National University of Ireland, Maynooth
Maynooth, Co. Kildare, Ireland.

Department of Computer Science
Technical Report Series

NUIM-CS-TR-2003-07

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

Aine Mitchell and James Power

http://www.cs.may.ie/ Tel: +353 1 7083847 Fax: +353 1 7083848

Run-time Coupling 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

Keywords
Dynamic analysis, software metrics, coupling

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. Coupling 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 this static measure only captures certain underlying dimensions of coupling. 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 access the runtime quality may aid in the analysis of software quality.

This paper describes new dynamic class level coupling metrics suitable for the runtime evaluation of a program. It characterizes the ability of these metrics to evaluate the external quality attributes of a design. These dynamic coupling 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.

Technical report
NUIM-CS-TR2003-07

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

Software metrics measure different aspects of software complexity and therefore play an important role in analyzing and improving software quality [1, 3]. 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 [9]. 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 [17]. For this reason this paper defines dynamic metrics that quantify the dynamic complexity of object-oriented software.

The dynamic complexity of a design may be quantified by measures that assess its internal quality. Many metrics have been proposed to statically evaluate the quality of a software design. One such measure is coupling.

Coupling has been proposed as one of the fundamental qualitative measures of the goodness of a software design or implementation [15]. It was originally defined as the degree of interdependency between modules [18]. Coupling is said to quantify the external complexity of a class, which is how dependent a class is on other classes. It is considered desirable to have a low level of coupling present within the classes of an object-oriented program. The theory behind this measure contributes to the external quality attributes of a software application, such as its maintainability, reusability and testability. A number of coupling metrics have been designed for use in different stages of the software-lifecycle [2].

Briand et. al. have carried out a through survey of the literature regarding coupling [2]. They concluded that no measures for dynamically evaluating coupling is currently available.

The remainder of this paper is organized as follows. Section 2 describes previous work that has been conducted on coupling metrics. Section 3 outlines a number of new dynamic coupling 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

2.1 Static Coupling Metrics

Chidamber and Kemerer defined a static coupling metric for object-oriented applications known as Coupling Between Objects(CBO). They then related the metric to the maintenance, testing and understandability of a design.

The CBO measure was originally defined as "a count of the number of non-inheritance related couples with other classes". Two objects are deemed to be coupled if they act upon one another, in other words, if an object of one class uses the methods or instance variables of the other [5]. If a method declared in one class uses a method or instance variable in another class, this pair of classes are said to be coupled since all objects instantiated from the same class are deemed to have the same properties.

However Chidamber and Kemerer later revised their definition of CBO. For a class C, CBO is a measure of the number of other classes to which it is coupled [6]. They amended their previous definition to include coupling due to inheritance, but they provided no explanation for this.

2.2 Current Research in Dynamic Coupling Metrics

Briand *et.al.* [2] carried out an extensive survey of the current available coupling literature in object–oriented systems and concluded that all the current metrics measured coupling at the class level (static analysis). No measures of object level coupling had been proposed (dynamic analysis). They suggested that the reason for this is the obstacle of determining the degree of coupling or cohesion between 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 coupling or 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].

Yacoub et.al. proposed a suite of dynamic metrics concerned with evaluating the quality of a design during the early development phase [17]. This suite contained two metrics for determining coupling between objects, Import Object Coupling (IOC) and Export Object Coupling (EOC).

These measures were obtained at an early development phase from executable object-oriented design models, which were used to model the application to be tested. They are both based on execution scenarios, that is "the measurements are calculated for parts of the design model that are activated during the execution of a specific scenario triggered by an input stimulus." The scenarios were then extended to have an application scope.

$\mathrm{EOC}_x(o_i\ ,\ o_j)$ is defined as the export coupling for object o_i with respect to object o_j, and is the percentage of the number of messages sent from o_i to o_j with respect to the total number of messages exchanged during the execution of scenario x. A higher EOC value indicates that a given object o_i is more tightly coupled to an object o_j and is therefore more difficult to maintain, understand and reuse and is also more error prone.

$\mathrm{IOC}_x(o_i\ ,\ o_j)$ is defined as the import coupling for object o_i with respect to object o_j, and is the percentage of the number of messages received by object o_i sent by o_j with respect to the total number of messages exchanged during the execution of scenario x. A higher IOC value indicates that an object will also be difficult to understand and maintain, however it will be more likely to be reused as the other object is dependent on it. A property of these metrics is that $\mathrm{EOC}_x(o_i\ ,\ o_j) = \mathrm{IOC}(o_j\ ,\ o_i)$.

However these metrics in themselves do not give an accurate representation of the actual runtime situation as they are evaluated during the early design stage of a program.

3. DEFINITION OF DYNAMIC COUPLING METRICS

In this section a number of coupling metrics are defined which are based on the static coupling 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 class level coupling. 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 Coupling Metrics

METRIC 1. **Dynamic CBO for a class A**

Definition:

The Dynamic CBO for a class A is a count of the number of couples with other classes at runtime.

Theoretical basis:

This metric is a direct translation of the CBO metric proposed by Chidamber and Kemerer, except it is examining how many classes a class A is coupled to at runtime.

Impact:

The greater the value obtained for this metric the greater the number of classes a class A will require to function. This will discourage the reuse of this class, as it will be not be self-sufficient. It is desirable that classes be as independent as possible to promote reuse.

This metric may provide an indication of how rigorously an implementation may need to be tested. The greater the level of coupling present, the more rigorous testing needs to be.

Maintenance is made more difficult the greater the Dynamic CBO is for a class, as the class will be more sensitive to changes in other classes the more couples there are.

METRIC 2. **Degree of Dynamic Coupling between a class A and class B at runtime**

Definition:
The Degree of Dynamic Coupling between two classes A and B, is a count of the amount 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.

Degree of Dynamic Coupling from a class A to class B =

$$\frac{\text{No. of times a class A accesses methods or instance variables from a class B at runtime}}{\text{Total no. of times a class A accesses any methods or instance variables}} * \frac{100}{1}$$

Theoretical basis:
In the definition of the static CBO, two classes are said to be coupled if one class uses the methods or instance variables contained in the other. Determining the level of coupling between these two classes is a matter of counting the number of times one class uses a method or instance variable from the other, relative to the total number of times it accesses any method or instance variable. This will give a measure of the Degree of Static Coupling between two classes.

This can be translated to the dynamic state, that is, what is happening at runtime. A class A is deemed to be dynamically coupled to a class B if at runtime, it accesses methods or instances variables from class B.

Impact:
The degree of coupling between two classes at runtime is useful to know as it reflects how dependent one class is on another. If this percentage is very high for a given class, this means that this class is heavily reliant on the other class to function, in other words it measure how tightly two classes are coupled together.

If a class A has a high value for dynamic degree of coupling to a class B this will discourage the reuse of the class A, as it will be tightly dependent on that class B and frequently request methods or instance variables from that class.

A class that is tightly coupled to another class will also be more difficult to maintain.

Errors are more likely to propagate between classes that are tightly coupled together.

A class that is tightly coupled to another class is more difficult to understand as its dynamic behaviour will be dependent on the other class.

METRIC 3. **Degree of Dynamic Coupling within a given set of classes**

Definition:

For a given set of classes c_1, c_2,...., c_n =

$$\frac{\text{Sum of number of accesses to methods or instance variables outside each class}}{\text{Sum of total no. of accesses from these classes}} * \frac{100}{1}$$

Theoretical basis:
This metric is an extension of metric 2, to indicate the level of Dynamic Coupling occurring within a given set of classes.

Impact:
For a given set of classes this metric will determine to what extent they are as a group accessing methods or instance variables outside of there own classes. This metric could be used, for example to test how reliant a group of classes are on one another. If this value is low, very few classes will be accessing methods or instance variables outside of their own class.

This metric is useful if you want to compare groups of classes,

for example in an application.

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 [14]. 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 [13]. A number of different Java benchmarks are available, the ones used for this study were the Java Grande Forum Benchmark Suite (JGFBS) [12] and specJVM98 [16].

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 maths library operations, method calls and casting.

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 Coupling Analysis

The dynamic analysis was conducted as described above. The Java programs from the JGFBS were complied 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, method calls and instance variable accesses.

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

- The total number of objects created and methods called during the execution of a Java program.
- The total number of classes that exhibited coupling during the execution of the application was recorded.
- For each of these classes the amount of other classes to which it was coupled was noted.

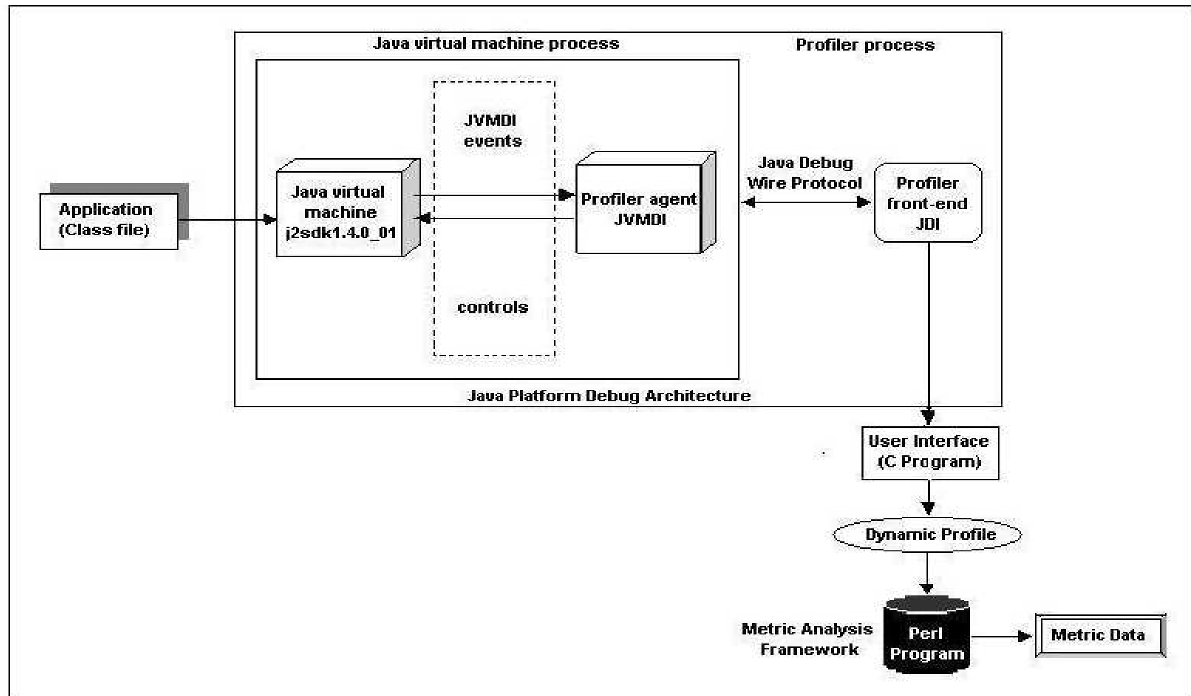


Figure 1: The Java Platform Debug Architecture and the Dynamic Analysis Framework

Application	Performance Attribute Measured
JGFEuler	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.
JGFMolDyn	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.
JGFMonteCarlo	This is a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset.
JGFRayTracer	This measures the performance of a 3D raytracer rendering a scene containing 64 spheres.
JGFSearch	This solves a game of connect–4 on a 6 x 7 board using a alpha-beta pruning technique. Memory and integer intensive.

Table 1: Programs in the Java Grande Forum Benchmark Suite, Section 3

Application	Description
_201_*compress*	A popular modified Lempel–Ziv method (LZW) compression program.
_202_*jess*	JESS is the Java Expert Shell System and is based on NASAs popular CLIPS rule-based expert shell system.
_205_*raytrace*	This is a raytracer that works on a scene depicting a dinosaur.
_209_*db*	Data management software written by IBM.
_213_*javac*	This is the Sun Microsystem Java compiler from the JDK 1.0.2.
_222_*mpegaudio*	This is an application that decompresses audio files that conform to the ISO MPEG Layer–3 audio specification.
_227_*mtrt*	This is a variant of _205_raytrace. This is a dual–threaded program that ray traces an image.
_228_*jack*	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: Programs in the SPEC JVM98 Benchmark Suite

- For each coupled pair, a record was made of the total number of methods called and instance variables accessed from one class by the other. The direction of coupling was also noted.

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

- The **Dynamic Coupling Between Objects (CBO)** was evaluated.
- The **Degree of the Dynamic Coupling** between each pair of classes was calculated.
- For the all the API and non-API classes that exhibited coupling the **Degree of Dynamic Coupling within a set of classes** was determined.

4.4 Static Analysis

A static analysis of the benchmarks of the JGFBS and SPEC suites was also performed. 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 disassembler called Gnoloo [8] was used for this purpose. Gnoloo 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 suitable for human interpretation.

4.4.1 Static Coupling Metrics

The following information was obtained from the Oolong source file:

- The number of couples a class A made with other classes was recorded (where calling a method or using an instance variable constitutes coupling). This included coupling due to inheritance.
- The total amount of methods called and instance variables used by class A was noted.
- For each of the coupled pairs found, the amount of calls and accesses made by class A from its coupled partner was noted.

Utilizing this information the following metrics were calculated.

A measure was recorded of the number of couples a class made with other classes. This measure is identical to the Coupling Between Classes (CBO) metric proposed by Chidamber and Kemerer [6]. For the purpose of this study this measure was called the Static CBO for a class.

The strength of the coupling relationship between pairs of classes, for example two classes A and B, was determined by expressing the total number of methods called and instance variables used by A from B as a percentage of the total number of call and accesses made by A. This metric was called the Degree of Static Coupling between two classes.

In order to determine the total amount of coupling present in a class the Degree of Static Coupling within a group of classes was evaluated. This was a measure of a sum of the total number of methods called and instance variables accessed by a class A from outside of its own class as a percentage of the total number of methods called and instance variables accessed by class A.

5. EXPERIMENTAL RESULTS

5.1 Program Size

The 'size' of a program is also thought to contribute to its overall complexity [4, 10]. A dynamic measure of the 'size' of each of the benchmarks under evaluation was obtained by recording the total numbers of objects created and methods called while each Java program was executing on the JVM. Tables 3 and 4 contain a list of these results.

JGFBS SizeA		
Application	**Number of Objects created**	**Number of Method Calls**
JGFEuler	3,765,267	9,041,022
JGFMolDyn	7,997	455,376
JGFMonteCarlo	276,320	35,499,401
JGFRayTracer	660,440	26,152,010
JGFSearch	3,339	33,378,256

Table 3: Program Size data for the JGFBS, Size A

SPECjvm98						
Application	**Number of Objects created**			**Number of Method Calls**		
	Size			Size		
	s1	s10	s100	s1	s10	s100
_201_*compress*	8,834	9,032	8,902	17,163,803	15,966,749	17,822,835
_202_*jess*	85,264	189,698	1,846,358	635,440	6,077,035	23,333,274
_205_*raytrace*	552,326	1,085,055	1,695,963	5,772,110	19,559,308	25,577,699
_209_*db*	13,520	176,459	3,554,259	136,726	2,299,380	34,183,241
_213_*javac*	64,081	382,060	1,553,455	443,790	3,319,026	14,050,905
_222_*mpegaudio*	15,824	18,035	15,215	1,185,653	9,368,543	21,452,712
_227_*mtrt*	551,114	1,567,812	1,955,785	5,758,391	23,772,645	25,856,949
_228_*jack*	484,952	953,582	4,009,120	4,007,716	7,921,292	33,064,049

Table 4: Program Size data for SPEC JVM98, sizes 1, 10 and 100

5.1.1 JGFBS

Table 3 illustrates the results for the programs from section three of JGFBS. The program JGFEuler exhibited the highest number of objects created while JGFSearch had the lowest. The numbers ranges from 3,765,267 to 3,339. JGFMonteCarlo had the greatest number of method calls while JGFMolDyn had the lowest. The number of method calls ranged from 35,499,401 to 455,376.

5.1.2 SPECjvm98

Looking at the Specjvm98 benchmarks in Table 4, programs like mtrt and raytracer exhibited a much higher degree of object creation and method calls than a class like mpegaudio. Based on this very crude measure the program mtrt could be considered more complex than mpegaudio.

Each of the SpecJVM98 benchmarks can be executed at one of three different problem sizes S1, S10 and S100. Problem size S1 is intended as a quick checkout of the benchmark programs on a JVM. It runs the minimum amount of work necessary to verify that the program runs. Problem size S10 requires approximately one tenth the execution time of the size S100 which is the size to use when generating reportable results. The programs were analyzed using each of the problem sizes. As expected generally the level of object creation and method calls increased as the problem size was increased from S1 to S100.

The metrics were evaluated for all of the programs in the JGFBS and SPECjvm98 benchmark suites, however due to the large volume of results collected only those from a selected few programs are illustrated in this paper.

Static and Dynamic CBO: JGFMolDyn		
Class	**Static CBO**	**Dynamic CBO**
JGFMolDynBenchSizeA	3	4
jgfutil.JGFInstrumentor	6	6
jgfutil.JGFTimer	5	8
moldyn.JGFMolDynBench	6	5
moldyn.md	5	7
moldyn.particle	3	2
moldyn.random	2	2

Table 5: Static and Dynamic CBO for non-API classes during execution of JGFMolDyn from section three of JGFBS.

5.2 Static and Dynamic CBO

A static analysis of each of the benchmarks under evaluation was performed. The Static CBO metric used in this study is equivalent to the CBO metric as outlined by Chidamber and Kemerer in their 1994 paper [6]. This metric is deemed to illustrate the dependencies between classes at the source code level. A high value for this metric signifies that at the code level a class relies on many other classes to function, through calling methods and accessing instance variables from them.

The Dynamic CBO metric was evaluated for the same set of benchmarks. The metric is a count of the number of classes a class is coupled to during runtime, therefore it depicts the actual class dependencies at runtime.

5.2.1 JGFBS

Static and Dynamic CBO: JGFRayTracer		
Class	**Static CBO**	**Dynamic CBO**
JGFRayTracerBenchSizeA	3	4
jgfutil.JGFInstrumentor	6	5
jgfutil.JGFTimer	5	1
raytracer.Interval	1	1
raytracer.Isect	1	1
raytracer.JGFRayTracerBench	7	5
raytracer.Light	2	2
raytracer.Primitive	3	4
raytracer.Ray	3	2
raytracer.RayTracer	12	14
raytracer.Scene	2	3
raytracer.Sphere	6	5
raytracer.Surface	3	2
raytracer.Vec	3	5
raytracer.View	1	1

Table 6: Static and Dynamic CBO for non-API classes during execution of JGFRayTracer from section three of JGFBS.

Tables 5 and 6 shows the Static CBO values obtained for the non-API classes involved in the execution of two of the programs from section three of the JGFBS, JGFMolDyn and JGFRayTracer.

Looking at the JGFMolDyn results, the Static CBO values for this group of classes range from 2 to 6. The JGFRayTracer results show as much wider distribution of values ranging from 1 for the class Interval, to 12 for the class RayTracer.

A greater volume of classes are involved in the execution of a program than is indicted by a simple static analysis. The Dynamic analysis provided information on the CBO values between all of the classes utilized during the execution of a Java file (API and non-API). As there was a huge number of classes involved, only CBO values for the non-API classes are illustrated by Tables 5 and 6.

There was a range of different Dynamic CBO values for the classes considered. The values for the JGFMolDyn classes ranged from 2 to 8, while there was a distribution of 1 to 14 for JGFRayTracer.

The results across all the benchmarks showed that a static and dynamic analysis of a program dose not produce equivalent results, suggesting that it may be worth while conducting both types of analysis.

5.2.2 SPECjvm98

Tables 7 and 8 illustrate the Static and Dynamic CBO values for the non-API classes in the 'main' package of _205_raytrace and _209_db programs. Again the Dynamic CBO results differed in some cases from the static results.

5.2.3 Problem with Inheritance and Dynamic CBO Metric

Static and Dynamic CBO: _205_raytrace		
Class	**Static CBO**	**Dynamic CBO**
CacheIntersectPt	1	1
Camera	1	1
Canvas	8	2
Color	2	1
Face	1	1
IntersectPt	7	9
Light	1	1
LightNode	1	1
LinkNode	1	1
Main	4	5
Material	1	1
MaterialNode	1	1
ObjNode	1	1
ObjectType	3	4
OctNode	8	9
Point	1	4
PolyTypeObj	6	8
PolygonObj	4	4
Ray	5	2
RayTracer	6	6
Runner	3	3
Scene	31	31
SphereObj	7	6
TriangleObj	4	4
Vector	2	4

Table 7: Static and Dynamic CBO for non-API classes in spec.benchmarks._205_raytrace package during full size execution of _205_raytrace program from SPEC JVM98 Benchmark Suite.

Some of the classes evaluated exhibited a greater value for the Dynamic CBO metric than the Static CBO. However it is not possible for a class to be coupled to a greater number of different classes at runtime than it is statically. The reason for these results have to do with the impact inheritance has on the coupling metrics.

Looking at Table 5 the classes JGFMolDynBenchSizeA, jgfutil.JGFTimer and moldyn.JGFMolDynBench all had higher Dynamic than Static CBO values. However all these classes were statically coupled to the java/lang/Object class. By default all classes in Java are sub-classes of this class, which means they will inherit all the public methods and instance variables from this class. Taking the class JGFMolDynBenchSizeA as an example, it was found to be dynamically coupled to the class java/lang/ClassLoader. However this class is a descendant of the java/lang/Object class, therefore it will possess all the methods and instance variables of this class. This illustrates a potential deficiency with this metric. If a class A statically calls methods or accesses instance variables from a class B, but dynamically makes accesses to a number of child classes of B, should the accesses to the child classes be considered as accesses to the same class as they are all descendants of the class B? The impact of Inheritance on Coupling is discussed further in Section 5.3.3.

Static and Dynamic CBO: _209_db		
Class	**Static CBO**	**Dynamic CBO**
Database	13	12
Entry	3	4
Main	11	9

Table 8: Static and Dynamic CBO for classes in spec.benchmarks._209_db package during execution of _209_db from SPEC JVM98 Benchmark Suite.

5.2.4 Conclusions

Overall the Dynamic CBO metric provides a crude measure of the dynamic dependencies of a class. Both the Static and Dynamic CBO metrics take a binary approach to coupling in that two classes are either coupled or they are not. If these metrics were to be use as indicators of the maintainability, reusability and testability of a class they may give skewed results as they do not make use of all the available information. Quality factors of a design are also likely to be influenced by the number of connections between coupled classes and not just the number of classes to which it is coupled. For example a class which is loosely coupled, that is has few connections, to say five other classes may be easier to maintain than a class that has many connections or is strongly coupled to only two other classes. The Degree of Static or Dynamic metrics endeavors to take the number of connections into account for this reason.

5.3 Degree of Static and Dynamic Coupling

It may also be useful to know the strength of the coupling that exists between two classes, as this will illustrate how often a class is calling methods or accessing instance variables from the class that it is coupled to. The Degree of Static or Dynamic Coupling metric illustrates this.

5.3.1 JGFBS

This static metric was calculated for the classes involved in the execution of JGFMolDyn and the results are illustrated in Figure 3. It is evident from the graph that the 'importing classes', which are illustrated on the left hand side of legend, are not coupled to the same degree to each of their 'exporting' classes on the right. Taking the class JGFTimer as an example, it is coupled to java/lang/Object to a degree of 1.15%, which is relatively slight in comparison to the 51.72% level of coupling with java/lang/StringBuffer.

Figure 4 depicts the dynamic dependencies of the non-API classes during the execution of JGFMolDyn. From the graph it can be seen that the strength of the coupling relationship between two classes at runtime can not be accurately reveled from a static analysis.

Looking at the static and dynamic results for the class moldyn.particle. This class exhibits static couplings to the classes java/lang/Object and java/lang/Math to a degree of 0.82%, and also to moldyn/md to a degree of 14.05%. The dynamic findings show the moldyn/particle class was again coupled to java/lang/Object however it was to a degree of 0.015%. It also had a much stronger coupling to moldyn/md with 57.41%. However it exhibited no dynamic coupling with class java/lang/Math.

5.3.2 SPECjvm98

The same metrics were also evaluated for the programs from the SPECjvm98 benchmark suite. The static results for the non-API classes involved in execution of _209_db program are illustrated by Figure 5. While the dynamic results are shown by Figure 6.

Under execution of the program the database class had its strongest coupling to java/util/Vector with 48.68% where with the static analysis this value was 7.25%. It also showed a strong coupling to java/util/Vector$1 with 26.81% and 24.31% to java/lang/String, the static results suggested no coupling to java/util/Vector$1 and only a 4.83% coupling to the latter. The Entry class showed a strong coupling to java/util/Vector, as it did in the static analysis, and java/util/Vector$1 with 41.13% and 27.13% respectively. The main classes tightest couplings were with java/lang/StringBuffer to a degree of 19.05%, and the Database class with 14.29%. The static results suggested a higher coupling with Database with a value of 20.98%.

These results show that while the static results can give an indication of the Degree of Dynamic Coupling of a class, the actual runtime situation does show variation.

Variations in the Degree of Dynamic Coupling results were also observed under the other size executions of the _209_db program, 10% (S10) and 1% (S1). [Not illustrated here]

5.3.3 Problems with Inheritance and Degree of Dynamic Coupling Metric

The question of whether the invocation of an inherited method or instance variable constitutes a separate coupling has to be addressed?. In this study it seemed wise to consider separately accesses to methods and instance variables that have been inherited from a parent class. This can be observed from the results for the class moldyn.md from Figure 4. It was found that this class was exhibiting a dynamic coupling to the class java/text/DecimalFormat and java/lang/ClassLoader, but exhibited no such coupling in the static analysis. Looking at the diagram for the class hierarchy in the API classes depicted in Figure 2, it can be seen that the class java/text/DecimalFormat is a sub class of the class java/text/NumberFormat. This means that this class possesses all the public methods and instance variables of the parent class. Therefore it is perfectly plausible that at runtime the class moldyn/md could be using methods or instance variables that have been inherited from the parent class. A similar situation exists for the dynamic coupling with the java/lang/ClassLoader class. This class is a descendent of java/lang/Object class. Such information can not be obtained from a simple static analysis and may also be useful in the potential redesign of a class. However the question of whether this type of relationship really constitutes a coupling merits further investigation.

5.4 Degree of Static and Dynamic Coupling within a set of classes

It may also be desirable to determine what is the overall level of static and dynamic coupling within a set of classes. For

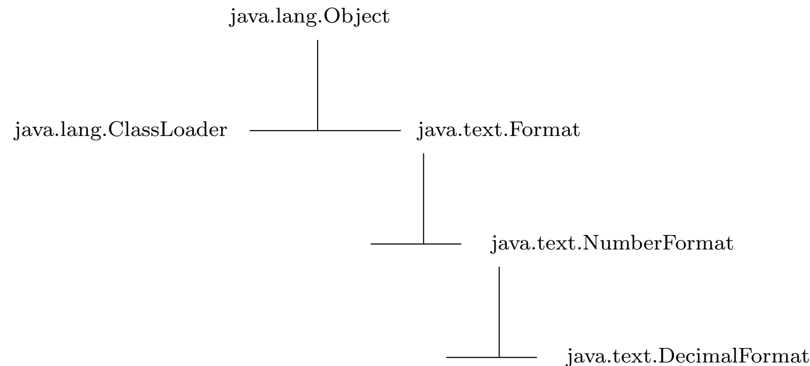


Figure 2: An example of the depth of inheritance possible with the Java API

a given set of classes this metric is a measure of the number of calls and accesses outside of the classes as a function of the total number of calls or accesses. This metric gives a means of comparing the level of coupling present in different programs.

5.4.1 *JGFBS*

This metric was calculated for the non-API classes for all programs in section three of the JGFBS as illustrated by Figure 7. The static results ranged from 17.53% for JGFMolDyn to 48% for JGFRayTracer. However, the dynamic results only spanned a range from 11.98% for JGFMolDyn to 19.95% for JGFEuler. It can be seen from the graph that the level of dynamic coupling present with the sets of classes was approximately half of that which was suggested by the static results. This would suggest that the static results may give a misleading indication of the actual level of coupling present within an application.

5.4.2 *SPECjvm98*

Table 8 presents the results for the SPECjvm98 benchmark. Again the overall level of coupling exhibited for the programs evaluated differed in the static and dynamic analysis..

6. CONCLUSION

In this study a number of dynamic class level coupling 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 [9]. A design can be evaluated in terms of both its internal and external complexity and previous research has shown that static coupling metrics provide a good indication of the external complexity of a design [18]. For this reason a number of dynamic coupling metrics were proposed which may provide an important supplement to existing static metrics.

Three complementary dynamic coupling metrics were defined. The Dynamic CBO and Degree of Dynamic Coupling between classes metrics were defined to quantify the external complexity of a class at runtime. The Degree of Dynamic Coupling within a set of classes was proposed to determine the external complexity within a group of classes.

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 CBO metric proposed by Chidamber and Kemerer to determine the level of static coupling.

The results of this study indicate that both static and dynamic metrics can give different indications of the levels of coupling present in a class. The reasons for the different results obtained form the static and dynamic analysis may arise form 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" [17].

This study has also shown how this 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:

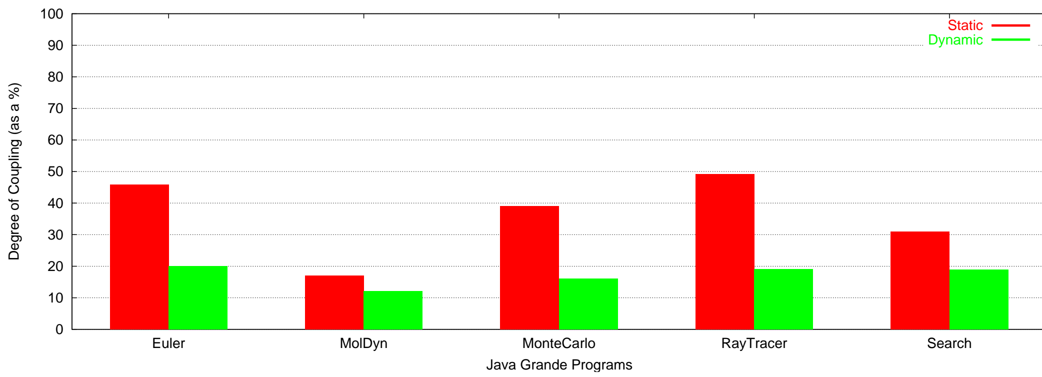


Figure 7: Degree of Static and Dynamic Coupling within a group of classes for the non-API classes of the programs from section three of the JGFBS.

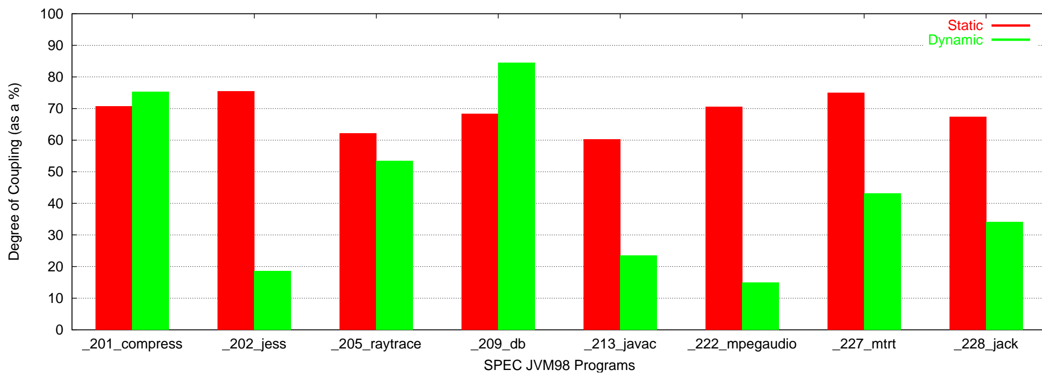
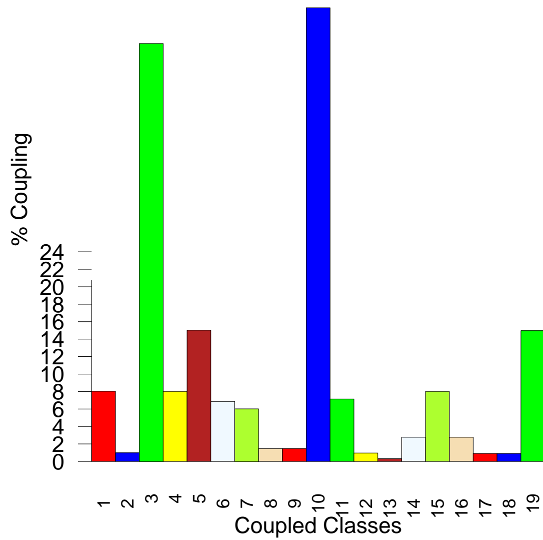


Figure 8: Degree of Static and Dynamic Coupling within a group of classes for the non-API classes of the programs from SPECjvm98 benchmark suite.

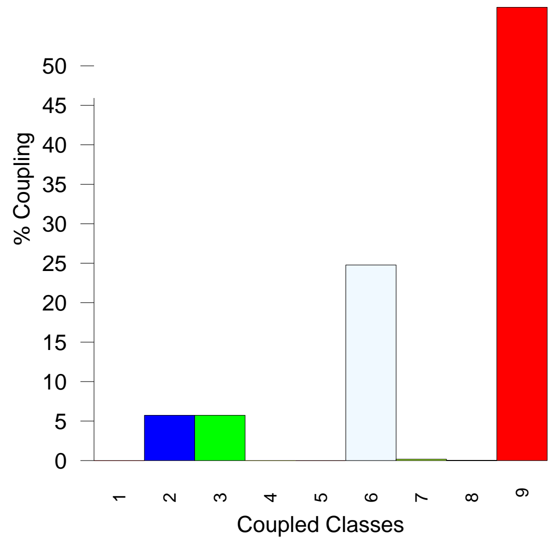


Key	Coupled Classes	%
1	JGFInstrumentor & java.io.PrintStream	8.0%
2	JGFInstrumentor & java.lang.Object	1.0%
3	JGFInstrumentor & java.lang.StringBuffer	47.8%
4	JGFInstrumentor & java.lang.System	8.0%
5	JGFInstrumentor & java.util.Hashtable	15.0%
6	JGFInstrumentor & jgfutil.JGFTimer	6.9%
7	JGFTimer & java.io.PrintStream	6.0%
8	JGFTimer & java.lang.Object	1.5%
9	JGFTimer & java.lang.String	1.5%
10	JGFTimer & java.lang.StringBuffer	51.9%
11	JGFTimer & java.lang.System	7.1%
12	moldyn.md & java.lang.Math	1.0%
13	moldyn.md & java.lang.Object	0.3%
14	moldyn.md & java.text.NumberFormat	2.8%
15	moldyn.md & moldyn.particle	8.0%
16	moldyn.md & moldyn.random	2.8%
17	moldyn.particle & java.lang.Math	0.9%
18	moldyn.particle & java.lang.Object	0.9%
19	moldyn.particle & moldyn.md	15.0%

Number of method calls that represent 100% Coupling

Class	Calls
gfutil.JGFInstrumentor	157
jgfutil.JGFTimer	174
moldyn.md	455
moldyn.particle	121
Total	970

Figure 3: Degree of Static Coupling for non-API classes involved in execution of JGFMolDynBench-SizeA. Only classes that exhibited above 10% of the total number of accesses are shown.

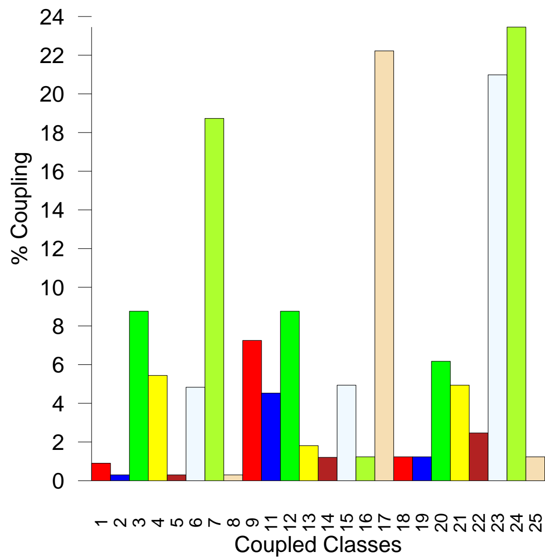


Key	Coupled Classes	%
1	moldyn.md & java.lang.ClassLoader	0.1%
2	moldyn.md & java.lang.Math	5.7%
3	moldyn.md & java.lang.Object	5.7%
4	moldyn.md & java.text.DecimalFormat	0.1%
5	moldyn.md & java.text.NumberFormat	0.1%
6	moldyn.md & moldyn.particle	24.8%
7	moldyn.md & moldyn.random	0.2%
8	moldyn.particle & java.lang.Object	0.1%
9	moldyn.particle & moldyn.md	57.4%

Number of method calls that represent 100% Coupling

Class	Calls
moldyn.md	1,744,408
moldyn.particle	13,218,243
Total	14,989,104

Figure 4: Degree of Dynamic Coupling for non-API classes during execution of JGFMolDyn. Only classes that exhibited above 10% of the total number of accesses are shown.

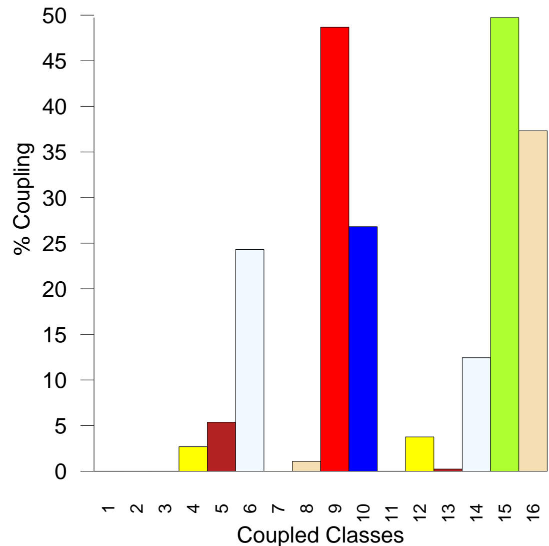


Key	Coupled Classes	%
1	Database & java.io.DataInputStream	0.9
2	Database & java.io.OutputStream	0.3
3	Database & java.io.PrintStream	8.761
4	Database & java.io.StreamTokenizer	5.4
5	Database & java.lang.Object	0.3
6	Database & java.lang.String	4.8
7	Database & java.lang.StringBuffer	18.7
8	Database & java.lang.System	0.3
9	Database & java.util.Vector	7.3
11	Database & spec.benchmarks._209_db.Entry	4.5
12	Database & spec.harness.Context	8.8
13	Database & spec.io.FileInputStream	1.8
14	Database & spec.io.FileOutputStream	1.2
15	Main & java.io.DataInputStream	4.9
16	Main & java.io.FilterInputStream	1.2
17	Main & java.io.PrintStream	22.2
18	Main & java.lang.Integer	1.2
19	Main & java.lang.Object	1.2
20	Main & java.lang.String	6.2
21	Main & java.lang.StringBuffer	4.9
22	Main & java.lang.System	2.5
23	Main & spec.benchmarks._209_db.Database	21.0
24	Main & spec.harness.Context	23.5
25	Main & spec.io.FileInputStream	1.2

Number of method calls that represent 100% Coupling

Database	331
Main	81
Total	427

Figure 5: Degree of Static Coupling for non-API classes in spec.benchmark._209_db package involved in execution of _209_db. Only classes that exhibited above 10% of the total number of accesses are shown.



Key	Coupled Classes	%
1	Database & java.io.DataInputStream	0.1
2	Database & java.io.PrintStream	0.1
3	Database & java.io.StreamTokenizer	0.1
4	Database & java.lang.ClassLoader	2.7
5	Database & java.lang.Object	5.4
6	Database & java.lang.String	24.3
7	Database & java.lang.StringBuffer	0.1
8	Database & java.lang.ref.Finalizer	1.1
9	Database & java.util.Vector	48.7
10	Database & java.util.Vector$1	26.8
11	Database & spec.benchmarks._209_db.Entry	0.1
12	Database & spec.io.FileInputStream	3.8
13	Entry & java.lang.Object	0.2
14	Entry & java.lang.String	12.4
15	Entry & java.util.Vector	49.7
16	Entry & java.util.Vector$1	37.3

Number of method calls that represent 100% Coupling

Database	18,616,692
Entry	6,410,043
Total	25,026,796

Figure 6: Degree of Dynamic Coupling for non-API classes in spec.benchmark._209_db during execution of _209_db. Only classes that exhibited above 10% of the total number of accesses are shown.

- 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 coupling class level metrics and their correlation with the external quality attributes of a design.
- Further investigating the correlation between static and dynamic coupling metrics.
- Further investigating the impact inheritance has on dynamic coupling metrics.
- Developing a set of dynamic object level coupling metrics to measure the levels of coupling in individual objects at runtime.
- Investigating how useful dynamic measures of coupling may be in testing. As the more coupled a class is the more rigorously it may need to be tested.

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] Briand, L.C., Daly, J.W. and Wust, J.K., "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering* , Vol. 25, no. 1 pp. 91–121, Jan/Feb 1999.

[3] Briand, L.C., "Empirical Investigations of Quality Factors in Object-Oriented Software," *Empirical Studies of Software Engineering* , Ottawa, Canada, March 4–5, 1999.

[4] Briand, L.C. and Wust, J., "The Impact of Design Properties on Development Cost in Object-Oriented Systems," *Proc. Seventh International Symp. Software Metrics, Metrics '01* , pp. 260–271, 2001.

[5] Chidamber, S.R. and Kemerer, C.F., "Towards a Metrics Suite for Object-Oriented Design," *Proc. Conference on Object-Oriented Programming: 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* $5^t h$ *International Symposium on Requirements Engineering* , pp. 212–219, 2001.

[8] Engel, J., "Programming for the Java Virtual Machine," Addison-Wesley, 1999.

[9] 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.

[10] Gonzalez, R.R., "A Unified Metric of Software Complexity: Measuring Productivity, Quality, and Value," *The Journal of Systems and Software* , Vol. 29, pp. 17–37, 1995.

[11] Gosling, J., Joy, B. and Steele, G. "The Java Language Specification," Addison–Wesley, Reading, MA, 1996.

[12] Java Grande Forum Benchmark Suite, Available at the following WWW site: `http://www.epcc.ed.ac.uk/research/javagrande/-benchmarking.html`.

[13] 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.

[14] Mehlitz, P.C., "Performance Analysis of Java Implementations", Available at the following WWW site: `http://www.transvirtual.com/presentations/-speed/index.html`.

[15] Shooman, M.L., "Software Engineering: Design, Reliability and Management," , McGraw Hill, New York, pp. 150–151, 1983.

[16] SpecJVM98, Available at the following WWW site: `http://www.spec.org/org/jvm98`.

[17] 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.

[18] Yourdon, E. and Constantine, L.L., "Structured Design," , Prentice Hall, Englewood Cliffs, NJ, 1979.