November 2001.
- [13] Java Grande Forum Benchmark Suite, Available at the following WWW site:  
<http://www.epcc.ed.ac.uk/research/javagrande/-benchmarking.html>.
- [14] Humphries, T.O., Klauser, A., Wolf, A.L. and Zorn, B.G. “An Infrastructure for Generating and Sharing Experimental Workloads for Persistent Object Systems,” *Software-Practice and Experience* , Vol. 30, pp. 387–417, 2000.
- [15] Mehlitz, P.C., “Performance Analysis of Java Implementations”, Available at the following WWW site:  
<http://www.transvirtual.com/presentations/-speed/index.html>.
- [16] Shooman, M.L., “Software Engineering: Design, Reliability and Management,” , McGraw Hill, New York, pp. 150–151, 1983.
- [17] SpecJVM98, Available at the following WWW site:  
<http://www.spec.org/org/jvm98>.
- [18] Yacoub, S.M., Ammar, H.H. and Robinson, T., “Dynamic Metrics for Object-Oriented Designs,” *Software Metrics Symposium* , Boca Raton, Florida, USA, pp. 50–61, Nov 4–6, 1999.
- [19] Yourdon, E. and Constantine, L.L., “Structured Design,” , Prentice Hall, Englewood Cliffs, NJ, 1979.