Computer Science
Technical Report

Colorado State University

Measuring Class Cohesion in Java

Martin F. Shumway

M.S. Thesis
June 11, 1997

Technical Report CS-97-113

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

Thesis:
Measuring Class Cohesion in Java

Martin F. Shumway

June 11, 1997

Abstract

Cohesion is an internal software attribute which tells how tightly the components of a software module are bound together in design or implementation. Highly cohesive software modules have a basic function and are difficult to decompose. Cohesion is thought to be a desirable goal in software construction, leading to better values for external attributes such as maintainability, reusability, and reliability. To test this hypothesis, a good measure of class cohesion is needed.

We examine the problem of measuring class cohesion in object-oriented systems written in Java. The research utilizes the class cohesion measure as proposed by Bieman and Kang [Bieman95]. The measure counts the proportion of method pairs in a class exhibiting connectedness through the use of one or more common instance variables of that class.

For a measure to be valid, it must obey the requirements of measurement theory. The Bieman and Kang cohesion measure is shown to meet in large part the requirements of its empirical relation system. The measure does not completely reflect the property of factorability, which is the ability to split a class without breaking method connections. A new approach to measuring factorability is proposed.

A Java-specific object model suitable for measurement is defined. A new measure of class size based on Java byte code statements is proposed. The results of an empirical study demonstrate that cohesion is not a measure of class size.

To facilitate automatic gathering of class cohesion data from publicly available Java programs, we present a tool called *Celebes*. *Celebes* performs static analysis on parsed Java codes. Acting as a compiler front-end, *Celebes* can also serve as a platform for other measurement, analysis, and instrumentation functions.

ACKNOWLEDGEMENTS

I wish to thank the support of my committee members, Drs J Bieman, S Gupta, and H Iyer. In particular, I wish to thank Dr Bieman for supplying me with this research topic, guiding it to its completion, and for remaining accessible during its investigation. Byung-Kyoo Kang's doctorate work on cohesion provided a starting point for my investigation.

I also wish to recognize the Computer Science Department for its service in educating me, and my friends and family for their support throughout the four years I have spent here.

Several institutions have provided me support in various guises. Storage Technologies Corporation (Louisville, CO) financed my research assistanceship during 1994-1996. I wish to thank Robert McNitt of StorageTek for his support. The SunTest group of Sun Microsystems Incorporated (Mountain View, CA) provided the parser generator, parse tree builder, and Java grammar necessary to build a static analyzer tool for Java programs. I wish to recognize Sriram Sankar and his team for their efforts at making these tools available to the Java research community free of charge. This generosity continues the open systems culture which has made so many useful tools available to the public over the years.

Two companies provided publicly accessible Java source codes used in this research. Sun Microsystems supplied the JDK demo suite. Object Space Incorporated (Dallas, TX) provided the JGL Class Library. Java is a registered trademark of Sun Microsystems, Incorporated.

Finally, one should mention that Colorado State University is a land grant institution charged with training the citizens of Colorado and disseminating modern technologies and practices. As such, the people of Colorado deserve recognition for the educational support they provide through their taxes.

DEDICATION

This work is dedicated to my parents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cohesion is an internal software attribute which tells how tightly the components of a software module are bound together in design or implementation. Highly cohesive software modules have a basic function and are difficult to decompose. Uncohesive modules are marked by components whose presence is completely coincidental. Such components could just as well be located in another module.

Many software engineering researchers have asserted the importance of high cohesion in software construction [Yourdon79]. Highly cohesive program components have better external attributes, such as number of faults found (reliability), number of components that can be used without modification (reusability), and cost of modification and enhancement (comprehensibility and maintainability). We need a clear understanding of cohesion before the relationship between cohesion and these external quality attributes can be demonstrated.

Cohesion has been studied in the context of procedural programs for some time. Object-oriented cohesion is a concept still under development. The need to establish a firm basis for object-oriented cohesion motivates the current research.

1.2 Overview of Research

This research applies the class cohesion measure of Bieman and Kang [Bieman95] to Java programs. Classes are measured as to their relative internal connectivity. The measure of cohesion is the proportion of visible method pairs in a class which exhibit connectivity through their communication with a common instance variable of the class, compared with the total number of visible method pairs in the class. There are two variants of class cohesion: Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC).

The Bieman-Kang cohesion measures were informally proposed in [Bieman95]. The research places the Bieman and Kang proposal on a more firm mathematical foundation by demonstrating the representation condition of the measure [Fenton94]. This requires that the numerical measure of cohesion does

not contradict any notional properties of the empirical nature of cohesion. A scale type for the measure is also derived.

The original Bieman and Kang cohesion measure was applied to C++ programs. This research extends the application to Java programs. An object model specific to Java was developed to deal with numerous issues arising in class cohesion research. One example is whether to measure cohesion exhibited through static class methods and variables.

A major contribution of this research is the introduction of a tool to automatically measure cohesion in Java classes. The tool, called *Celebes*, is a metrics platform for Java programs. It performs static analysis on source code and Java byte codes. *Celebes* can be extended to implement other forms of static analysis and instrumentation.

The Bieman-Kang cohesion measures were applied to publicly available non-trivial Java software. The relationship between class size and cohesion was investigated with the hypothesis that the two attributes are independent.

Chapter 2

Measuring Cohesion

2.1 Foundations

Cohesion is an attribute of software modules that captures the degree of association of elements within a module. The programming paradigm used determines what is an element and what is a module. A closely related concept called *coupling* indicates the degree of interconnectedness between modules. These concepts were first introduced in 1974 by Stevens, Constantine, Myers [Stevens74] as a way of evaluating the effective use of modules. Designers generally try to achieve "high" cohesion and "low" coupling for modules in their software products [Budd91]. A highly cohesive module is one which has a single basic function and is difficult to split [Bieman95].

Cohesion of elements in a module can be characterized in an rank scale of desirability, with functional and data cohesion ranking highest [Budd91] [Fenton91].

- Coincidental : Elements occur together in a module for no reason.
- Logical : Elements occurring together do similar things but otherwise have no formal connection.
- Temporal : Elements occurring together execute at the same time.
- Communication : Elements occurring together share input or output data.
- Sequential : Elements execute in a particular order.
- Functional : Elements contribute to performing a particular function.
- Data : Elements occurring together present a public interface that allows for data abstraction.

It is possible for more than one kind of cohesion to be present. Since this is a rank order, statistics such as *mean* are invalid.

2.2 Measuring Cohesion in Procedural Programs

In procedural programs (programs with procedures and data declared independently, for example C, C++, Pascal, Ada), the *module* is a procedure and *element* is a global value visible throughout the program. The approaches taken to measure cohesiveness of these kind of programs have generally tried to evaluate cohesion on a procedure by procedure basis. The notional measure is one of "functional strength" of a procedure, meaning the degree to which data and procedures contribute to performing the basic function [Fenton91].

2.2.1 Program Slice Abstraction of Bieman and Ott

Bieman and Ott [Bieman93][Bieman94] formulate a measure of functional cohesion on procedures based on a relation between output tokens (output variables) and program slices. A *program slice* is a set of program statements which include references to a particular program variable. A *glue token* is a token which is used in more than one program slice that includes a certain statement. A *super glue token* unites all the program slices at some statement. The measures capture the number of program slices having glue or super glue tokens as a proportion of total program slices.

Bieman and Ott give three measures:

Strong Functional Cohesion (SFC) of Procedure p

$$SFC(p) = \frac{|super\ glue\ tokens|}{|tokens|}$$

Weak Functional Cohesion (WFC) of Procedure p

$$WFC(p) = \frac{|glue\ tokens|}{|tokens|}$$

Adhesiveness (A) of Procedure p

$$A(p) = \frac{\Sigma_{tokens} \frac{|program\ slices\ containing\ a\ glue\ token|}{|program\ slices|}}{|tokens| * |program\ slices|}$$

A more cohesive module will be more difficult to separate by factoring the procedure into separate ones containing distinct output tokens and data slices involving them. A procedure having no cohesion would have no glue tokens (not to mention super glue tokens). A procedure having perfect cohesion would have super glue tokens at every statement.

2.2.2 Design Level Cohesion of Kang and Bieman

Cohesion can be measured at design level as well as at code level. This is advantageous if the code has yet to be implemented. Building on their earlier work, Kang and Bieman [Kang96a] [Kang96b] show how the relationship between pairs of output tokens can indicate the relative "strength" of a module.

An ordinal scale of cohesion measures is defined: Coincidental, Conditional, Iterative, Communicative, Sequential, and Functional. Each pair of output tokens in a module is evaluated for the strongest cohesion the pair exhibits. The minimum such value over all output token pairs gives the *Design Level Cohesion* (DLC) for the module. DLC is an associative measure which serves as a lower bound for our intuition of module cohesion.

Kang and Bieman developed another measure called *Design Functional Cohesion* (DFC). DFC is a slice based measure which averages adhesiveness of output token slices corresponding with the interface points of the module. Since the module's interface is known at design time, cohesion can be measured. Averaging is possible because of the proportional (ratio scale) nature of adhesiveness. In practice, DFC and DLC measures correlate closely.

Both DLC and DFC measure the intuitive "relatedness" of module components. These measures can identify modules that perform multiple functions having little or nothing to do with one another. These modules may be poorly designed and present good candidates for restructuring. In [Kang96b] Kang and Bieman show how design level cohesion can be visualized. They propose program transformations which decompose modules exhibiting low cohesion.

DLC and DFC are cohesion measures applied to procedural programs. One goal of the present research is to bring the analysis of class cohesion to a similar starting point where visualization and restructuring may proceed.

2.3 Measuring Cohesion in Object-Oriented Programs

In the object-oriented programming paradigm, modules are instances of classes. Elements are the public methods accessible through the class interface. Functional and Data cohesion are the same in this paradigm. Most cohesion measurement approaches consider interactions between methods and instance variables. The cohesion measure of a class refers to the difficulty of factoring the class into separate classes: the more cohesive a class, the more difficult it is to factor.

2.3.1 Class Slice Abstraction of Ott and Mehra

Ott and Mehra [Ott95] extend the earlier work of Bieman and Ott that uses program slices to measure connectedness of class slices. Ott is currently pursuing this line of investigation.

2.3.2 Module Interactions of Briand, Morasca, and Basili

Briand, Morasca, and Basili define cohesion in design terms [Briand94, Briand96]. They view the elements of the module as its exported features, and a module as an abstract data type (ADT). An exported feature A interacts with feature B if the change of one of A's definitions or uses may require a change in one of B's definitions or uses. A single procedure may export features A, B. Thus, cohesion is the interaction between the exported features of a procedure.

Given a procedure, three measures are defined: *Neutral Ratio of Cohesive Interactions, Pessimistic*

Ratio of Cohesive Interactions, and *Optimistic Ratio of Cohesive Interactions*. These measures capture the degree to which the software modules share features. The idea that cohesion only matters with respect to exported, or public, interfaces is a useful one. The reason a module exhibits low cohesion is that its public procedures and data have few interactions with one another.

2.3.3 The Lack of Cohesion Measure of Chidamber and Kemerer

Chidamber and Kemerer [Chidamber94] were one of the first to formulate a measure of cohesion based strictly on objects. We take many ideas from this work. Objects are instantiations of classes. A class is considered a design for the production of an object. Classes can be sub-classed (a more specific version can be derived through inheritance), factored into smaller classes, or composed into larger classes [Booch94]. This work makes an appeal to the deeper structure of classes, in which the combination of two classes yields a class whose properties are the union of those of the constituent classes. Where there is an intersection of these properties, the constituents exhibit an intersection of sets of instance variables that are used by methods. If the methods of a class perform their operations on all the same instance variables, then the class is maximally cohesive. If each method of a class performs its operations on a singleton or empty set of instance variables, then the class exhibits zero cohesiveness.

The *Lack of Cohesion Measure* (LCOM) indicates the difference between the number of method pairs which share access to an instance variable, and the number of method pairs which do not. It is an inverse measure of cohesiveness:

$$LCOM = |P| - |Q| \; if \; |P| > |Q|, \; 0 \, otherwise.$$

where P is a method pair with intersecting instance variable sets, and Q is a method pair with non-intersecting instance variable sets.

One key question is what constitutes a method. In a critique of Chidamber and Kemerer, Churcher and Shepherd [Churcher95] consider whether the number of methods should be equal to those defined within the current class, or those inherited from superclasses. The former view emphasizes the functionality of the class. The latter view admits to its state space.

Another issue raised by Churcher and Shepherd is whether to count only visible methods (those that are public and not hidden), based on the notion that the class interface, or the view it presents to the world, is the thing of interest, not the set of all methods (whether inherited or not). The difficulty with this approach is that one class may present multiple interfaces. The main point here is that one must reconcile the language-specific issues with the general applicability of the class cohesion measure, a view that this research adopts.

The Chidamber and Kemerer approach, while exhibiting key ideas, fails to meet the basic representation requirements of an empirical relation system. This is one of the first requirements of measurement theory, so the LCOM measure cannot be used as such. Hitz and Montazeri [Hitz96] point out some examples where LCOM fails to show enough sensitivity to changes in method connection configurations, or shows too much. One example shows LCOM to have different values for a constant number of splits of a graph of method connections. Another example shows the case in which no further splits are possible in a chain of method connections, and yet incremental additions to the connection graph yield marginal changes in the LCOM measure.

2.3.4 Relative Strength Measure of Hitz and Montazeri

Hitz and Montazeri propose an improvement to the LCOM measure of Chidamber and Kemerer by making it more sensitive to small changes in structure when the graph of method connections is complete [Hitz96]. The idea here is to measure the "connectivity of degree K", where K is the number of edges which must be removed in order to disconnect the graph. This would be done to distinguish "ties" of the kind which happen when $LCOM = 1$. A linear mapping is proposed from the interval $[N-1, \begin{pmatrix} N \\ 2 \end{pmatrix}]$, where N is the number of methods and K is the number of method connections, to the interval $[0, 1]$ with the following relation:

$$C = 2\frac{|E| - (N-1)}{(N-1)(N-2)}$$

Hitz and Montazeri take care to confirm the representation condition for their cohesion measure. In their view, class cohesion is based on difficulty of separating methods in a class. With more connections, cohesiveness should increase. This corresponds to counting more edges in the connectivity graph. C indicates the extent to which the graph has deviated from the miminal graph required to maintain connectivity among a group of methods.

2.3.5 Relative Connectivity Measure of Bieman and Kang

In their work on object-oriented cohesion, Bieman and Kang [Bieman95] adopt the notion of cohesion as the interaction of methods of a class through the instance variables of the class. Two measures are proposed: Loose Class Cohesion (LCC), which counts direct and indirect connections between methods, and Tight Class Cohesion (TCC), which counts only direct connections between methods.

The goal of LCC is to detect coincidental components in a class. Low LCC suggests that some of the components of the class belong in another class instead. High LCC indicates few components can be split from the class.

The goal of TCC is to gauge the "visibility" of the connections in the class. High LCC and low TCC measures for a class indicate that many of the method connections are indirect and possibly not obvious from the source code. High LCC and high TCC indicate a highly cohesive class: no components can be split and all connections are visible from the source code.

The concept of "splitability" is succinctly stated in their paper:

> "If a class is designed in an ad hoc manner and unrelated components are included in the class, the class represents more than one concept and does not model an entity. A class designed so that it is a model of more than one entity will have more than one group of connections in the class. The cohesion value of such a class is likely to be less than 0.5. For example, if five of the six methods in a class are connected and the remaining method has no connections, the TCC and LCC of the class are both 0.67. If three of the six methods in a class are connected, and the other three are also connected with no connection between

those two groups, both the TCC and LCC of the class are 0.40. Therefore, the class cohesion measures can be used to locate the classes that may have been designed inappropriately." [Bieman95]

Chapter 3 is devoted to a formal validation of the Bieman-Kang cohesion measures.

2.3.6 Taxonomy of Briand

Briand *et al* review the cohesion measures currently under proposal with the intent of unifying them under a common framework [Briand97a][Briand97b]. A unified empirical relation system is proposed with the following properties, informally stated:

- The *Nonnegativity and Normalization* property says that cohesion values must lie in a bounded interval.
- *Null Value and Maximum Value* says that cohesion is zero if there are no relationships in a class, and maximum if no relationships can be added to the class.
- *Monotonicity* says that the addition of a relationship to a class should not decrease cohesion.
- *Composition* says that merging unconnected classes must not result in a cohesion value for the merged class which is greater than those of its constituent classes.

The Bieman-Kang cohesion measures are shown to fulfill this empirical relation system.

The authors make some interesting observations about cohesion measures in general [Briand97b]:

- Indirect connections appear to be better indicators than direct connections for showing whether a class should be split.
- A class having maximum cohesion as measured by direct connections would be required to possess direct connections between every element to every other element. This situation is rare.
- Many classes are maximally cohesive when measured by indirect connections.

Chapter 3

Formal Validation of the Bieman-Kang Cohesion Measures

3.1 Theoretical Requirements of a Measure

Historically, work in software measurement has suffered from a lack of basis in scientific measurement technique, or a misunderstanding of its application. The contributions of [Baker90] [Fenton91] [Fenton94] [Melton90] [Kitchenham95] [Zuse91] have placed software measurement on a proper mathematical foundation. The work of [Bieman94] [Hitz95] [Briand96] are examples of modern software measures firmly grounded in measurement theory. We wish to apply the same rigor to the Bieman-Kang cohesion measure.

The design of a measure should start with an understanding the entity being studied [Fenton94]. The entity in this research is a Java class. Next, the notional, or intuitive, meaning of the attribute which is to be measured must be understood. The attribute studied here is class cohesion. Notions about cohesion include "basic function," "difficult to split," "connectedness." Cohesion is an internal attribute derivable from the entity itself.

An *empirical relation system* identifies the entity (a Java class) and a relation which obtains on the entity (*is less cohesive than*). A *numerical relation system* is defined over a set of numbers (any kind) and the relation *is less cohesive than*. With respect to the Bieman-Kang measure, this is the proportion of methods connected to total methods (TCC and LCC). TCC and LCC are measures because they define a mapping from Java classes, and the relation *is more cohesive than*, to TCC and LCC values of Java classes. These values are defined over the rational interval $[0.0, 1.0]$.

The *representation condition* imposes an additional requirement on a measure: all empirical relations must be preserved in the mapping. In other words, there exists no contradiction between an entity's image in the numerical system and any notion about cohesion of the entity in the empirical system. A mapping which meets all these requirements is a *representation*.

As an example, consider two classes A and B, where A has more partitions than B. Our empirical understanding of cohesion is such that A is more *factorable* than B. Therefore, A is less cohesive than

B. If a case exists in which two classes have the same cohesion measure but different factorability, that case would contradict the representation condition. If on the other hand it were possible to prove no such case exists, then that proof would help confirm the representation condition.

In addition to demonstrating the representation condition, a measure must have a scale type. This may be any one of nominal, ordinal, interval, ratio, and absolute. A measure of nominal scale type does nothing more than distinguish among the entities being measured. A measure of ordinal scale type imposes a total order on the entities being measured. We should specify the *admissible transformation* on the measure which distinguishes its scale type. In the case of nominal scale type, the admissible transformation is a bijection to a different set of distinguishing values. For ordinal scale type, it is a monotonic increasing function remapping the values to a new total order. For a ratio scale type, it is multiplication by a scalar.

3.2 Definitions

In their work on object-oriented cohesion, Bieman and Kang [Bieman95] adopt the notion of cohesion as the interaction of methods of a class through the instance variables of the class. This section states the measure's definition in terms where the representation condition for measures can be assessed. Then the empirical relation systems and numerical relation systems are defined, and the representation condition is assessed for the measures.

Definition 1 *(Directly Uses Relation) An instance variable I is directly used by a visible method M in a class C if I appears as a data token in M.*

A method is visible if it can be accessed from outside the class. An instance variable does not have to be visible. In this model, visible methods do not include the class constructors or destructors, (`finalize` in Java). This is because in general the constructor directly uses every instance variable. These relations would obscure any meaningful cohesion measurement. The same applies for the self pointer (`this` in Java) since it is implicitly used in every expression.

Definition 2 *(Indirectly Uses Relation) An instance variable I is indirectly used by a method M in a class C if I is directly used by some other M' in C such that M' is directly called by M, or is indirectly called through a chain of intermediate M_i also of C.*

Definition 3 *(Uses Relation) A method* uses *an instance variable if it directly or indirectly uses it and the method and instance variable are in the same class.*

Definition 4 *(Method Abstraction) A method abstraction is a set of instance variables used by a method.*

Definition 5 *(Directly Connects Relation) A relation* directly connects *is defined on a pair of distinct methods of a class if there exists a non-empty intersection of the method abstractions.*

Definition 6 *(Indirectly Connects Relation) A relation* indirectly connects *is defined on a pair of distinct methods of a class if there exists a chain of direct connections between the methods through one or more intermediate methods also in the class.*

The relation *directly connects* is anti-reflexive and symmetric. The relation *indirectly connects* is symmetric and transitive. The relation *indirectly connects* is also the transitive closure of the *directly connects* relation less the *directly connects* relation.

Definition 7 *(Tight Class Cohesion) Tight Class Cohesion is the proportion of direct connections of visible methods in a class.*

$$TCC(C) = \frac{|\,(M_i, M_j)_{DirectlyConnects}\,|}{\begin{pmatrix} |M| \\ 2 \end{pmatrix}}$$

Definition 8 *(Loose Class Cohesion) Loose Class Cohesion is the proportion of direct or indirect connections of visible methods in a class.*

$$LCC(C) = \frac{|\,(M_i, M_j)_{DirectlyConnects}\,| + |\,(M_i, M_j)_{IndirectlyConnects}\,|}{\begin{pmatrix} |M| \\ 2 \end{pmatrix}}$$

An invariant is that $TCC(C) \leq LCC(C)$.

3.3 Empirical Relation System

The empirical relation system captures the intuition of a measure. In this section the notions of the cohesion measure are formally stated.

Section 2.3.4 presented the proposed measure of cohesion by Hitz and Montazeri. The goal for that measure is the assessment of the cost of detaching a method from its class. This is not the goal of the Bieman-Kang cohesion measures.

The goal of the Bieman-Kang loose cohesion measure is to identify components which are "coincidental." Such classes are "factorable" into separate classes without affecting method call and field use relationships. For two MIGs a cohesion measure should yield lower cohesion for the MIG that has more factors.

The goal of the Bieman-Kang tight cohesion measure is to find the degree to which the class is "filled out" with explicit method connections. It is not meant to identify separable factors.

In general, a class cohesion measure based on method connectivity should possess the following properties. See Definition 13 for a definition of *partition*.

Property 1 *(Maximum) A class to which no method connections can be added should have maximum cohesion.*

Property 2 *(Minimum) A class in which no method connections exist should have no cohesion.*

Property 3 *(Well-definedness) Every class should have a measure.*

Property 4 *(Ordering Relation) A relation* "is less cohesive than" *should be defined for any two classes. This relation should impose a total order on the universe of classes.*

Property 5 *(Homomorphic Invariance) Two classes which are the same except for naming should have the same cohesion.*

The next properties should be true of a measure capturing the notion of tight cohesion.

Property 6 *(Monotonicity) A class from which a connection has been removed should have less cohesion than prior to the removal.*

Property 7 *(Tight Invariance) Tight cohesion should be invariant to rearrangement of connections.*

Property 8 *(Partition Invariance) A class with a given partition should have cohesion bounded in some interval.*

The following property relates tight cohesion to loose cohesion.

Property 9 *(Boundedness) The loose cohesion of a given partition bounds its tight cohesion.*

The last two properties should be true of a measure capturing the notion of loose cohesion.

Property 10 *(Loose Invariance) For a class of a given partition, loose cohesion should be invariant to the addition of connections.*

Property 11 *(Factorability) A class which has more factors should have less cohesion than one which has fewer factors.*

3.4 Numerical Relation System

Here we define a useful tool called the the Method Interconnection Graph (MIG). Nodes represent methods and edges represent *directly connects* relations between them. We wish to recast the numerical relation system defined in Section 3.2 in graph terms for easier analysis.

Definition 9 *A Method Interconnection Graph* $MIG(V, E)$ *is defined on a class such that* $V = \{M \epsilon C\}$ *and* $E = \{(M_i, M_j)_{DirectlyConnects}\}$.

Definition 10 *Tight Class Cohesion (TCC) is defined as the proportion of edges possible in a Method Interconnection Graph of N nodes:*

$$TCC = \frac{|E|}{\begin{pmatrix} N \\ 2 \end{pmatrix}}$$

Definition 11 *Loose Class Cohesion (LCC) is defined as the transitive closure of the Method Interconnection Graph of N nodes.*

$$LCC = \frac{|E|_{TransitiveClosure}}{\begin{pmatrix} N \\ 2 \end{pmatrix}}$$

The MIG is also a handy visual abstraction. Figure 3.1 displays some of the method connection configurations possible for a 5-method class. The configurations range from maximally cohesive (MIG A is complete), to minimally cohesive (MIG P has no edges). The notional "factorability" of a class is the extent to which the class can be split (partitioned) into disconnected subclasses each with one or more methods. Each factor maps to a subgraph of the MIG.

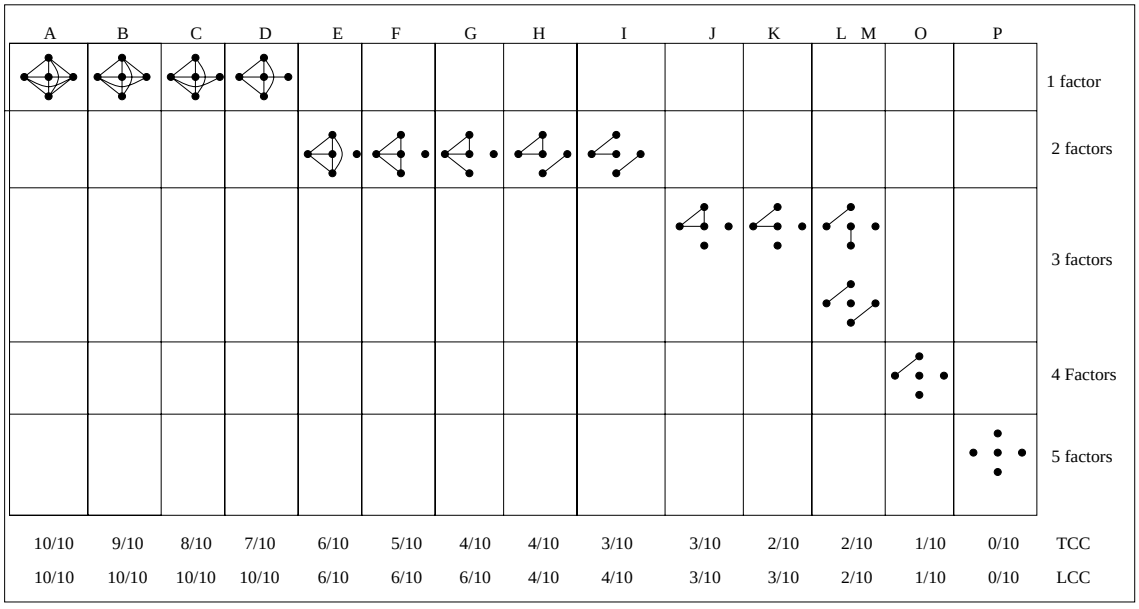


Figure 3.1: Some Method Interconnection Graphs (MIG) for a 5-node Class

Definition 12 *A subgraph is a graph defined over a subset of nodes in the MIG, and which has a unique configuration of edges. A subgraph of an MIG with N nodes can have from* 1 *to N nodes.*

Definition 13 *A partition on an MIG is a mapping of its nodes to its subsets, or factors. Each factor can have a family of subgraphs defined on it.*

3.5 The Representation Condition for Tight Cohesion

In this section we confirm the representation condition for tight cohesion (TCC). Each property of the empirical relation system for TCC is verified. Most are trivial to prove but the procedure is done anyway for completeness. Several anomalies are noted but none contradict the tenets of the measure's empirical relation system.

3.5.1 Demonstration of the Maximum and Minimum (Properties 1, 2)

The following result establishes the existence of a maximum and minimum cohesion value that is independent of the number of nodes in the MIG.

Proposition 1 *For an Method Interconnection Graph of N nodes, the maximal TCC is that defined for the complete subgraph of 1 factor, or $TCC_{maximal} = 1$. The minimal TCC is that defined for the singleton subgraphs of N factors, or $TCC_{minimal} = 0$. LCC = TCC in both cases.*

Proof With one factor in the partition, all nodes are in the single fully connected graph that makes up the factor. In this case $TCC = \frac{\binom{N}{2}}{\binom{N}{2}} = 1$. With N factors, there are no edges in any of the subgraphs, so TCC = 0. LCC is computed from the transitive closure of the directly connects relation, but in these two cases the transitive closure of the relation is itself. $\diamond$

3.5.2 Demonstration of Well-Definedness (Property 3)

Proposition 2 *LCC and TCC are well-defined measures over the rational interval [0.0, 1.0].*

Proof We need to show that LCC and TCC are total functions from the domain of MIGs with N nodes to a co-domain that is a proper subset of the rational numbers. By Proposition 1 the maximum value of LCC and TCC is 1.0 and the minimum value is 0 for all $N \geq 2$. Any other value falls in between since it is a proportion. The special case of $N = 1$ is handled by specifying LCC = TCC = 0. $\diamond$

3.5.3 Demonstration of the Ordering Relation (Property 4)

The next result shows that an ordering relation *is less cohesive than* exists between any two MIGs, and that this ordering imposes a total order on a set of MIGs.

Proposition 3 *The domain of method interconnection graphs can be ordered by a relation* is less cohesive than *which is defined for both LCC and TCC.*

Proof The relation *is less cohesive than* is a mapping $(LCC_A, LCC_B) \longrightarrow \{true, false\}$, or $(TCC_A, TCC_B) \longrightarrow \{true, false\}$. LCC and TCC are well-defined for every MIG. Any two values can be compared as rational numbers. The result is true or not true. $\diamond$

There will always be cases where distinct MIGs have the same cohesion measure. Figure 3.2 shows a case where $N = 6$. The following result states that in general it is not possible to establish a cohesion measure based on counting graph edges which can distinguish among all possible MIGs for a given N. Consequently, a strict total ordering of the domain of MIGs according to LCC or TCC is impossible. The requirement that there be a strict total order was never stated in the empirical relation system, so this is not a contradiction.

Proposition 4 *For $N > 3$ nodes, a strict total ordering of method interconnection graphs by an edge-counting cohesion measure is not possible.*

Proof By a simple construction, a ring of N nodes can be established (hence the requirement that $N > 3$). Now reconnect one edge between nodes n_1 and n_N such that the new edge connects nodes n_1 and n_{N-1}. This new subgraph has the same number of edges, so TCC is invariant. The number of subgraphs has not changed, so LCC is invariant. Node n_1 still has degree 2, but node n_N now has degree 1 and n_{N-1} has degree 3. The sets of node degrees of each subgraph are no longer the same. By graph theory there cannot be an isomorphism. Since there exist MIG pairs which cannot be distinguished by either LCC or TCC, a strict total ordering of MIGs is not possible. $\diamond$

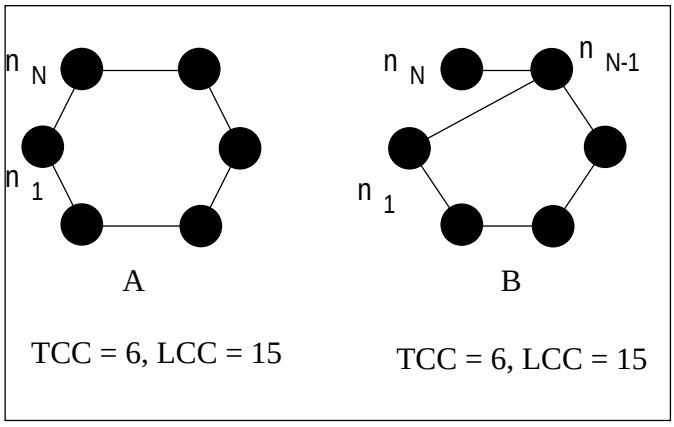


Figure 3.2: Two MIGs with Equal Cohesion, Equal Factors, but Non-Isomorphic Subgraphs

The previous example shows an instance where both the measure and the notion of the measure could be improved. If the measure would take into account the minimum in-degree of a node in A is 2, and

the minimum in-degree in B is 1, then a node in B can be detached (*ie* a method factored) by removing 1 edge instead of 2. Therefore B would be *less cohesive than* A.

Another anomaly concerns the case in which LCC and TCC measures for two MIGs give a contradictory ordering of MIGs. Consider Figure 3.3. Two 7-node MIGs are partitioned differently: A has three factors and B has two. A has LCC of $6/21$ and TCC of $6/21$. B has LCC of $9/21$ and TCC of $5/21$. Compared by LCC, A is "less cohesive" than B. Compared by TCC, B is "less cohesive" than A. Our notion of cohesion suggests B is "more cohesive" than A because B has fewer factors. But whether this fact can be derived depends on which measure is used. Once again, there is no requirement in the empirical relation system that the measures "tell the same story," for if there were, no reason would exist for defining more than one cohesion measure.

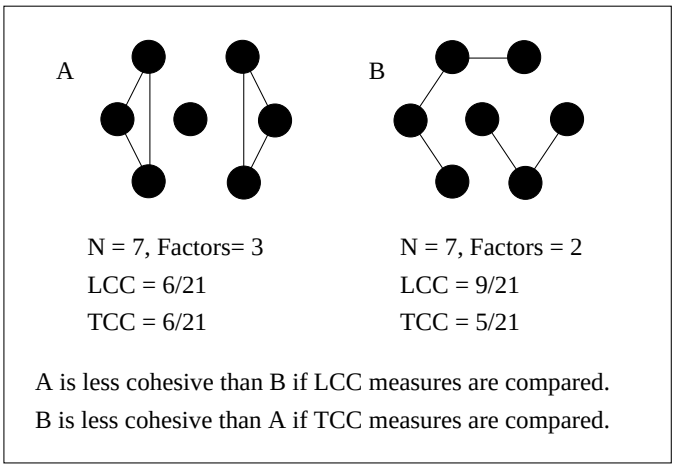


Figure 3.3: Two MIGs with Contradictory Cohesion Measures

3.5.4 Demonstration of Homomorphic Invariance (Property 5)

Each subgraph of an MIG can be replaced by its homomorphisms without affecting the MIG's cohesion. For instance, in Figure 3.1 MIGs L and M are the same except for relabeling. They have the same cohesion measures.

Proposition 5 *The cohesion measures TCC and LCC are invariant under graph homomorphism.*

Proof Each MIG has an adjacency matrix defined for it. From graph theory we know that the adjacency matrix is homomorphic under relabeling. TCC is measured directly by the adjacency matrix. Therefore, TCC is invariant under relabeling. LCC also includes nodes that are indirectly connected. The transitive closure of the *indirectly connects* relation is a fully connected graph with an adjacency matrix defined for it which is also isomorphic under relabeling. $\diamond$

3.5.5 Demonstration of Boundedness (Property 9)

Proposition 6 *For a given partition on a Method Interconnection Graph, LCC is the least upper bound of TCC.*

Proof LCC is calculated from the transitive closure of each factor. TCC is calculated from the actual number of edges. When the set of edges of each factor each match their transitive closure, the measures are the same. In this case LCC = $TCC_{maximum}$ of the bounded interval proved in Proposition 9. So LCC is an upper bound. LCC is a member of a set of upper bounds B which includes 1. The elements of B are positive rational numbers which are well-ordered, and consequently have a least member. This is LCC. So LCC is a least upper bound. $\diamond$

The boundedness property also suggests the notion of "filling out" a partition over a MIG with edges until its "potential" cohesiveness is realized with explicit connections between methods [1] . This notion is not at variance with the empirical relation system since it makes no attempt to combine the two measures.

3.5.6 Demonstration of Monotonicity (Property 6)

A class which has a method connection added or removed should increase or decrease cohesion. Note that only TCC necessarily changes from the removal of an edge in the MIG.

Proposition 7 *TCC is monotonically decreasing with the deletion of a connection in the Method Interconnection Graph.*

Proof From the definition of TCC, a removal of a connection decreases the edge count of the graph. The resulting proportion is less. If no edges result then TCC = 0. $\diamond$

3.5.7 Demonstration of Tight Invariance (Property 7)

The following property says that tight cohesion measure is not affected by the change in connections in the MIG. For a class this means that method direct connections can be changed without affecting tight cohesion.

Proposition 8 *For a Method Interconnection Graph TCC is invariant to the rearrangement of edges.*

Proof This follows from the definition of TCC since the edge count is what determines the measure. $\diamond$

3.5.8 Demonstration of Partition Invariance (Property 8)

The next result shows that for a given partition, tight cohesion is bounded in some interval.

[1] One could even think of a cohesion tuple of (LCC, TCC) where MIGs are ordered based first on comparing LCC values, and if equal, then on TCC values. However, this approach does not work.

Proposition 9 *Given a partition of P factors defined over of a Method Interconnection Graph of N nodes, each factor having n_i nodes, and the subgraphs ranging from the minimally to maximally connected have TCC values in the interval*

$$\left[TCC_{minimum} = \frac{\Sigma_{i=1}^{P}(n_i - 1)}{\binom{N}{2}}, \frac{\Sigma_{i=1}^{P}(n_i - 1) + 1}{\binom{N}{2}}, \ldots \frac{\Sigma_{i=1}^{P}\binom{n_i}{2}}{\binom{N}{2}} = TCC_{maximum} \right]$$

Moreover, TCC is a strict total order within this partition: $\{TCC_0 < TCC_1 < \ldots < TCC_{maximal}\}$.

Proof The minimal TCC is derived from the minimal number of edges required to connect a graph of n_i nodes, or $n_i - 1$. These figures are summed for all factors in the MIG. The maximal case is the number of node pairs possible for each subgraph. These are summed for all subgraphs. The admissible values for TCC range from that for a subgraph which is minimally connected, to that for subgraph which is maximally connected. Each incremental value is obtained by adding an edge to the subgraph, increasing the TCC numerator by one. Eventually, no more edges can be added because each subgraph is fully connected: this is the maximal TCC case. The strict total ordering of the TCC interval follows directly from the arithmetically increasing sequence of terms. ◇

3.6 Representation Condition for Loose Cohesion

The Bieman-Kang loose cohesion measure (LCC) does not meet its representation condition due to the contradiction of factorability.

3.6.1 Demonstration of Loose Invariance (Property 10)

The loose cohesion of a class with a certain partition (a certain number of factors each with a certain number of methods) should not change from the deletion of edges which do not affect the partition. This is because loose cohesion is measuring the *potential* of cohesion of a class.

Proposition 10 *For a Method Interconnection Graph of a given partition, LCC is invariant to the deletion of an edge if the resulting partition is the same.*

Proof For the partition to be the same after edge deletion, no cuts in the graph can occur. An edge deletion under this condition implies some other edge still connects the nodes of a factor. LCC is calculated from the transitive closure of the edges of the MIG. The closure set contains the same edge that was deleted. Thus LCC is unaffected. ◇

3.6.2 Contradiction of Factorability (Property 11)

The Bieman-Kang cohesion measures do not completely distinguish factorability of classes. Counterexamples are numerous. Figure 3.4 shows an example of two MIGs with the same cohesion but different

number of factors. In this case, the measure fails to discriminate between MIGs having different cohesion.

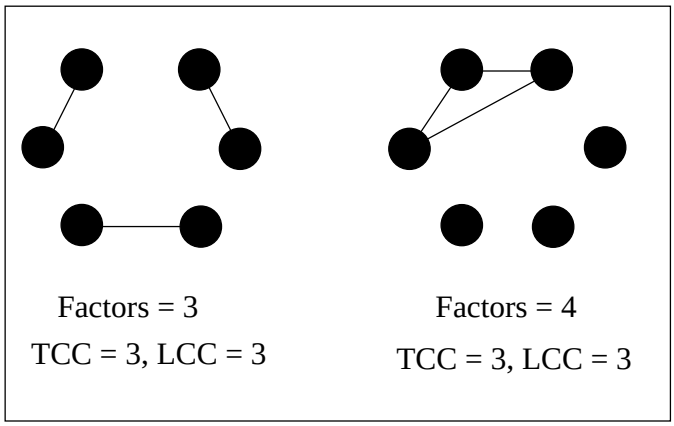


Figure 3.4: MIGs of Equal Cohesion but Different Number of Factors

3.6.3 Analyzing Factorability

To better study the population of graph partitions, we appeal to to a classic problem in combinatorics. The Integer Partition Problem tells us in how many ways an integer n can be represented by a multiset of positive numbers that add up to n. For example, $P(4) = |\{4\}, \{3+1\}, \{2+2\}, \{2+1+1\}, \{1+1+1+1\}| = 5$. The following theorem gives a recurrence for generating this sum [Bogart83] [Hall86]. The proof may be found in the references.

Theorem 1 *The number of ways to put n indistinguishable objects into k indistinguishable boxes with no box empty is*

$$P(n,k) = \sum_{i=1}^{k} P(n-k,i)$$

where $P(n,1) = P(k,k) = 1$, *and* $P(m,k) = 0$ *for* $m < k$.

A Ferrer diagram helps visualize the decomposition process. Figure 3.5 shows the derivation of $P(10,4)$, a MIG of 10 nodes and 4 factors. Each Ferrer diagram shows the configuration of multiset counts. The counts are derived by totaling each column in the diagram. Some counts may be duplicate, so more than one factor may have the same number of nodes.

Theorem 1 only counts the number of partitions. Algorithm 1 generates an enumeration of the partitions. The set of partitions for each $K = 1..N$ can be obtained by varying K.

Algorithm 1 *Generate the distinct partitions of a graph with N nodes and K factors.*

```
inputs: N = number of nodes, K = number of factors
internal state: empty Stack[1..K][1..K] = 0
                sp = 1
output: set of graph partitions of N nodes and K factors

partition(n, k)
  if n = k  OR  k = 1 then
    push(bins(n, k))
    emit()  // copy to the output set
  else
    for i = 1..k do
      push bins(k, k)
      partition(n-k, i)
      pop
    end
  end
end

push(bins)  // build the Ferrer diagram
  Stack[sp][1..length] = bins[1..length]
  Stack[sp][length+1..K] = 0
  ++sp
end

pop()
  Stack[sp][1..K] = 0
  --sp
end

emit()  // form the partition from the Ferrer diagram
  for c = columns in Stack
    count[c] = 0;
    for r = rows in Stack
      if (Stack[r][c])
        ++count[c]
      end
    end
  end
  add count[] to output set
end
```

Lemma 1 *Proof of Correctness of Algorithm 1*

Proof (i) By Theorem 1, a finite number of partitions is visited in the recurrence. Consequently the algorithm terminates.

(ii) It is enough to demonstrate a bijection between the leaves of the recursion tree of Algorithm 1 (see Figure 3.5 for an example recursion tree) and each term of the recurrence of Theorem 1. The relation is onto because the algorithm subsumes the recurrence, and exactly one partition is generated for each terminal condition in the recurrence. It is one-to-one by the following argument. The partition is generated by totaling the columns of the Ferrer diagram at each leaf of the algorithm's recursion. The recurrence counts each distinct partition. But if the partition is not distinct, then the recurrence has double counted and is incorrect. But it is correct by the proof of Theorem 1. By contradiction, each leaf of the recursion tree produces a distinct partition. To show the other direction, we simply observe that the terminals-only (leaf-only) traversal of the recursion tree gives the exact sequence of terms produced by the recurrence. $\diamond$

Does the ordering of MIGs implied by Algorithm 1 make sense? Algorithm 1 operates on a vector of $K \geq 1$ bins each of which contains a count of a partition of $N \geq 1$ nodes. Initially, the vector is most "orderly", with $bin_{2..K} = 1$ and $bin_1 = N - K + 1$. Terminally, the bin vector equals the "entropy" vector for this N and K: $bin_i = (NdivK) + \lceil \frac{(N mod K)}{K} \rceil, i = \{1..K\}$. The algorithm progresses by introducing more bins and "spreading" the energy over a wider area. This corresponds with the notion of factorability: MIGs with more factors should have less cohesion. In addition, with the number of factors fixed, MIGs whose factors contain a higher maximum node count are farther away from "entropy," and therefore should be more cohesive. These two properties indicate the partition ordering renders the desired sequence. To see the intuition more clearly, consider Table 3.1, which gives the sequence of integer partitions for a MIG of 10 nodes.

3.6.4 Population Study of Cohesion in a MIG with 29 Nodes

Algorithm 1 produces all 4566 distinct partitions for $N = 29$ (the results for other N are similar). Figure 3.6 shows the distribution of LCC values with respect to the number of factors over which they were measured. Surprisingly, each factor count has a high degree of overlap with its neighbors. Thus LCC cannot be used as a predictor of factorability. However, the measure appears to retain a rank order by range ordering LCC values for each factor count. This fact is even more striking in Figure 3.7, which shows the beautifully self-similar distribution of cohesion values with respect to partition order (the first partition generated is the most cohesive). Self-similarity is perhaps not so surprising given the recursive way the distinct partitions are generated.

Another positive indication lies in the apparent distribution of factor counts in the partition population. Figure 3.8 shows this to be some kind of long-tailed distribution. We would expect the population of LCC measures to behave similarly, and it does, as shown in Figure 3.9.

In general, the cohesion measure cannot distinguish factorability of a class. But there exists a definite, if non-linear, relationship between the two which further research must elucidate.

N	K	Partition with Nodes per Factor	LCC
10	1	[10]	1.0
10	2	[9,1]	0.8
10	2	[8,2]	0.64444447
10	2	[7,3]	0.53333336
10	2	[6,4]	0.46666667
10	2	[5,5]	0.44444445
10	3	[8,1,1]	0.62222224
10	3	[7,2,1]	0.4888889
10	3	[6,3,1]	0.4
10	3	[6,2,2]	0.37777779
10	3	[5,4,1]	0.35555556
10	3	[5,3,2]	0.31111112
10	3	[4,4,2]	0.2888889
10	3	[4,3,3]	0.26666668
10	4	[7,1,1,1]	0.46666667
10	4	[6,2,1,1]	0.35555556
10	4	[5,3,1,1]	0.2888889
10	4	[5,2,2,1]	0.26666668
10	4	[4,4,1,1]	0.26666668
10	4	[4,3,2,1]	0.22222222
10	4	[4,2,2,2]	0.2
10	4	[3,3,3,1]	0.2
10	4	[3,3,2,2]	0.17777778
10	5	[6,1,1,1,1]	0.33333334
10	5	[5,2,1,1,1]	0.24444444
10	5	[4,3,1,1,1]	0.2
10	5	[4,2,2,1,1]	0.17777778
10	5	[3,3,2,1,1]	0.15555556
10	5	[3,2,2,2,1]	0.13333334
10	5	[2,2,2,2,2]	0.11111111
10	6	[5,1,1,1,1,1]	0.22222222
10	6	[4,2,1,1,1,1]	0.15555556
10	6	[3,3,1,1,1,1]	0.13333334
10	6	[3,2,2,1,1,1]	0.11111111
10	6	[2,2,2,2,1,1]	0.08888889
10	7	[4,1,1,1,1,1,1]	0.13333334
10	7	[3,2,1,1,1,1,1]	0.08888889
10	7	[2,2,2,1,1,1,1]	0.06666667
10	8	[3,1,1,1,1,1,1,1]	0.06666667
10	8	[2,2,1,1,1,1,1,1]	0.044444446
10	9	[2,1,1,1,1,1,1,1,1]	0.022222223
10	10	[1,1,1,1,1,1,1,1,1,1]	0.0

Table 3.1: Partitions Produced by Algorithm 1 with $N = 10$ and $K = 1..10$

3.6.5 Improving LCC

One approach to improving LCC so that it can reflect factorability of a class would be to linearlize the rank of a MIG in its integer partition list. For example, in Figure 3.1 the MIG having factors of 4,3,1,1,1 ranks 17th of 42 partitions. Its LCC would be $17/42 = 0.4048$. Such an ordering would preserve factorability and would also preserve the notion of entropy described in Section 3.6.3. Such a measure would have ordinal scale properties because it is a percentile.

TCC could be integrated with the new LCC measure by offsetting each LCC value by enough points to account for the possible edges of the partition. For example, the MIG of partition 4,3,1,1,1 would have from 5 to 9 edges. TCC would be added to LCC to yield a single scalar value. For the example, this would be $17/42 + ((E - 5)/5)/42)$, which gives possible cohesion values for this partition of 0.4048..0.4238. The value of the least cohesive member of the next partition is 0.4286. In this way a much desired strict total order which distinguishes all partitions can be imposed on the universe of MIGs for a given N.

The question arises whether MIGs of different orders (N) can be compared for cohesion under this regime. If the measure is viewed as a "potential for completeness," such a comparison might make sense. If not, a partial order over a complete lattice would suffice.

Computation of this measure would be expensive. There is no known analytic solution to the recurrence of Theorem 1. The terms for each MIG would have to be calculated using the recurrence. The number of possible partitions over a 100-node MIG exceeds 190 million [Hall86]. Fortunately, few classes have that many methods.

3.7 Scale Type of the Measure

Here we wish to demonstrate that the cohesion measures are ratio scale measures. To do this we need to show that a statement like *A twice as cohesive as B* remains meaningful under rescaling.

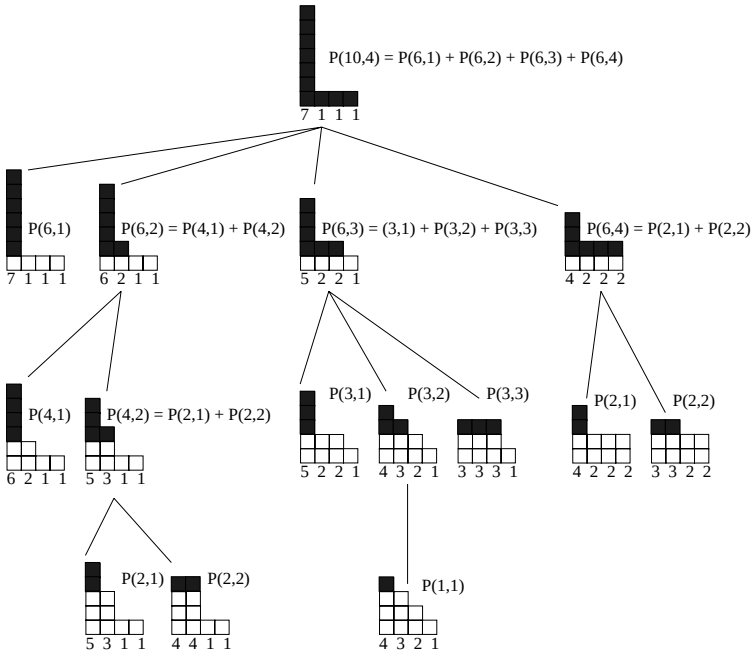
Proposition 11 *The Bieman-Kang cohesion measures are ratio scale.*

Proof The admissible transformation for ratio scale types is rescaling: for scale M and $\alpha > 0$,

$$M' = \alpha M$$

Any relationships between cohesion measures A and B must hold true under the rescaled measure. The cohesion measure is a value in the rational interval $M = [0.0..1.0]$. Under rescaling, this interval becomes $M' = [0.0..\alpha]$. Let a meaningful [2] statement under M exist such that $A = \beta B$, *ie* B is β times more cohesive than A. Under M', the statement is still meaningful: $\alpha A = \alpha\beta B$, or $A = \beta B$. $\diamond$

[2] Meaningfulness does not imply truth: 0.9 is not twice as cohesive as 0.8, but the statement is nonetheless meaningful.



(a) Derivation of integer partitions using Ferrer Diagrams. P(N,K) is the partition term to determine the number of ways N can be divided into K parts. Note that P(N,1) = 1 and P(K,K) = 1 in which case the recursion stops. The partition is derived by adding the columns of the Ferrer Diagram.
Shaded boxes denote those parts under manipulation in the recursion step.

List of Partitions

7 1 1 1
6 2 1 1
5 3 1 1
4 4 1 1
5 2 2 1
4 3 2 1
3 3 3 1
4 2 2 2
3 3 2 2

(b) List of distinct partitions obtained by performing a terminals-only (leaf-only) traversal of the recursion tree.

Figure 3.5: Derivation of the Partitions of $P(10, 4)$ Using Ferrer Diagrams

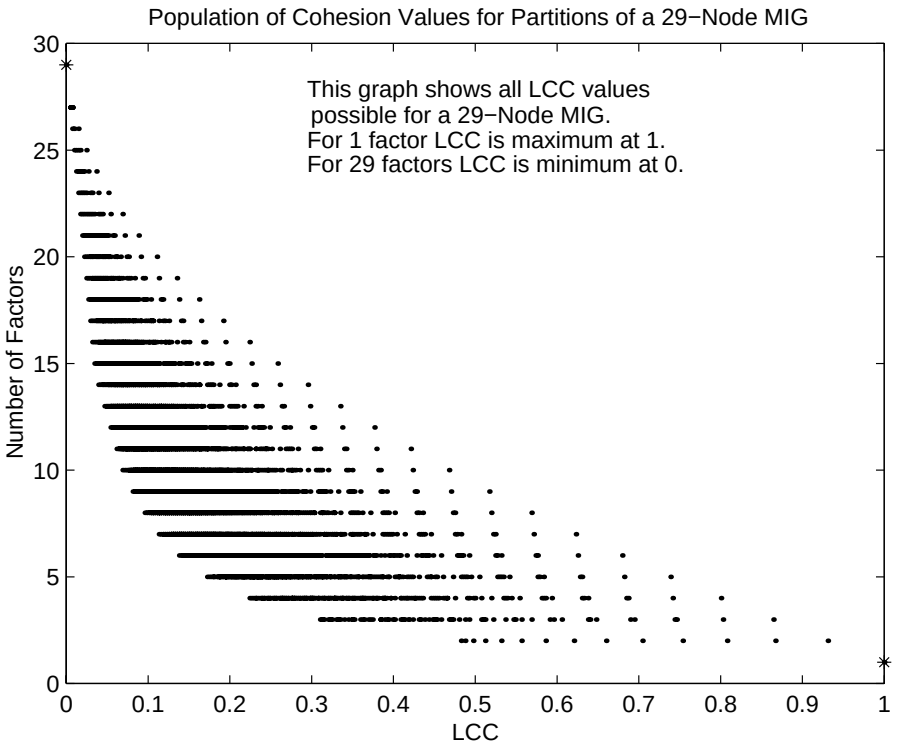


Figure 3.6: The Relationship between Number of Factors and Cohesion

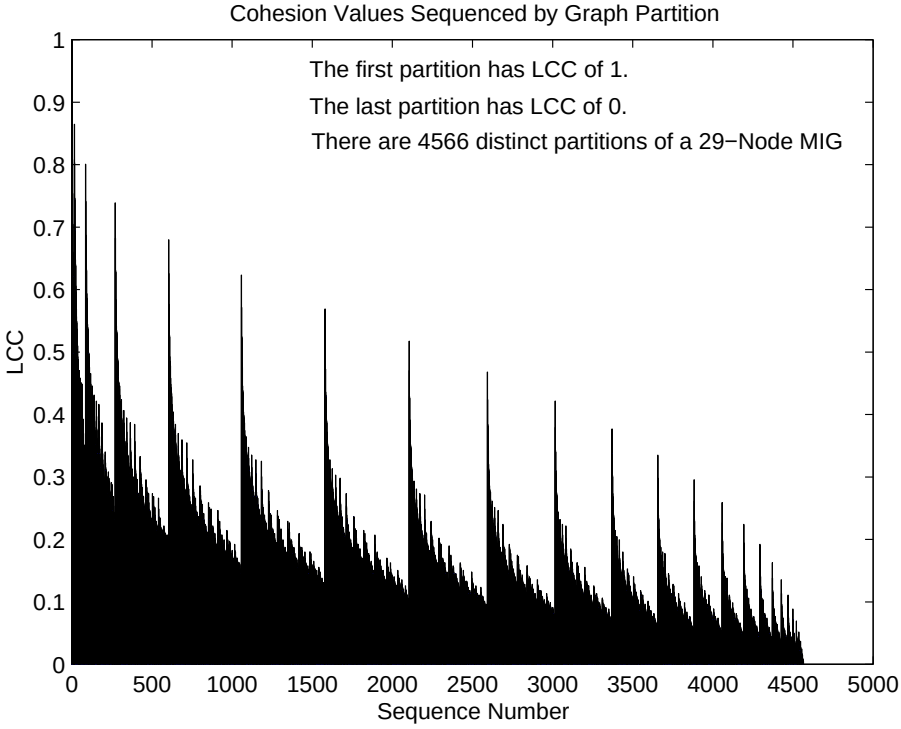


Figure 3.7: Sequence of Cohesion Values by Graph Partition

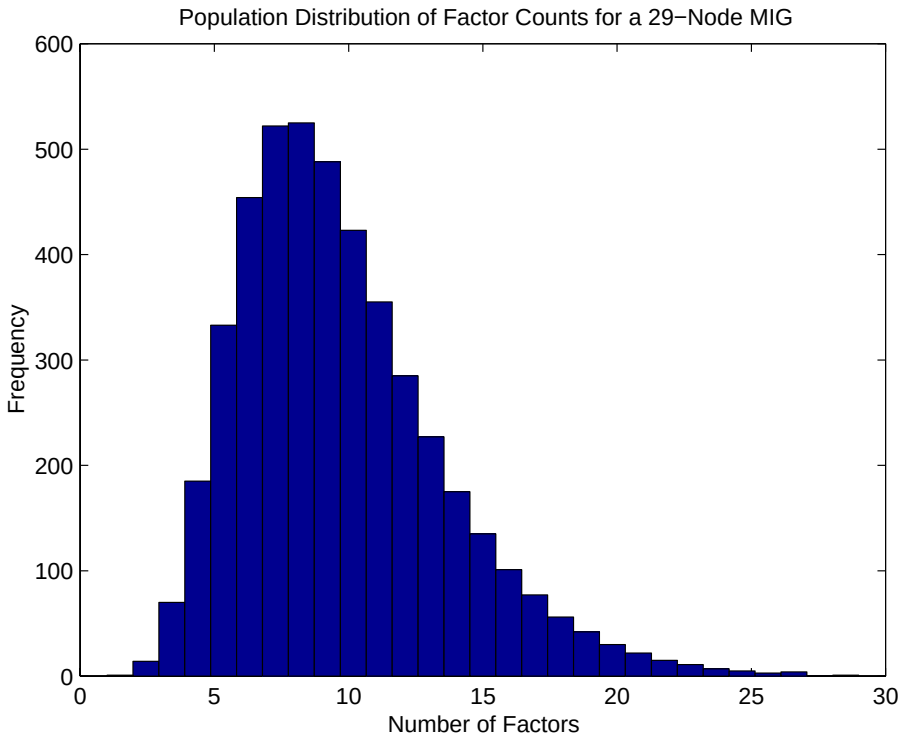


Figure 3.8: Population of Factor Counts

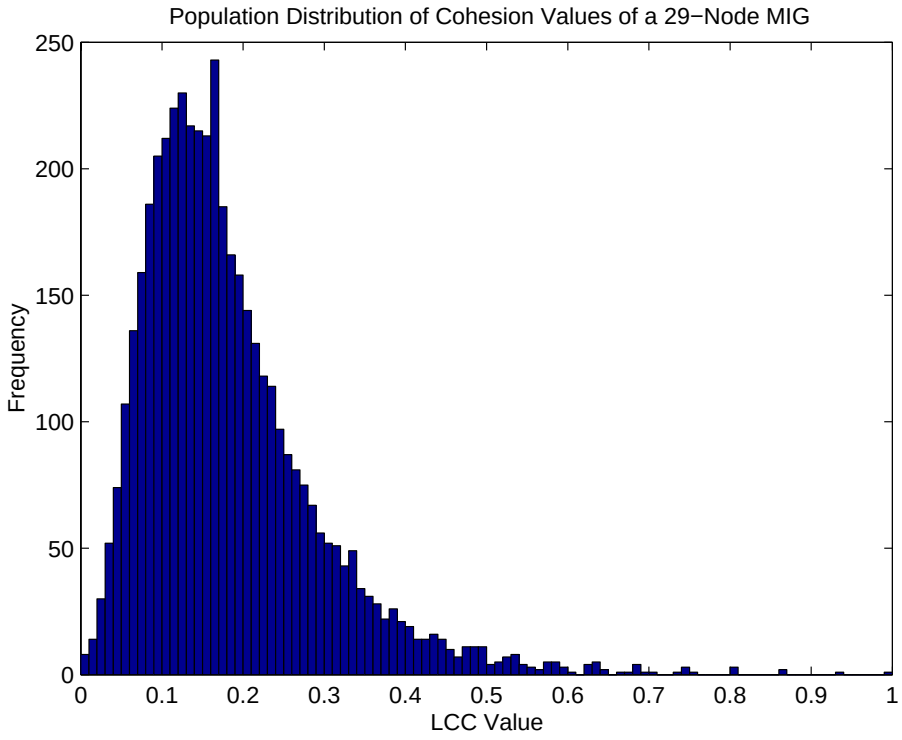


Figure 3.9: Population of Cohesion Values

Chapter 4

Java Models

This section defines the language-specific issues concerning cohesion measurement in Java.

4.1 Java Language Model

4.1.1 Syntactic Organization and Language Features

The top level syntactic organization of Java programs is the package [Arnold96]. A package consists of an optional name, which may qualify it with respect to enclosing super packages. In addition, any number of import declarations and type (class and interface) declarations may exist. If a source file does not possess a package declaration, the classes and interfaces declared in the "current environment" are assigned to the unnamed package. There is at least one unnamed package for each Java program, but the rule for selecting the current unnamed package from which type references may be resolved is compiler-dependent. In most implementations it is the current working directory.

The package construct serves as a way of organizing code for use by clients and to control its visibility. Type declarations (classes and interfaces) which are public are visible to all clients of a package. Type declarations which are not public are package-wide.

The interface construct declares fields and methods available to a client from an object implementing the interface.

In Java, the basic language construct is a class, which consists of methods and instance variables. Unlike C++, there exist no free functions or fields. [Pohl93]. The only way to implement functionality is within a class. Consequently, the subject of measurement is the Java class construct.

Figure 4.1 gives a schematic representation of Java syntactic organization using OMT [Rumbaugh91].

There are two Java language definitions currently in use. Version 1.1 of the Java Development System (JDK) supports inner classes, a construct useful for implementing helper classes. Because this language feature is experimental [Javasoft97], the Java language corresponding to Version 1.0 of the JDK was used by this research.

4.1.2 Method Qualification and Overloading

A qualified method name consists of a base name, an ordered set of types corresponding to the formal parameter list, and a return type. The Java Virtual Machine specification defines a canonical representation (mangling) of the fully qualified method name [Lindholm96]. Use of the canonical qualification in naming makes overloaded methods distinct in the name space of the Java program. In the work of Bieman and Kang [Bieman95], no distinction was made among overloaded methods. This vastly simplifies matters. Unfortunately, method overloading is very common in the Java idiom and must be tackled if any meaningful cohesion results are to be obtained.

4.1.3 Dynamic Method Dispatch

Method dispatch is dynamic in Java. This means the type of the reference is bound at the time of assignment (run-time), and the method lookup depends on the current type of the reference. In this way polymorphism of objects is realized.

Static analysis cannot completely resolve method lookups. If the choice lies between a method in the class being measured, and a method of the same qualified name in a super class, then we consider the type of the sending object to be its declared type at the call site.

The problem with this simplifying approach is that any occurrence of a locally defined, qualified method name in an expression will be resolved in favor of the locally defined method (no polymorphism). Connections between methods in a class may be counted when in fact the relation does not exist. The omission of dynamic method dispatch analysis therefore may result in higher cohesion measurements than are warranted.

It is possible through static analysis to determine the domain of possible method calls and thereby eliminate the local reference [Dean96] [Chambers96].

4.1.4 Type Resolution in Expressions

Another difficulty in resolving qualified method names arises from the fact that an argument to a method call must be evaluated as to its type so that the method qualification and lookup can take place. The argument may itself be an expression. Moreover, an identifier may be an object reference, and its type is determined by its last assignment, rather than its declared type. Data flow is normally used to resolve types in this case.

4.2 Class Size

Another measure of importance to this research is the size of the class being measured. Size is an internal software attribute known to correlate with a number of structure measures [Fenton91]. If a relationship between cohesion and size can be shown, then other relationships may follow.

Historically, program size has been measured in lines of non-comment source code (LOC) [Fenton94].

While LOC is an absolute measure (because it is a bijection to the natural numbers), it suffers from a general failure to meet its representation condition. LOC is highly sensitive to programming style, and without reference to the style of the code's author, the measure is not entirely meaningful. Programming style is an attribute that is little understood, and is unlikely to be measured successfully.

The Java compiler provides an alternative size measure which eliminates programming style as an attribute. The output of the *javac* compiler produces a class file. The tool *javap* decompiles the class file into the byte codes (intermediate instructions) that make up the portable code of the class. The measure is the number of lines of byte code. Only the byte codes for which source code was written appear in the class file. All external references are denoted as such. Unlike other measures of binary code size, this measure is platform independent.

4.3 Applying Cohesion Measures to Java

4.3.1 Cohesion Established Through Class Fields and Methods

A client accesses a class through its class variables (fields), class methods, instance variables (fields), and instance methods. Class fields and methods exist for the loadtime of the class. Instance fields and methods exist only for the lifetime of the object, which is an instance of the class. More precisely, the values of the class fields and the current state of the class methods depend on the history of accesses to the class. By contrast, instance fields and methods have their value and state affected solely by the history of accesses to that particular instance.

One of the principal issues regarding cohesion measurement in Java is whether instance methods may connect through class fields or class methods. (Class methods are prevented from using instance fields by the compiler.) For example, consider the following code:

```
class ClassA
{
    private static int maxSize;
    private static void changeLimit(int newSize) { maxSize += newSize; }

    public void increaseLimit(int amount) { changeLimit(amount); }
    public void decreaseLimit(int amount) { changeLimit(-amount); }
}
```

In class `ClassA` methods `increaseLimit` and `decreaseLimit` are connected through their common use of the class method `changeLimit`, which in turn directly uses the class field `maxSize`.

There are two approaches for treating this kind of cohesion.

- Regard the class field `maxSize` and the class method `changeLimit` as instances of an object created at the loadtime of the class. This object contains the state of the class, but no state of any instances of the class. There is some justification of this view in Java because a class

possesses some properties of an object: its type can be obtained during runtime, and its fields and methods can be dereferenced with respect to the class name. Thus, class fields and methods extend the state of the putative class object.

- Regard the class fields and methods as added state of each instance of the class. Moreover, the class fields and methods have the power to change the state of all live instances of the class. This introduces coupling between objects that is contrary to the notion of objects having discrete state and independent history.

For the purposes of this research we exclude class fields and methods from analysis. This means we also exclude the class method `main`. The `main` method is the entry point for a java program. It may appear in any class. Excluding the main method does not detract from cohesion analysis because `main` acts more like a client to the class.

4.3.2 Cohesion Established Through Inheritance

Another key question concerns the treatment of cohesion through methods and fields defined in an ancestor class. Consider the following fragment:

```
class Point
{
    private int x, y;
    Point(int a, int b) { a = x; b = y; }
    void translate(int a, int b) { x += a; y += b; }
    int getX() { return x; }
    int getY() { return y; }
}
class ColorPoint extends Point
{
    private int color;
    ColorPoint(int a, int b, int c) { super(a, b); color = c; }
    void reflectXAxis() { translate(0, -2*getY()); }
    void reflectYAxis() { translate(-2*getX(), 0); }
}
```

The class `ColorPoint` exhibits high cohesion through the connection of the methods `reflectXAxis` and `reflectYAxis` to the instance variables `x` and `y` in the super class `Point` (the `super` keyword is implicit).

In Java, all classes are inherited from a root class called `Object`. The subclass `ColorPoint` inherits some functionality from `Point`, but gets the rest from `Object`. To what degree do we look at inheritance to resolve questions of cohesiveness? There are three approaches:

- Notionally, the two methods `reflectXAxis` and `reflectYAxis` are not connected in the class `ColorPoint` because `ColorPoint` can be split between them. In fact, a better design would be to push the two methods upstairs to `Point`.

- If we were to look at the immediate superclass for connections, then why not the super-superclass and so on right up to `Object`? This approach gives a specious view of instance variable usage in Java, since all fields and methods not hidden or overridden are available to the subclass. In fact, such methods as `java.lang.Object.hashCode` and `java.lang.Object.equals` are frequently used in their pure form, and their presence in expressions could give arbitrarily high values for cohesion of subclasses.
- We could construct a vector measure giving cohesion through inherited fields and methods arising from each of N levels of inheritance.

This research addresses only *Local Cohesion*, or cohesion arising from definitions available within a class. This approach diverges from that of [Bieman95], which admits to optional measurement of connections through super class instance variables.

4.3.3 Cohesion Established Through Composition

Consider the following container class which is composed of a vector of complex values and performs certain vector operations on it.

```
package ComplexNumber;
import java.util.Vector;

public class ComplexVector
{
  private Vector v = new Vector(0);

  public ComplexVector() {}

  public void addElement(Complex a) { v.addElement(a); }
  public int size() { return v.size(); }

  public void add(ComplexVector u)
  {
    for (int i = 0; i < v.size(); ++i)
    {
      ((Complex)v.elementAt(i)).add((Complex)v.elementAt(i));
    }
  }
}
```

The class `ComplexVector` has a component of class `Vector`. It also has an `ComplexVector.add` method which calls `Vector.size`. The `ComplexVector.size` method also calls `Vector.size`. Do `ComplexVector.size` and `ComplexVector.add` not connect because they share no common instance variables but share only calls to an object outside of this class? Or do the methods directly connect

through their common use of the instance variable `v`? Our model treats this case as a *use* rather than a *call*. Consequently, `ComplexVector.size` and `ComplexVector.add` directly connect.

4.3.4 Cohesion and Helper Classes

Although we have made the decision not to measure cohesion across class boundaries, strict adherence can lead to anomalies.

Helper classes are classes whose instances are paired with instances of a principal class. They are not used by any other class. These kinds of classes are nevertheless ignored for the purposes of analyzing cohesion, but they tend to add to the cohesiveness of a principal class. Consider the following class `TreeIterator`, which is a package-only iterator object associated strictly with class `Tree` and its subclasses, and modeled after the iterator design pattern of [Gamma95].

```
package Tree;
import java.util.Enumeration;
import java.util.Vector;

class TreeIterator implements Enumeration
{
  private Tree tree;
  private Vector vector;
  private Enumeration iterator;

  TreeIterator(Tree t)
  {
    tree = t;
    vector = new Vector(tree.size());
    if (!tree.isEmpty())
    {
      tree.setRoot();  // set the current pointer to the root
      tree.preorder(vector);   // fill up vector with the elements in preorder
    }
    iterator = vector.elements();
  }

  public boolean hasMoreElements() { return iterator.hasMoreElements(); }
  public Object nextElement() { return iterator.nextElement(); }
  public Enumeration elements() { return iterator; }

  public String toString()
  {
    String s = new String("");
    while( hasMoreElements() ) { s += ((nextElement()).toString() + "\n"); }
    return s;
```

```
  }
}
```

In the principal class `Tree` there is a method called `toString` which dumps the tree, and a method called `elements` which returns an iterator object. The text for this code is :

```
public class Tree
{
  ...
  public String toString() { return (new TreeIterator(this)).toString(); }
  public Enumeration elements() { return (new TreeIterator(this)).elements(); }
  ...
}
```

As can be seen from the code for the `TreeIterator` constructor, any method in `Tree` which creates an iterator object will indirectly use the method `Tree.size` and therefore the instance variable `Tree.count`. Thus, the methods `Tree.toString` and `Tree.elements` are indirectly connected without being measured as such.

Incidentally, the inner classes feature of Java 1.1 lends itself very well to the inclusion of helper classes such as iterators and exceptions.

4.3.5 Cohesion Established Through Constants

A cohesion measure should not try to connect methods which use common constants but otherwise have no connection. Constants have no state and are immutable for the lifetime of an object. If a class were factored, any shared constants can be duplicated. (In Java constants can be moved to an interface so that a single copy in the source code is maintained.)

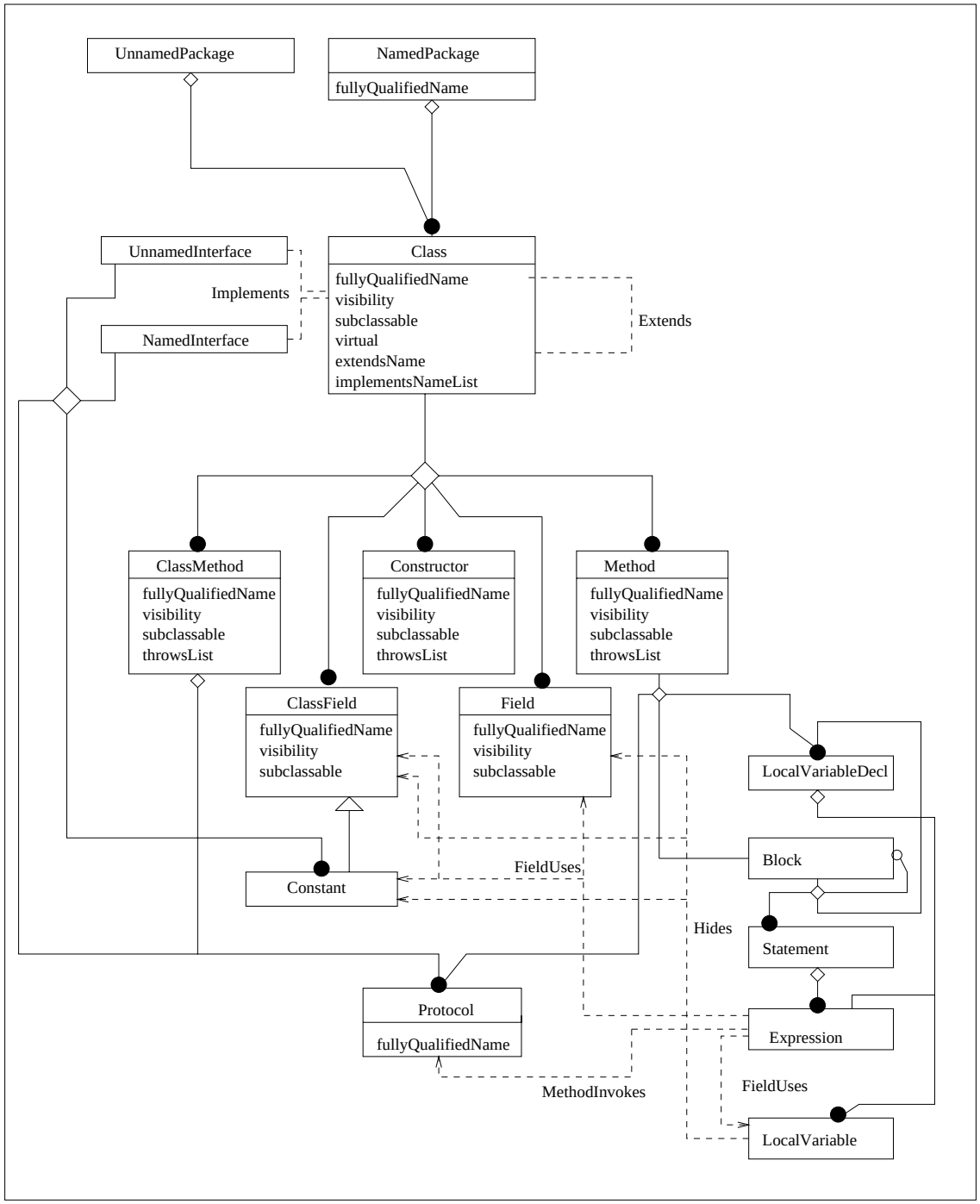


Figure 4.1: OMT Diagram of Java Object Model Used for Cohesion Measurement

Chapter 5

Celebes, a Java Metrics Platform

The research produced a static analysis tool called *Celebes* [1]. This chapter presents a brief overview of the tool's implementation and a tutorial covering a simple example.

5.1 Overview

Celebes is a static analysis tool which takes as input Java source files and compiled byte code files, and generates method call and field information. This information is used to compute the cohesion measures. The tool leverages the compiler in order to fully distinguish overloaded method names, resolve superclass and self references, resolve actual parameter types for method protocols, and to resolve actual type of the calling object at a call site.

Figure 5.1 shows a block diagram of *Celebes* components. The inputs to the tool consist of compilable Java source codes. These files are run through the *javacc* parser generator and *jjtree* abstract syntax tree (AST) builder. These tools along with a Java 1.0 language grammar are supplied by Sun Microsystems, Inc [Javacc97]. *javacc* is an LL(k) parser.

Celebes uses AST output by *jjtree* as the starting point for a tree transformation that changes syntactic entities into semantic ones. For instance, a branch of the AST defining a method declaration is changed into an object of type `MethodNode` which embeds its qualified name, constituent local variable declarations, and the top level statement block (which itself is composed of statements and embedded blocks). The closure of this recursive process is a decorated syntax tree (DST).

The DST retains the structure of the class being analyzed, but also propagates information through the tree. Among the results pushed down the tree are:

- Qualified package, class names.
- Instance variable declarations.
- Visible method declarations.

[1] *Celebes* is named after a coffee rich island in the Java Sea. Its modern name is Sulawesi, now part of Indonesia.

- Enclosing local variables which may kill references to instance variables.

Among the results pushed back up the tree are:

- Qualified method names.
- Instance variable uses by qualified method.
- Visible qualified method uses by qualified method.

Cohesion measures depend on method call and field use data propagated up the structure tree to the class level. These are assembled into graphs, from which the measures are computed. Metrics are extracted from each class and reported in a metrics file.

Size is obtained by counting the number of lines of java byte code in the associated class file decompilation as produced by *javap*. The decompiled files could be parsed within *Celebes* by building another (much simpler) parser generator.

5.2 Tutorial

This section shows the process of cohesion analysis and measurement of an example class using *Celebes*. The source code for the test class is found in *ClassO.java*:

```
class Class0
{
  int field0, field1, field2, field3, field4, field5;
  static int mode;

  void m0() { field0 = field1 = 1; }
  void m1() { field2 = 2; }
  void m2() { field3 = field4 = field5 = 2; }
  void m3() { field2 = 4; }

  void m4() { m0(); }
  void m5() { m4(); }
  void m6() { m5(); m2(); }
  void m7() { m1(); }
  void m8(int a)     { m3(); }
  void m8(boolean a) { m3(); }

  public static void main(String[] args) {}
}
```

First, compile the source file:

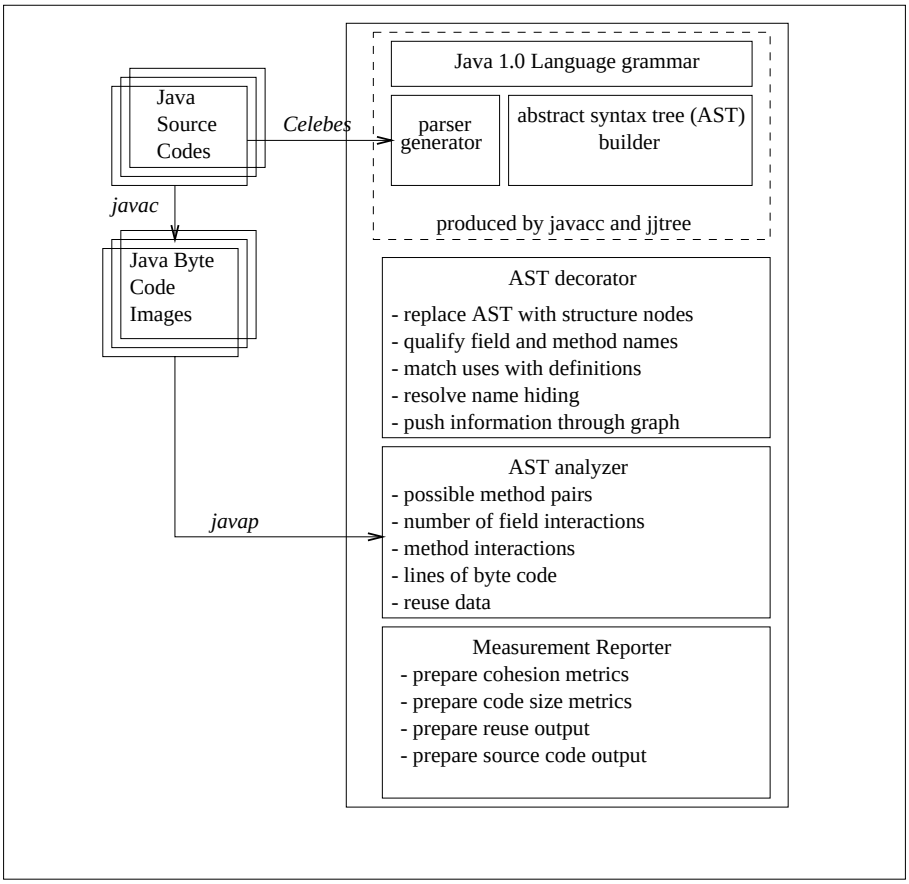


Figure 5.1: Block Diagram of Major Components, Inputs, and Outputs to *Celebes*

```
javac Class0.java
```

Celebes runs on the Java source code file. If the file contains multiple class declarations, these will be analyzed separately. However, you must generate a byte code file for each class file produced by the compiler.

```
javap -c -p Class0 > Class0.bc
```

Now run *Celebes*.

```
scruggs> java Celebes -do -dir=$CELEBES/examples Class0.java
```

```
Celebes Java Metrics Platform v0.1a
(c) 1997 Colorado State University.
Run ``celebes -about'' for information and usage restrictions.
Reading from /s/bach/h/proj/reuse/celebes/examples/Class0.java
Opening /s/bach/h/proj/reuse/celebes/examples/Class0.parse for output.
Decorating syntax tree...
Found 1 classes.
Processing byte codes for ...Class0
Did not find : Class0.main([Ljava.lang.String;)V
Requalifying Class0.main([LString;)V with Class0.main([Ljava.lang.String;)V
Found 1 classes.
Writing parse tree...
Generating cohesion information for Class0
Writing parse time information...
Parse time was 5 seconds.
scruggs>
```

The first output file is the source code parse file *Class0.parse*. This gives the overall structure of the program. A portion of the file is shown below, including the class information, field information, and information for one method. The method consists of its source code body (under the `Block` subtree), and its byte code body (under the `MethodBody` subtree). The method body subtree incorporates field use and method call information, which is passed up the tree to the class level. Field use and method call data are used to generate cohesion metrics.

The cohesion computation excludes from consideration static (class) fields, constants, static (class) methods, and private methods.

```
...
Class: "Class0"
 public Class0 extends java.lang.Object
 implements []
 Fields: [Class0.field0, Class0.field1, Class0.field2, Class0.field3,
          Class0.field4, Class0.field5, Class0.mode]
 Methods: [Class0.m0()V,  Class0.m1()V,  Class0.m2()V, Class0.m3()V,
           Class0.m4()V,  Class0.m5()V,  Class0.m6()V, Class0.m7()V,
           Class0.m8(I)V, Class0.m8(Z)V, Class0.main([Ljava.lang.String;)V,
           Class0.<init>()V, Class0.<clinit>()V]
 Non-Class Non-Constant Fields: [Class0.field0, Class0.field1,
                  Class0.field2, Class0.field3, Class0.field4,
                  Class0.field5]
 Visible Non-Class Methods: [Class0.m0()V, Class0.m1()V, Class0.m2()V,
                             Class0.m3()V, Class0.m4()V, Class0.m5()V,
                             Class0.m6()V, Class0.m7()V, Class0.m8(I)V,
                             Class0.m8(Z)V]
 Aggregate size (byte code statements) = 45
```

```
Field: "Class0.field0"
Field: "Class0.field1"
Field: "Class0.field2"
Field: "Class0.field3"
Field: "Class0.field4"
Field: "Class0.field5"
Field: "Class0.mode"
Method: "Class0.m0()V"
 Calls: []
 Uses:  [Class0.field1, Class0.field0]
 Size = 7
 Block: "Block0"
  Statement: "Stmt1" Empty
   Expression: "Expr2"
 MethodBody: "Class0.m0()VBody"
  Calls: []
  Uses:  [Class0.field1, Class0.field0]
  Size = 7
...
```

Celebes also produces a parse tree for the byte code file. Portions of the file *Class0.p* are shown below. Notice the call and use information embedded within this tree. Method and field names are qualified with the enclosing class and package names throughout the *Celebes* data structures. Method names are also mangled with their calling protocol and return type. In this way, overloaded method names are unique. The mangling scheme is the same as that used by the Java compiler.

```
...

MethodBody
 ResultType
  voidKeyword
 Name
  m3
 FormalParameters
 Statement
  Instruction
 Statement
  Instruction
 Statement
  Instruction
   UseStatement
    <Field Class0.field2 I>
 Statement
  Instruction
 StatementCount
```

```
MethodBody
 ResultType
  voidKeyword
 Name
  m4
 FormalParameters
 Statement
  Instruction
 Statement
  Instruction
   CallStatement
    <Method Class0.m0()V>
 Statement
  Instruction
 StatementCount
```

You may now examine the metrics output file *Class0.metrics* produced by *Celebes*. The first part of report gives the summary of the class cohesion and class size measures.

```
Cohesion Measures for Class: Class0
nLinks         = 45
nDirectLinks   = 17
nIndirectLinks = 3
mTCC           = 0.37777779
mLCC           = 0.44444445
nSize          = 45
nMethods       = 10
mMethodSize    = 4.5
```

The All Uses Graph shows the direct and indirect usage of fields by visible non-static methods. The indirect usage relations are obtained by taking the transitive closure of the graph of direct uses. "T" means that a usage relation exists. This graph is made anti-reflexive.

```
All Uses Graph =
DiGraph:
               C C C C C C C C C C C C C C C C
               l l l l l l l l l l l l l l l l
               a a a a a a a a a a a a a a a a
               s s s s s s s s s s s s s s s s
               s s s s s s s s s s s s s s s s
               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
               . . . . . . . . . . . . . . . .
               f f f f f f m m m m m m m m m m
```

```
                i i i i i i 0 1 2 3 4 5 6 7 8 8
                e e e e e e ( ( ( ( ( ( ( ( (
                l l l l l l ) ) ) ) ) ) ) I Z
                d d d d d d V V V V V V V ) )
                0 1 2 3 4 5               V V
ClassO.field0   F F F F F F T F F F T T T F F F
ClassO.field1   F F F F F F T F F F T T T F F F
ClassO.field2   F F F F F F F T F T F F F T T T
ClassO.field3   F F F F F F F F T F F F T F F F
ClassO.field4   F F F F F F F F T F F F T F F F
ClassO.field5   F F F F F F F F T F F F T F F F
ClassO.m0()V    T T F F F F F F F F T T T F F F
ClassO.m1()V    F F T F F F F F F F F F F T F F
ClassO.m2()V    F F F T T T F F F F F F T F F F
ClassO.m3()V    F F T F F F F F F F F F F F T T
ClassO.m4()V    T T F F F F T F F F F T T F F F
ClassO.m5()V    T T F F F F T F F F T F T F F F
ClassO.m6()V    T T F T T T T F T F T T F F F F
ClassO.m7()V    F F T F F F F T F F F F F F F F
ClassO.m8(I)V   F F T F F F F F F T F F F F F F
ClassO.m8(Z)V   F F T F F F F F F T F F F F F F
```

Next, each method abstraction is produced by forming the set of fields used by the method as obtained from the All Uses Graph.

```
Method Abstractions =
ClassO.m0()V: [ClassO.field0, ClassO.field1]
ClassO.m1()V: [ClassO.field2]
ClassO.m2()V: [ClassO.field3, ClassO.field4, ClassO.field5]
ClassO.m3()V: [ClassO.field2]
ClassO.m4()V: [ClassO.field0, ClassO.field1]
ClassO.m5()V: [ClassO.field0, ClassO.field1]
ClassO.m6()V: [ClassO.field0, ClassO.field1, ClassO.field3,
               ClassO.field4, ClassO.field5]
ClassO.m7()V: [ClassO.field2]
ClassO.m8(I)V: [ClassO.field2]
ClassO.m8(Z)V: [ClassO.field2]
```

The Directly Connects Graph is computed by finding which method abstractions have pair-wise non-empty intersections. The number of such connections (after accounting for symmetry) is the "direct links" number in the class metrics summary above.

```
Directly Connects Graph =
DiGraph:
               C C C C C C C C C C
               l l l l l l l l l l
               a a a a a a a a a a
               s s s s s s s s s s
               s s s s s s s s s s
               0 0 0 0 0 0 0 0 0 0
               . . . . . . . . . .
               m m m m m m m m m m
               0 1 2 3 4 5 6 7 8 8
               ( ( ( ( ( ( ( ( ( (
               ) ) ) ) ) ) ) ) I Z
               V V V V V V V V ) )
                                 V V
Class0.m0()V   F F F F T T T F F F
Class0.m1()V   F F F T F F F T T T
Class0.m2()V   F F F F F F T F F F
Class0.m3()V   F T F F F F F T T T
Class0.m4()V   T F F F F T T F F F
Class0.m5()V   T F F F T F T F F F
Class0.m6()V   T F T F T T F F F F
Class0.m7()V   F T F T F F F F T T
Class0.m8(I)V  F T F T F F F T F T
Class0.m8(Z)V  F T F T F F F T T F
```

The Indirectly Connects Graph is computed by obtaining the transitive closure of the Directly Connects Graph and subtracting from this result the Directly Connects Graph. Also, the graph is made anti-reflexive. The "indirect links" value reported in the summary is computed by counting the number of non-empty method abstraction intersections (after accounting for symmetry) .

```
Indirectly Connects Graph =
DiGraph:
                C C C C C C C C C C
                l l l l l l l l l l
                a a a a a a a a a a
                s s s s s s s s s s
                s s s s s s s s s s
                0 0 0 0 0 0 0 0 0 0
                . . . . . . . . . .
                m m m m m m m m m m
                0 1 2 3 4 5 6 7 8 8
                ( ( ( ( ( ( ( ( ( (
                ) ) ) ) ) ) ) ) I Z
                V V V V V V V V ) )
                                V V
Class0.m0()V    F F T F F F F F F F
Class0.m1()V    F F F F F F F F F F
Class0.m2()V    T F F F T T F F F F
Class0.m3()V    F F F F F F F F F F
Class0.m4()V    F F T F F F F F F F
Class0.m5()V    F F T F F F F F F F
Class0.m6()V    F F F F F F F F F F
Class0.m7()V    F F F F F F F F F F
Class0.m8(I)V   F F F F F F F F F F
Class0.m8(Z)V   F F F F F F F F F F
```

Finally the cohesion numbers are computed by dividing the number of actual links (direct for TCC and direct+indirect for LCC) by the number of possible links.

5.3 Extending *Celebes*

5.3.1 Extension Hooks

Celebes is designed to serve as a Java metrics platform onto which further measures can be added. The tool maintains a parse tree integrating source code and byte code. A parse tree consisting of the byte code of each method is attached to the method's structure node. Measurements are computed by traversing the structure nodes of the parse tree and gathering or depositing information. An iterator-visitor object is defined for each measurement task.

Celebes implements several families of iterator-visitors. The basic pattern is inspired by the iterator and visitor patterns of object-oriented design [Gamma95]. These can be extended or subclassed to add additional metrics capabilities.

- The *Counter* iterator-visitor counts the number of structure nodes of a certain type. One appli-

cation is a counter class which counts the visible instance methods of each class.

- The *Identifier* iterator-visitor builds a list of names of structure nodes of a certain type. An application is the list of visible instance methods in a class.
- The *Collector* pushes information around the parse tree. An example application is the consolidation of call and use information from the method body into the method node. The collector works by visiting first the child node (the method body), followed by the parent node (the method structure node). The iterator-visitor object holds the information in its internal state between visits. A code scrap from the class `CallCollector` illustrates the concept:

```
public void apply(Object t)
{
  try
  {
    BodyNode node = (BodyNode)t;
    // apply the server function, set the buffer contents.
    buffer = node.offerCallData();
    return;
  }
  catch (ClassCastException error2) { } // visit does not apply
  try
  {
    MethodNode node = (MethodNode)t;
    // apply the client function, read the buffer contents.
    node.acceptCallData(buffer);
    reinit();
    return;
  }
  catch (ClassCastException error0) { }// visit does not apply
}
```

  The parse tree object launches the collector according to a post-order tree traversal. The post-order traversal guarantees that the child node will be visited first, collecting the relevant call and use data. The parent node will be visited after each child visit, and the call and use data are deposited with it. The algorithm is correct because every method node has exactly one method body child node.

```
CallCollector callCollector = new CallCollector();
this.visitInPostorder(callCollector);
```

- The *Reporter* iterator-visitor builds an output string representation of the measurements collected at the specified level (package, class, method). The result is then sent to the output stream once the parse tree traversal is complete.

Additional measurement services can be defined by subclassing one of the above families.

5.3.2 Example Extension

We add a measurement capability which distinguishes vacuous visible instance methods (methods with no body). It is simplest to subclass the closest `Collector` iterator-visitor type, which is `VisibleInstanceMethodCollector`, and replace its `apply` method. The code for `VisibleNonEmptyInstanceMethodCollector` now looks like this:

```
public class VisibleNonEmptyInstanceMethodCollector
extends VisibleInstanceMethodCollector
{
  public void apply(Object t)
  {
    ...  // same as superclass
    try
    {
      MethodNode node = (MethodNode)t;
      // apply the client function, set the buffer contents.
      if ( !node.isPrivate() &&  !node.isStatic() &&
           node.getSize() > 0  // this part is new
         )
      {
        buffer.union((Set)node.offerMethodDecl());
      }
      return;
    }
    catch (ClassCastException error1) { }// visit does not apply
    catch(IncompatibleTypeException e) { e.report(); } // bad
  }
}
```

The structure node which contains the summary information of the method is called the `MethodNode`. In *MethodNode.java* we add a new method to report the number of byte code statements:

```
public int getSize() { return statementCount; }
```

Now we add a new field in the class object to hold the set containing the names of all visible, non-empty, instance methods. In *ClassNode.java* we add :

```
private Set visibleNonEmptyInstanceMethodList;
```

To *ClassNode.java* we also add the accepting service for the collector object:

```
public void acceptVisibleNonEmptyInstanceMethodDecl(Object buffer)
```

```
{
  try
  {
    // merge propagating info
    visibleNonEmptyInstanceMethodList.union((Set)buffer);
  }
  catch(IncompatibleTypeException error) {} // always String elements
}
```

At the top level in *Celebes.java*, we create the collector object and perform the visitation on the parse tree in post-order:

```
VisibleNonEmptyInstanceMethodCollector
  visibleNonEmptyInstanceMethodCollector
  = new VisibleNonEmptyInstanceMethodCollector();
dst.visitInPostorder(visibleNonEmptyInstanceMethodCollector);
```

After the program runs this code over the complete parse tree, we have collected the set of visible non-empty instance methods of each class at the class level. To modify the cohesion measurement calculations, the internals of `ClassNode` can be updated, replacing occurrences of `visibleInstanceMethodList` with `visibleNonEmptyInstanceMethodList`. Alternately, with more effort one can augment the measurement functions by adding code to separately calculate cohesion measures for non-empty methods. To do this, duplicate the occurrences of code concerning `visibleInstanceMethodList` with similar code using `visibleNonEmptyInstanceMethodList`. Be sure to keep a separate count of the number of methods.

Chapter 6

An Experiment to Investigate the Relationship between Class Cohesion and Class Size

A new measure should undergo a process of empirical validation [Fenton94] [Kitchenham95]. This kind of validation does not require that the measure be shown to correlate with some external attribute it is thought to affect. Rather, we wish to see whether cohesion values obtained through measurement of real programs are consistent with the values predicted by our model of cohesion. To this end we design an experiment which tests the relationship between cohesion of a class and another internal attribute: class size.

6.1 Experimental Design

Chapter 4 defines a cohesion model specific to Java. Section 4.2 formulates a model of class size that is specific to Java. The subjects of the study are two publicly available Java software packages. The classes in these packages are of non-trivial size. Together they constitute a large and representative sample of well-designed classes. These properties should give the experiment reasonable external validity.

The null hypotheses we seek to disprove state that there exists a relationship between class size (the independent variable) and TCC or LCC (the dependent variables). The alternate hypotheses state that there exists no relationship.

$$H_0 : TCC = \beta_0 + \beta_1(Size) \implies \beta_1 \neq 0.$$

$$H_0 : LCC = \beta_0 + \beta_1(Size) \implies \beta_1 \neq 0.$$

We reason as follows. Assume that the number of methods in a class and class size are strongly correlated (anecdotal evidence strongly suggests such a relationship, but this was not examined empirically). The Bieman-Kang cohesion measures react differently to various scenarios under which the number of

methods is increased. Four are listed below. In all, the effect on cohesion in the change in the number of methods should be minimal.

- If there is a uniform probability of coincidental components (say 1/3 of the methods in the class are connectionless), then cohesion rises slightly: For $N = 6$, $E[LCC] = 0.4$, for $N = 12$, $E[LCC] = 0.55$.
- If there is a uniform probability of an edge being a connection edge, say 3/4, then for $N = 4$, $E[LCC] = 0.75$, for $N = 8$, $E[LCC] = 0.75$. In other words, cohesion is invariant.
- If there is uniform probability of a node being in one of two factors, say 1/2, then for $N = 4$, $E[LCC] = 0.33$, for $N = 12$, $E[LCC] = 0.42$. Cohesion in this case is higher.
- If there is a constant number of method interactions, say 2 (*ie,* through a ring of connections), then for $N = 6$, $E[LCC] = 1$ and $E[TCC] = 0.4$, for $N = 12$, $E[LCC] = 1$, $TCC = 0.18$. In this case cohesion is lower.

6.2 Experimental Results

Celebes produced automatic Bieman-Kang cohesion measures on two data sets. Certain source files of these data sets could not be analyzed due to use of obsolete or experimental language features.

6.2.1 JDK Demonstration Suite

The demonstration suite in the JDK 1.1 [Javasoft97] is a collection of programs (actually so-called applets) which are fairly short. Each performs a distinct function. The classes in this suite are generally small and much code is to be found in the `main` method. The `main` method is static so it is not used for cohesion measurement.

Of the 61 classes in this data set, 43 have two or more methods, according to Table 6.1. The remainder are not further analyzed because classes with fewer than two methods necessarily have zero cohesion.

The measurable classes have mean LCC of 0.55 and median of 0.50. The TCC mean is 0.48 and median is 0.33. The series have a standard deviation of 0.39 and 0.37, respectively. The distribution of cohesion values is uniform except at the end points (lots of 1.0 and 0.0 cohesion outcomes).

The distribution of size measures for the remaining data set is clearly non-Gaussian, is concentrated around smaller values, and is highly variable, according to Figure 6.1.

A regression was performed to determine the effect on cohesion of class size, as measured by the aggregate number of byte code statements to be found in the visible instance methods of the class. Figure 6.2 shows a scatter plot for the LCC measures (asterisk) and TCC measures (circle), and size. Our null hypothesis states that the regression line (parameter β_1) is non-zero.

The F-statistic for the regression ($F_{0.05,1,41}$) gives values of 0.6057 and 0.0308 for LCC and TCC, respectively. These results are far too weak conclude the null hypotheses that the slope parameter

$\beta_1 \neq 0$. Consequently, we conclude the alternate hypothesis that there is no relationship between class cohesion and class size.

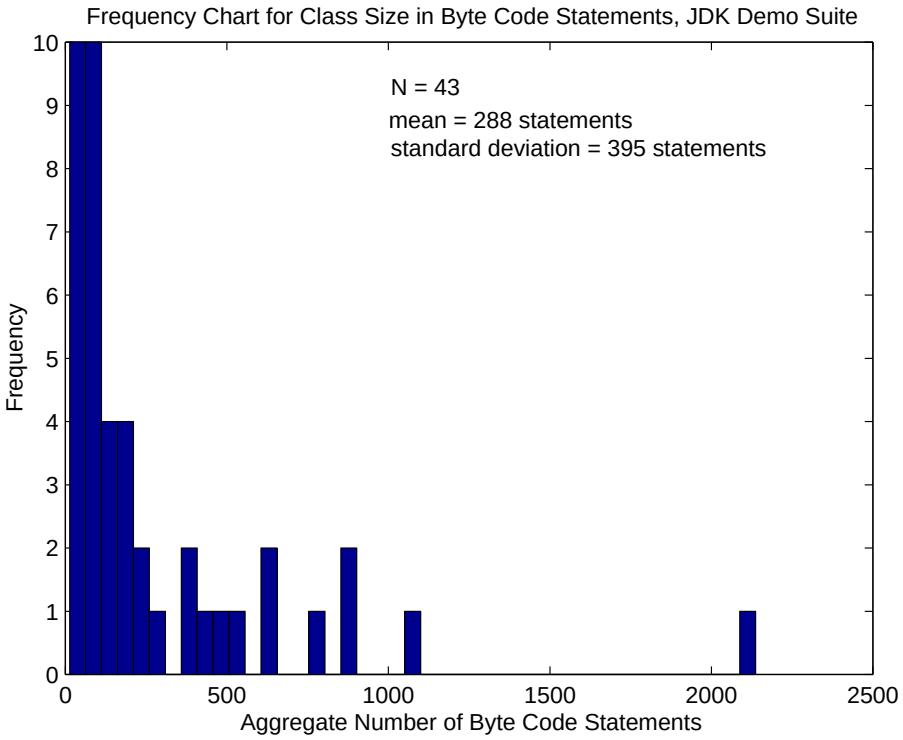


Figure 6.1: Class Size Histogram, JDK Demo Suite

6.2.2 JGL Class Library

The JGL version 1.1 class library [ObjectSpace97] consists of reusable data structures. Most classes are small. Table 6.2 shows that of 116 classes, 73 have fewer than two methods and are trivially non-cohesive. These are not used for measurement. There are 43 classes with two or more methods.

The measurable classes exhibit a mean LCC of 0.65 and median of 0.77. TCC mean is 0.62 and median is 0.72. TCC is much closer than in the JDK data set because many classes in this data set have no indirect links, and TCC is the same as LCC. Both series have a standard deviation of 0.25. The distribution of cohesion values is unclear but possibly Gaussian.

Size measures for the remaining classes are non-Gaussian, are concentrated near zero, and are highly variable, according to Figure 6.3.

A regression was performed to determine the relationship on cohesion of class size, as measured by the aggregate number of byte code statements to be found in the visible instance methods of the class. Figure 6.4 shows a scatter plot for the LCC measures (asterisk) and TCC measures (circle).

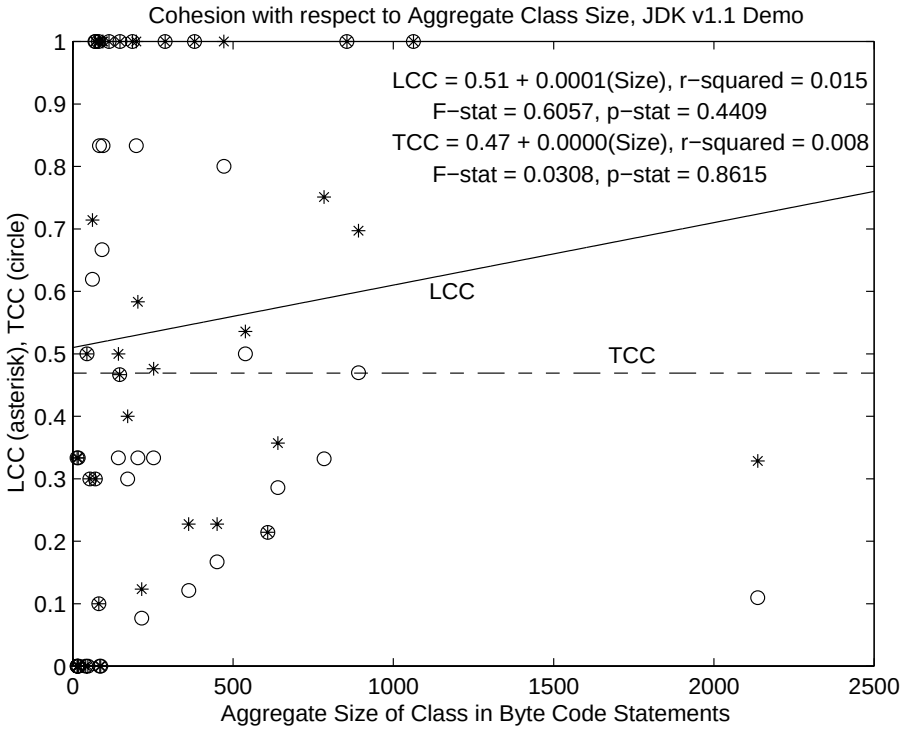


Figure 6.2: Cohesion as a Function of Class Size, JDK Demo Suite

As before, the F-statistic for the regression ($F_{0.05,1,41}$) gives values of 2.1191 and 3.8181 for LCC and TCC, respectively. These results are not as extreme as with the JDK data set, but they are nevertheless too weak to conclude the null hypotheses that the slope parameter $\beta_1 \neq 0$. Once again we conclude the alternate hypotheses that there is no relationship between class cohesion and class size.

6.3 Analysis of General Results

The experimental results confirm the hypotheses proposed for this research: that class size and class cohesion are unrelated. Clearly, factors other than size affect cohesion.

Another question is whether the choice in data sets affects cohesion. We expect the cohesion sample means of the two data sets to be similar. The cohesion distributions both appear to be random but not necessarily Gaussian. Assuming this is an artifact of sampling and the populations are in fact Gaussian, a one-way ANOVA test concludes that the LCC means are alike only at the 85% confidence level. TCC means are alike at the 95% confidence level. This weakly supports the conclusion that cohesion is independent of the data sets from which it is measured.

The lack of a Gaussian sample distribution in the independent variable Class Size amounts to a failure to ensure the properties required of the linear regression model. Indeed, the size histograms suggest a

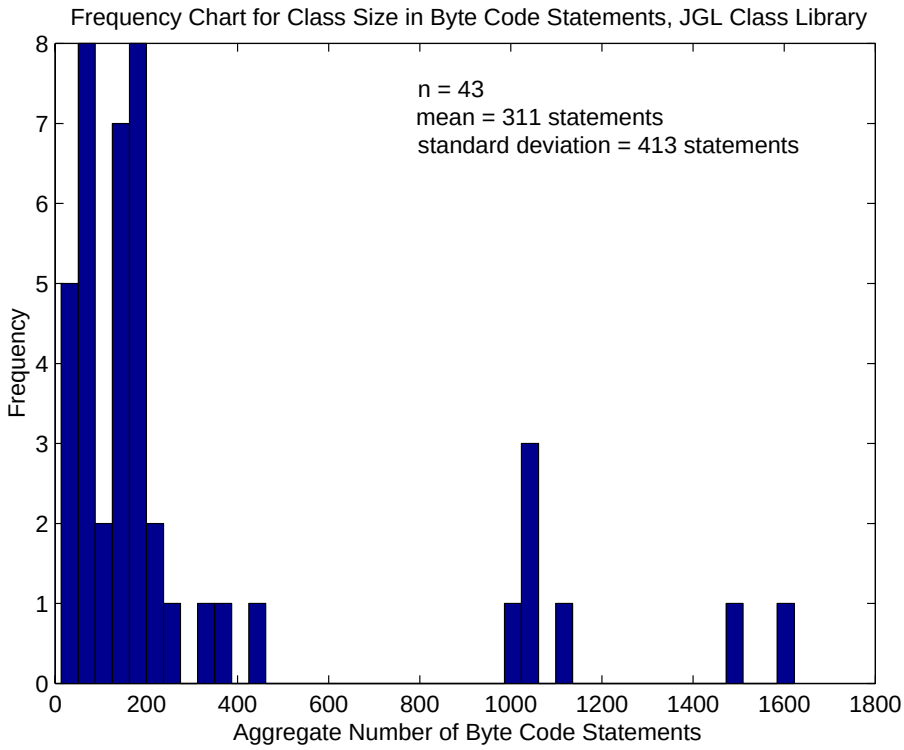


Figure 6.3: Class Size Histogram, JGL Class Library

negative exponential distribution. It is possible that a sample including only classes above a certain threshold size may yield proper Gaussian properties. On the other hand, the population of class sizes may not be Gaussian at all.

The high variance of size measures in the data sets makes any statistical determination of linear relationships between size and cohesion difficult if not impossible. It may be that a much larger sample is needed to reduce size variance.

There are many cases in which a class is stripped of all constructors and static methods (including `main`), leaving nothing left to measure. One class with no instantiable state is `ArcTest` in the JDK test suite. Fully 48% of the 177 classes measured failed to yield instantiable methods that could be measured for cohesion. Any class cohesion measure which excludes the static state of a class from consideration excludes from measurement a large proportion of the available classes.

An anomaly arises from the presence of a solitary instantiable method in the class. In the case of `ShapeTest`, two constructors and one visible method share access to two instance variables. The class is clearly not factorable. However, due to the anti-reflexive property of the cohesion measure, there are no method links and the cohesion is zero.

The presence of vacuous methods (methods lacking any body) can greatly affect cohesion measurements. For example, consider the class `TicTacToe`. Four mouse call backs are declared which do nothing

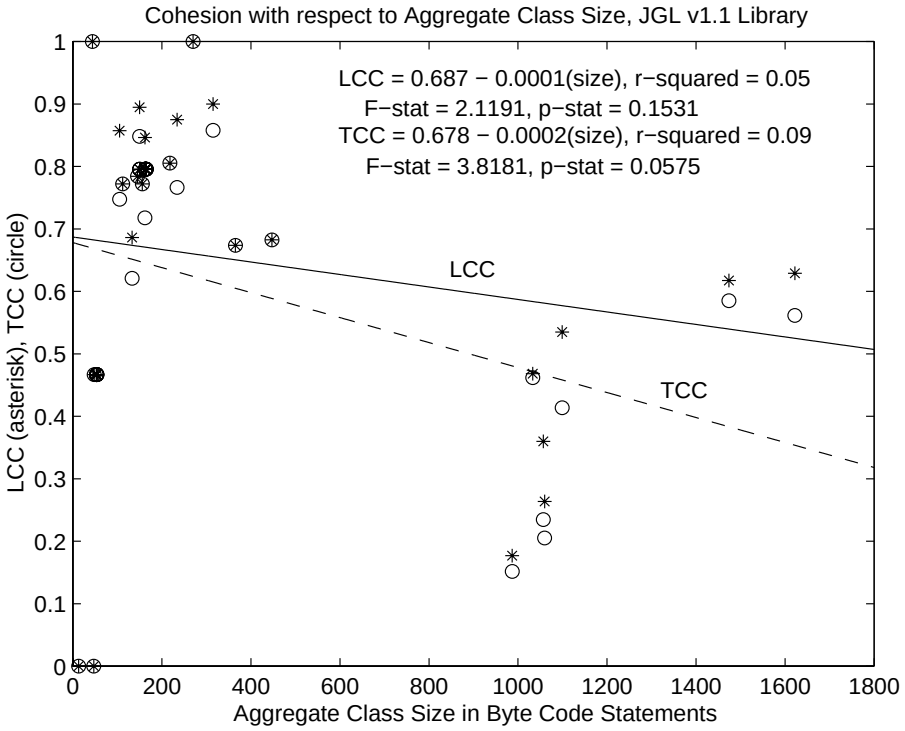


Figure 6.4: Cohesion as a Function of Class Size, JGL Class Library

except provide an implementation for the `MouseListener` interface. This increases the number of visible instantiable methods from 8 to 12. Since the number of links is proportional to the square of the number of methods, this increases the number of links from 28 to 66! Were it not for these vacuous methods, the LCC measure of 0.22 would be 0.68, and the TCC measure of 0.17 would be 0.39. In Java, cohesion is clearly sensitive to language and design requirements that impose vacuous methods on a class implementation. In our view, such methods should be excluded from analysis.

One question that arises in cohesion measurement is whether highly cohesive modules are more readily reused. This question was studied in [Bieman95], which hypothesized that high cohesion leads to high reuse. The study, which examined private reuse through inheritance, found the opposite: the most reused modules were of those having low cohesion. Table 6.4 shows a statistically insignificant sample of those classes in the JDK data set which subclass other classes within the data set. These data suggest the same conclusion. Further work on this topic would require a data set exhibiting more inheritance than this one. Incidentally, there was no private reuse in the JGL data set.

Class name	LCC	TCC	direct links	indirect links	links	number of visible instance methods	instance size (byte codes)	average method size (byte codes)
AniArea	0.5	0.33333334	2	1	6	4	142	35.5
Animator	0.32859176	0.10953058	77	154	703	38	2137	56.236843
AppletFrame	0.0	0.0	0	0	0	1	12	12.0
ArcCanvas	1.0	1.0	1	0	1	2	186	93.0
ArcCard	0.0	0.0	0	0	0	0	0	0.0
ArcControls	0.0	0.0	0	0	0	1	20	20.0
ArcDegreePanel	1.0	1.0	1	0	1	2	147	73.5
ArcPanel	0.0	0.0	0	0	0	1	220	220.0
ArcTest	0.5	0.5	3	0	6	4	44	11.0
BidirBubbleSortAlgorithm	0.0	0.0	0	0	0	1	110	110.0
Blink	0.4	0.3	3	1	10	5	171	34.2
BubbleSortAlgorithm	0.0	0.0	0	0	0	1	57	57.0
ButtonFilter	0.35714287	0.2857143	8	2	28	8	640	80.0
CardPanel	0.0	0.0	0	0	1	2	87	43.5
CardTest	1.0	1.0	1	0	1	2	71	35.5
Chart	1.0	1.0	1	0	1	2	855	427.5
ClickArea	1.0	0.8333333	5	1	6	4	83	20.75
Clock2	0.21428572	0.21428572	6	0	28	8	608	76.0
ColorUtils	0.0	0.0	0	0	0	0	0	0.0
DelayedSoundArea	1.0	0.6666667	4	2	6	4	91	22.75
DescriptionFrame	0.33333334	0.33333334	1	0	3	3	17	5.6666665
DitherCanvas	0.3	0.3	3	0	10	5	70	14.0
DitherControls	0.0	0.0	0	0	3	3	40	13.333333
DitherTest	0.53571427	0.5	14	1	28	8	538	67.25
Edge	0.0	0.0	0	0	0	0	0	0.0
FileFormatException	0.0	0.0	0	0	0	0	0	0.0
Graph	1.0	1.0	6	0	6	4	288	72.0
GraphApplet	0.0	0.0	0	0	1	2	47	23.5
GraphPanel	0.6969697	0.46969697	31	15	66	12	891	74.25
GraphicsCards	0.0	0.0	0	0	0	0	0	0.0
GraphicsTest	1.0	1.0	1	0	1	2	113	56.5
HighlightArea	0.1	0.1	1	0	10	5	81	16.2
HighlightFilter	0.0	0.0	0	0	0	1	127	127.0
HrefButtonArea	0.5833333	0.33333334	12	9	36	9	203	22.555555
ImageMap	0.7509881	0.3320158	84	106	253	23	784	34.086956
ImageMapArea	0.123333335	0.07666667	23	14	300	25	215	8.6
LinkArea	0.3	0.3	3	0	10	5	53	10.6
Matrix3D	1.0	1.0	45	0	45	10	1063	106.3
Model3D	1.0	0.8	12	3	15	6	471	78.5
MouseTrack	0.47619048	0.33333334	7	3	21	7	252	36.0
NameArea	0.33333334	0.33333334	1	0	3	3	13	4.3333335
NervousText	0.46666667	0.46666667	7	0	15	6	146	24.333334
Node	0.0	0.0	0	0	0	0	0	0.0
OvalShape	0.0	0.0	0	0	1	2	14	7.0
ParseException	0.0	0.0	0	0	0	0	0	0.0
PolygonShape	1.0	1.0	3	0	3	3	80	26.666666
QSortAlgorithm	0.0	0.0	0	0	1	2	84	42.0
RectShape	0.0	0.0	0	0	1	2	14	7.0
RoundButtonFilter	1.0	1.0	3	0	3	3	380	126.666664
RoundHrefButtonArea	1.0	1.0	1	0	1	2	69	34.5
RoundRectShape	0.0	0.0	0	0	1	2	18	9.0
ShapeTest	0.0	0.0	0	0	0	1	131	131.0
SortAlgorithm	0.71428573	0.61904764	13	2	21	7	61	8.714286
SoundArea	1.0	0.8333333	5	1	6	4	94	23.5
ThreeD	0.22727273	0.121212125	8	7	66	12	361	30.083334
TicTacToe	0.22727273	0.16666667	11	4	66	12	450	37.5
TickerArea	1.0	0.8333333	5	1	6	4	198	49.5

Table 6.1: Cohesion Results for JDK version 1.1 Demo Suite

Class name	LCC	TCC	direct links	indirect links	links	number of visible instance methods	instance size (byte codes)	average method size (byte codes)
Allocator	0.0	0.0	0	0	0	0	0	0.0
Applying	0.0	0.0	0	0	0	0	0	0.0
Array	0.4680851	0.46187943	521	7	1128	48	1033	21.520834
ArrayIterator	0.8947368	0.8479532	145	8	171	19	150	7.894737
BinaryCompose	0.0	0.0	0	0	0	1	12	12.0
BinaryComposePredicate	0.0	0.0	0	0	0	1	12	12.0
BinaryNot	0.0	0.0	0	0	0	1	10	10.0
BinaryPredicateFunction	0.0	0.0	0	0	0	1	10	10.0
BindFirst	0.0	0.0	0	0	0	1	7	7.0
BindFirstPredicate	0.0	0.0	0	0	0	1	7	7.0
BindSecond	0.0	0.0	0	0	0	1	7	7.0
BindSecondPredicate	0.0	0.0	0	0	0	1	7	7.0
BooleanArray	0.46666667	0.46666667	21	0	45	10	53	5.3
BooleanIterator	0.79532164	0.79532164	136	0	171	19	164	8.631579
ByteArray	0.46666667	0.46666667	21	0	45	10	54	5.4
ByteIterator	0.79532164	0.79532164	136	0	171	19	165	8.684211
CharArray	0.46666667	0.46666667	21	0	45	10	53	5.3
CharIterator	0.79532164	0.79532164	136	0	171	19	163	8.578947
Comparing	0.0	0.0	0	0	0	0	0	0.0
Copying	0.0	0.0	0	0	0	0	0	0.0
Counting	0.0	0.0	0	0	0	0	0	0.0
DList	0.26363635	0.2051948	316	90	1540	56	1060	18.928572
DListIterator	0.6862745	0.62091506	95	10	153	18	133	7.388889
DListNode	0.0	0.0	0	0	0	0	0	0.0
Deque	0.62896407	0.56131077	531	64	946	44	1622	36.863636
DequeIterator	0.9	0.8578947	163	8	190	20	315	15.75
DividesInteger	0.0	0.0	0	0	0	1	11	11.0
DoubleArray	0.46666667	0.46666667	21	0	45	10	53	5.3
DoubleIterator	0.79532164	0.79532164	136	0	171	19	163	8.578947
EqualTo	0.0	0.0	0	0	0	1	4	4.0
Filling	0.0	0.0	0	0	0	0	0	0.0
Filtering	0.0	0.0	0	0	0	0	0	0.0
Finding	0.0	0.0	0	0	0	0	0	0.0
FloatArray	0.46666667	0.46666667	21	0	45	10	53	5.3
FloatIterator	0.79532164	0.79532164	136	0	171	19	163	8.578947
GreaterEqualInteger	0.0	0.0	0	0	0	1	11	11.0
GreaterEqualString	0.0	0.0	0	0	0	1	10	10.0
GreaterInteger	0.0	0.0	0	0	0	1	11	11.0
GreaterString	0.0	0.0	0	0	0	1	10	10.0
Hash	0.0	0.0	0	0	0	1	10	10.0
HashComparator	0.0	0.0	0	0	0	1	9	9.0
HashMap	0.3598485	0.23484848	124	66	528	33	1057	32.030304
HashMapIterator	0.875	0.76666665	92	13	120	16	234	14.625
HashMapNode	0.0	0.0	0	0	0	0	0	0.0
HashSet	0.53475934	0.41354725	232	68	561	34	1099	32.32353
HashSetIterator	0.84615386	0.71794873	56	10	78	13	162	12.461538
HashSetNode	0.0	0.0	0	0	0	0	0	0.0
Hashing	0.0	0.0	0	0	0	0	0	0.0
Heap	0.0	0.0	0	0	0	0	0	0.0
IdenticalTo	0.0	0.0	0	0	0	1	7	7.0
InsertIterator	0.0	0.0	0	0	6	4	13	3.25
InsertResult	0.0	0.0	0	0	0	0	0	0.0
IntArray	0.46666667	0.46666667	21	0	45	10	53	5.3
IntIterator	0.79532164	0.79532164	136	0	171	19	163	8.578947
InvalidOperationException	0.0	0.0	0	0	0	0	0	0.0
LengthString	0.0	0.0	0	0	0	1	7	7.0
LessEqualInteger	0.0	0.0	0	0	0	1	11	11.0
LessEqualString	0.0	0.0	0	0	0	1	10	10.0
LessInteger	0.0	0.0	0	0	0	1	11	11.0
LessString	0.0	0.0	0	0	0	1	10	10.0
LogicalAnd	0.0	0.0	0	0	0	1	12	12.0
LogicalNot	0.0	0.0	0	0	0	1	8	8.0
LogicalOr	0.0	0.0	0	0	0	1	12	12.0
LongArray	0.46666667	0.46666667	21	0	45	10	53	5.3
LongIterator	0.79532164	0.79532164	136	0	171	19	163	8.578947
MinMax	0.0	0.0	0	0	0	0	0	0.0
MinusInteger	0.0	0.0	0	0	0	1	11	11.0
ModulusInteger	0.0	0.0	0	0	0	1	11	11.0

Table 6.2: Cohesion Results for JGL version 1.1 Class Library, 1 of 2

Class name	LCC	TCC	direct links	indirect links	links	number of visible instance methods	instance size (byte codes)	average method size (byte codes)
NegateInteger	0.0	0.0	0	0	0	1	8	8.0
NegativeInteger	0.0	0.0	0	0	0	1	8	8.0
NotEqualTo	0.0	0.0	0	0	0	1	8	8.0
NotIdenticalTo	0.0	0.0	0	0	0	1	7	7.0
ObjectArray	0.46666667	0.46666667	21	0	45	10	48	4.8
ObjectIterator	0.79532164	0.79532164	136	0	171	19	150	7.894737
OrderedMap	0.6737968	0.6737968	378	0	561	34	365	10.735294
OrderedMapIterator	0.80526316	0.80526316	153	0	190	20	218	10.9
OrderedSet	0.68235296	0.68235296	406	0	595	35	447	12.771428
OrderedSetIterator	0.77205884	0.77205884	105	0	136	17	156	9.176471
OrderedSetOperations	0.0	0.0	0	0	0	0	0	0.0
OutputStreamIterator	0.0	0.0	0	0	6	4	47	11.75
Pair	1.0	1.0	3	0	3	3	44	14.666667
Permuting	0.0	0.0	0	0	0	0	0	0.0
PlusInteger	0.0	0.0	0	0	0	1	11	11.0
PlusString	0.0	0.0	0	0	0	1	11	11.0
PositiveInteger	0.0	0.0	0	0	0	1	8	8.0
Print	0.0	0.0	0	0	0	1	6	6.0
Printing	0.0	0.0	0	0	0	0	0	0.0
Randomizer	0.0	0.0	0	0	0	0	0	0.0
Range	1.0	1.0	3	0	3	3	44	14.666667
Removing	0.0	0.0	0	0	0	0	0	0.0
Replacing	0.0	0.0	0	0	0	0	0	0.0
ReverseIterator	0.77205884	0.77205884	105	0	136	17	112	6.5882354
Reversing	0.0	0.0	0	0	0	0	0	0.0
Rotating	0.0	0.0	0	0	0	0	0	0.0
SList	0.17679945	0.1516422	217	36	1431	54	987	18.277779
SListIterator	0.85714287	0.74725276	68	10	91	14	105	7.5
SListNode	0.0	0.0	0	0	0	0	0	0.0
SelectFirst	0.0	0.0	0	0	0	1	4	4.0
SelectSecond	0.0	0.0	0	0	0	1	4	4.0
SetOperations	0.0	0.0	0	0	0	0	0	0.0
ShortArray	0.46666667	0.46666667	21	0	45	10	54	5.4
ShortIterator	0.79532164	0.79532164	136	0	171	19	165	8.684211
Shuffling	0.0	0.0	0	0	0	0	0	0.0
Sorting	1.0	1.0	6	0	6	4	270	67.5
Swapping	0.0	0.0	0	0	0	0	0	0.0
TimesInteger	0.0	0.0	0	0	0	1	11	11.0
ToString	0.0	0.0	0	0	0	1	7	7.0
Transforming	0.0	0.0	0	0	0	0	0	0.0
Tree	0.61742425	0.58522725	309	17	528	33	1474	44.666668
TreeNode	0.0	0.0	0	0	0	0	0	0.0
UnaryCompose	0.0	0.0	0	0	0	1	8	8.0
UnaryComposePredicate	0.0	0.0	0	0	0	1	8	8.0
UnaryNot	0.0	0.0	0	0	0	1	9	9.0
UnaryPredicateFunction	0.0	0.0	0	0	0	1	9	9.0
VectorArray	0.78431374	0.78431374	120	0	153	18	145	8.055555
VectorIterator	0.79532164	0.79532164	136	0	171	19	150	7.894737

Table 6.3: Cohesion Results for JGL version 1.1 Class Library, 2 of 2

Class name	LCC	Super class name	LCC
AniArea	0.50	ImageMapArea	0.12
ClickArea	1.00	ImageMapArea	0.12
DelayedSoundArea	1.00	ImageMapArea	0.12
HighlightArea	0.10	ImageMapArea	0.12
HrefButtonArea	0.58	ImageMapArea	0.12
LinkArea	0.30	ImageMapArea	0.12
NameArea	0.33	ImageMapArea	0.12
SoundArea	1.00	ImageMapArea	0.12
TickerArea	1.00	ImageMapArea	0.12
RoundButtonFilter	1.00	ButtonFilter	1.00
RoundHrefButtonArea	1.00	HrefButtonArea	0.58
BidirBubbleSortAlgorithm	0.00	SortAlgorithm	0.71
BubbleSortAlgorithm	0.00	SortAlgorithm	0.71
QSortAlgorithm	0.00	SortAlgorithm	0.71

Table 6.4: Cohesion of Classes Involved in Private Reuse, JDK Demo Suite

Chapter 7

Conclusions and Discussion

7.1 Results from the Research

7.1.1 Formal Validation of the Bieman-Kang Cohesion Measures

The research attempted to confirm the representation condition for the Bieman-Kang cohesion measures. This was done by observing a number of properties in the empirical relation system of each measure, and proving their validity in the corresponding numerical relation system. An analytical tool called the Method Interconnection Graph (MIG) was developed to aid in this task. Finally, the ratio scale type of the cohesion measures was demonstrated.

Two anomalies of the measures were observed which did not contradict the empirical relation systems established for the measures.

- There exist cases where one class is less cohesive than another by LCC, but the converse is true by TCC.
- The measures cannot distinguish between graphs which have nodes that are easier to remove than others, but otherwise have equal cohesion measures. A result was presented which says that in general it is not possible to distinguish these cases with a cohesion measure based only on graph edges.

A more serious issue concerns the ability of the loose cohesion measure to recognize factorability of classes. There exist many cases where a class of more factors has higher LCC than one with fewer factors. This situation runs counter to the empirical notion that a class with more factors is more splitable, and therefore *less* cohesive.

To better analyze the population of class factors, an algorithm was presented which generates the distinct partitions of a graph with N nodes into K factors. Loose cohesion (LCC) was computed for each of these partitions for a certain N. A large amount of overlap was observed in the ranges of LCC values with respect to the factor count of the partition. The contradiction of factorability diminishes the usefulness of LCC in predicting factorability. However, the fractal-like patterns of the plots strongly

suggest a deeper relationship between LCC and factorability. Moreover, the population distribution of the cohesion measures tracks closely with that of factorability.

7.1.2 Definition of a Java Object Model Suitable for Measurement

An object model appropriate for measurement of Java programs was proposed.

- Static fields, static methods, constructors, finalizers, and initializers are omitted from the Java model.
- Only local cohesion is considered. Cohesion through inheritance is problematic in Java since all objects subclass the top level object `Object`.
- An object composed within another is considered to be an instance variable use rather than call for the purposes of measurement.
- Helper classes and inner classes are considered to be outside class boundaries.
- Overloaded methods are considered distinct.
- Only the declared type of the call site is considered in determining the method call relation under dynamic method dispatch.

7.1.3 *Celebes*, a General-Purpose Java Measurement Tool

A general-purpose Java static analysis and measurement tool called *Celebes* was produced. *Celebes* parses Java source code and byte code files. From the source code it draws names and overall program structure. From the byte codes are derived method call and field use information. A structure tree is used to store and propagate this information throughout the class representation so that measurement can take place.

Some advantages to using byte code for static analysis are:

- Leverage the compiler to determine the type signature of a method calling protocol.
- Use the compiler to resolve self and superclass references.

7.1.4 Empirical Investigation into the Relationship between Class Cohesion and Class Size

Two publicly available Java software packages were measured. Both are of medium size (43 usable classes). One is a collection of Java applets. The other is a reusable class library. Neither exhibited any relationship between cohesion and size as measured in number of byte code statements. This conclusion confirms the hypothesis that class cohesion and class size are independent. Clearly, class cohesion is not just another size measure.

Some observations were made about the cohesion measurement task. First, there appear to be a large proportion of static methods and fields in real applications. Since the research model does not consider them, cohesion measurements were not possible in nearly half the classes considered.

Second, cohesion measures tend to be greatly lowered by the presence of vacuous methods (methods with no body). Only one class was found exhibiting such methods. However, the sensitivity of the cohesion measures to the presence of vacuous methods strongly suggests that they be excluded from the Java object model.

Finally, a few data were obtained regarding the cohesion of those classes most reused in the data set. Confirming earlier work on the subject, the most reused classes exhibited lower cohesion.

7.2 Discussion

7.2.1 What Are Cohesion Measures Supposed to Do?

The Bieman-Kang cohesion measures seem best suited to conveying information about the completeness of method connectivity in a class: the more connectivity the higher the measure. The empirical relation system of Hitz and Montazeri, by contrast, relates the ease of separating a method in a class with the removal of its connections to other methods. It does not relate the completeness of connectivity because a class can have many connections and still be considered minimally cohesive.

The Hitz and Montazeri measure captures a notion of connection strength: the more connections that bind the methods in a class, the higher its cohesion. This relation is also present in the Bieman-Kang measures.

Neither empirical relation system completely captures the degree to which a class can be decomposed without changing any code (factorability).

The question remains, "Why are we doing this?" Fenton speaks of the need to apply Basili's Goal-Question-Metric (GQM) paradigm prior to measurement definition [Fenton94]. Clearly the term *class cohesion* means different and possibly conflicting things. One task for the research community is to discern these differences and define more directed measures. For example, if the goal is to identify coincidental cohesion (presence of unrelated components in a class), then the measure's empirical relation system must include a notion of factorability. If, on the other hand, the goal is to identify components which can be removed from a class with minimal changes to the class, then the measure's empirical relation system should reflect the notion of cost of breaking connections. If the goal is to evaluate the potential of connectivity for prediction of external attributes like reusability, then the measure's empirical relation system needs to consider the degree of completeness of the possible method connections in the class. Assessing the attribute in terms of a more limited set of goals might diminish its utility. But it would be a better measure.

The general problem of quantizing structures of graphs deserves closer attention by cohesion researchers. For instance, the Integer Partition Problem was found to be most useful in analyzing the problem of factorability. However, the literature on cohesion is singularly lacking in application of this and other problems from combinatorics. Workers in software engineering should do more to tap into this rich

subject.

A special challenge presents itself in the measurement of factorability. The enumeration of the set of distinct partitions on a graph is a useful tool for analyzing the requirements of such a measure. This tool suggests an approach for a strict total ordering of classes according to their method interconnection graphs. But this defines a new measure which must undergo the same scrutiny as the one currently under study.

7.2.2 The Importance of Experimentation

Many assertions have been made over the years regarding the importance of having high cohesion in software modules if external properties such as quality, reliability, and maintainability are to be assured. Few experiments have been run to verify even the simplest of relationships; for example, those postulated between cohesion and size, and between cohesion and reuse. This research presented some results along these lines which are counter-intuitive. Admittedly, the experiments performed by this research suffered from certain flaws, most notably the lack of gaussian distribution of the independent variable Class Size. Software experimentation is certainly difficult, but all too often results are accepted as mantra without proper empirical work done to confirm them. This research took some care in designing repeatable experiments performed on publicly available software of non-trivial size. The experimental model should be reusable.

7.2.3 Cohesion and the Software Designer

Does high class cohesion imply better object-oriented software design? Class cohesion measures are in their infancy, but we may begin to think about designing suitable experiments. The next step for this research would be to find a data set in which reuse is high. Then a proper experiment could be set up to test the conventional wisdom that highly cohesive components (those with a single function) are the most reused.

The advent of automatic code generation by integrated development environments (IDEs) and pattern writing of code from recipe books [Gamma95] has caused classes to acquire more coupling, subclass deeper inheritance hierarchies, and implement more varied interfaces than would be manageable with hand-written code. It would be interesting to see whether cohesion is higher in these cases (the hypothesis being that classes are much more specific in their purpose).

One result of cohesion research in general is that the concept of cohesion is not a simple one. Cohesion measures need to be highly goal-directed. It is unlikely a single scalar measure can capture all the attributes we wish to distinguish under the rubric of cohesion.

7.3 Future Work

Here is a summary of future work that would be of interest.

- Reformulate the empirical relation system to eliminate the relation between factorability and

cohesion, and define a new measure which captures the notion of factorability.

- Obtain a data set in which reuse is high to investigate the relationship between reuse and cohesion.
- Obtain a data set with larger size classes to repeat the experiments relating class size and cohesion.
- Adapt the cohesion measures to selectively account for method connectivity established through static fields, static methods, and constructors.
- Adapt *Celebes* to capture connection strength information through counting of graph node in-degrees.
- Extend *Celebes* to perform source code transformations which separate coincidental class components.

Bibliography

[Arnold96] Ken Arnold, James Gosling. *The Java Programming Language* . New York: Addison Wesley, 1996.

[Baker90] A Baker, J Bieman, N Fenton, D Gustafson, A Melton, R Whitty, "A Philosophy for software Measurement," *Journal of Systems Software*, vol 12, July 1990.

[Basili96] Victor Basili, Lionel Briand, Walcelio Melo. "Validation of Object-Oriented Design Metrics as Quality Indicators." *IEEE Transactions on Software Engineering*, vol 22 no 10, October 1996.

[Bieman93] James Bieman and Byung-Kyoo Kang. "Measuring Functional Cohesion." *Computer Science Technical Report CS-93-109*. Colorado State University, 1993.

[Bieman94] James Bieman and Linda Ott. "Measuring Functional Cohesion." *IEEE Transactions on Software Engineering*, vol 20 no 8, August 1994.

[Bieman95] James Bieman, Byung-Kyoo Kang. "Cohesion and Reuse in an Object-Oriented System." *Proceedings of the ACM Symposium Software Reusability (SSR'95)*, pp. 259-262, April 1995.

[Bogart83] Kenneth Bogart. *Introductory Combinatorics*. Marshfield MA: Pitman Publishing, 1983.

[Booch94] Grady Booch. *Object-Oriented Analysis and Design with Applications*, 2nd Edition. Benjamin Cummings Publishing Co., Inc. (Addison-Wesley) 1994.

[Briand94] L Briand, S Morasca, V Basili. "Defining and Validating High-Level Design Metrics," *Computer Science Technical Report CS-TR 3301*, University of Maryland at College Park, 1994.

[Briand96] Lionel Briand, Sandro Moasca, Victor Basili. "Property-Based Software Engineering Measurement." *IEEE Transactions on Software Engineering*, vol 22 no 1, January 1996.

[Briand97a] Lionel Briand, John Daly, Jürgen Wüst. "A Unified Framework for Cohesion Measurement in Object-Oriented Systems." *Technical Report ISERN-97-05*. Kaiserlautern: Germany: Fraunhofer Institute for Experimental Software Engineering, 1997. Also available at `http://www.iese.fhg.de/ISERN/pub/isern.biblio.html`.

[Briand97b] Lionel Briand, John Daly, Jürgen Wüst. "A Unified Framework for Cohesion Measurement in Object-Oriented Systems." *Proceedings of the International Software Metrics Symposium* (upcoming).

[Budd91] Timothy Budd, *An Introduction to Object Oriented Programming* New York: Addison-Wesley Publishing, 1991.

[Chambers96] Craig Chambers, Jeffrey Dean, David Grove. "Whole-Program Optimization of Object-Oriented Languages." *Technical Report 96-06-02*, University of Washington, June 1996.

[Chidamber94] Shyman Chidamber and Chris Kemerer. "A Metrics Suite for Object Oriented Design." *IEEE Transactions on Software Engineering*, vol 20 no 6, June 1994.

[Churcher95] Neville Churcher, Martin Shepperd. "Comments on a Metrics Suite for Object -Oriented Design," *IEEE Transactions on Software Engineering*, vol 21 no 3, March 1995.

[Dean96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, Craig Chambers. "Vortex: An Optimizing Compiler for Object-Oriented Languages." *Proceedings of OOPSLA'96*, San Jose, CA, October, 1996.

[Fenton91] Norman Fenton. *Software Metrics, A Rigorous Approach.* London: Chapman and Hall, 1991.

[Fenton94] Norman Fenton. "Software Measurement: A Necessary Basis." *IEEE Transactions on Software Engineering*, vol 20 no 3, March 1994.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns.* Reading MA: Addison-Wesley, 1995.

[Graybill94] Franklin Graybill and Hariharan Iyer. *Regression Analysis.* Belmont CA: Duxbury Press, 1994.

[Gosling96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* New York: Addison Wesley, 1996.

[Hall86] Marshall Hall Jr. *Combinatorial Theory, Second Edition.* New York: John Wiley, 1986.

[Hitz95] Martin Hitz, Behzad Montazeri. "Measuring Coupling and Cohesion in Object Oriented Systems." *Proceedings of the International Symposium on Applied Corporate Computing, Oct 25 - 27, 1995.* Monterrey Mexico, 1995

[Hitz96] Martin Hitz, Behzad Montazeri. "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective." *IEEE Transactions on Software Engineering*, vol 22 no 4, April 1996.

[Kruglinksi96] David Kruglinksi, *Inside Visual C++.* Redmond WA: Microsoft Press, 1996.

[Javacc97] "Javacc, The Java Compiler Compiler." version 0.6.
`http://www.suntest.com/JavaCC/`
Sun Microsystems Inc, 19 Feb 1997.

[Javasoft97] "Inner Classes Specification."
`http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html`
Sun Microsystems, Inc. 4 Feb, 1997.

[Kang96a] Byung-Kyoo Kang, James Bieman. "Design-level Cohesion Measures: Derivation, Comparison, and Applications." *Computer Science Technical Report CS-96-104*. Colorado State University, 1996.

[Kang96b] Byung-Kyoo Kang, James Bieman. "Using Design Cohesion to Visualize, Quantify, and Restructure Software." *Computer Science Technical Report CS-96-103*. Colorado State University, 1996.

[Kitchenham95] Barbara Kitchenham, Shari Lawrence Pfleeger, Norman Fenton. "Towards a Framework for Software Measurement Validation," *IEEE Transactions on Software Engineering*, vol 21 no 12, December 1995.

[Lindholm96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. New York: Addison Wesley, 1996.

[Melton90] A Melton, D Gustafson, J Bieman, A Baker, "Mathematical Perspective of Software Measures Research," *IEE Software Engineering Journal*, vol 5 no 5, 1990.

[ObjectSpace97] "Java Generic Library (JGL version 1.1.
`http://www.objectspace.com/jgl/`
Object Space Inc. Jan 1997.

[Ott95] Linda Ott, James Bieman, Byung-Kyoo Kang, Bindu Mehra. "Developing Measures of Class Cohesion for Object-Oriented Software." *Proceedings of the Annual Oregon Workshop on Software Metrics (AOWSM95)*, 1995.

[Pohl93] Ira Pohl. *C++ for C Programmers*, Second Edition. Redwood City, CA: Benjamin Cummings, 1993.

[Ross85] Kenneth Ross, Charles Wright. *Discrete Mathematics*. Englewood Cliffs NJ: Prentice Hall, 1985.

[Rumbaugh91] James Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Stevens74] W Stevens, G Myers, L Constantine. "Structured Design," *IBM Systems Journal*, vol 13, pp 115-139, 1974.

[Yourdon79] E Yourdon and L Constantine. *Structured Design. Fundamentals of a Discipline of Computer Program and System Design.* Englewood Cliffs, NJ:Prentice Hall, 1979.

[Zuse91] Horst Zuse. *Software Complexity, Measures and Methods* New York: Walter de Gruyter, 1991.