INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa • University of Ottawa

Heuristic Program Reorganization Guided by Object-Oriented Metrics

Mark Abraham Israel

Thesis

submitted to the School of Graduate Studies and Research

in partial fulfillment of the requirements for the degree

Master of Computer Science

The Ottawa-Carleton Institute for Computer Science

University of Ottawa

© Mark Israel, 14 April 1997

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced with the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-20976-8

Canada

Abstract

The benefits of increased reusability and maintainability afforded by object-oriented programming are now widely recognised, but there is a vast legacy of procedural code that it may be more desirable to attempt to transform into object-oriented form than to redesign from scratch. A system is here presented that will assist a human programmer in performing such a transformation, by identifying potential objects in C code: the system produces a clustering of the procedures and global variables in the C code, which are to be considered as the methods and the data members respectively of a C++ object. Three different clustering techniques are compared: AGGLOM by Fisher, a bottom-up clustering technique by Basili, and a novel matrix diagonalization technique for simultaneously clustering entities and their attributes. A suite of object-oriented metrics proposed by Kemerer and experimentally found to correlate to maintainability is used to guide the clustering; metrics scores for a particular clustering are compared with scores for randomly constructed objects of the same size. The resulting objects, which are customizable according to which metric the user favours, are also compared against the objects produced when humans transform the C code to C++. The tool constructed is deemed to find useful candidate objects in several C test programs, and some novel refinements of the object-oriented metrics are demonstrated by the experimental results to be improvements.

Résumé

Les avantages de la réutilisabilité et de la maintenabilité des programmes orientés objet sont généralement reconnus. Il y a une grande quantité de programmes procéduraux qu'il serait peut-être préférable de transformer en programmes orientés objet que de réécrire complètement. Le système présenté ici aide le programmeur à effectuer cette transformation en détectant des objets potentiels dans du code en langage C. Le système produit un groupage des procédures et des variables globales du code C, qui doivent être considérées respectivement comme les méthodes et les membres d'un objet C++. Trois techniques de regroupement sont comparées: AGGLOM de Fisher, une technique ascendante de Basili, et une technique originale de diagonalisation de matrice permettant de grouper simultanément les entités et leurs attributs. Le regroupement est guidé par une série de métriques orientées objet proposées par Kemerer, et corrélées à la maintenabilité, comme cela a été démonstré expérimentalement. Les scores d'un groupage sont comparés avec les scores d'objets de même taille construits au hasard. Les objets résultants, qui sont adaptables suivant la métrique préférée de l'utilisateur, sont aussi comparés avec les objets produits lors d'une transformation manuelle du code C en C++. Cet outil est capable de découvrir utilement des candidats au statut d'objets dans plusieurs programmes de test en C, et certains raffinements de la métrique orientée objet sont effectivement des améliorations, comme le montrent les résultats expérimentaux.

Acknowledgements

I am grateful to my supervisor, Professor Robert Holte, and to his colleague, Professor Stan Matwin, for welcoming me to the Machine Learning research group at the University of Ottawa, and for their guidance and support throughout this research. Most of the research was conducted under a research assistantship funded from an NSERC Strategic Grant titled "Machine Learning Applied to Software Reuse".

I also thank my parents for their encouragement and support.

Chapter 1: Introduction....................................6
1.1 Problem Statement.......................................6
1.2 The Search for Objects..................................6
1.3 Legitimacy and the Schools of OO........................8
1.4 The Use of OO Metrics...................................9
1.5 Notes on Terminology....................................10
Chapter 2: Survey of Approaches to Object-Oriented Re-engineering of Procedural Code..............................12
2.1 Introduction..12
2.2 Automated and Semi-automated Approaches.................12
2.3 Manual Approaches.......................................17
2.4 Summary...23
Chapter 3: Survey of Object-Oriented Software Metrics.....24
3.1 Definition of "Software Metric".........................24
3.2 Traditional Software Metrics............................25
3.3 Limitations of Traditional Metrics for OO...............26
3.4 The MIT OO Metrics Suite................................27
3.5 Other OO metrics..32
Chapter 4: Clustering Techniques..........................35
4.1 Problem Statement.......................................35
4.2 The Globals Based Object Finder.........................36
4.3 BASILI Clustering.......................................39
4.4 AGGLOM Clustering.......................................44
Chapter 5: Clustering in Two Dimensions Simultaneously....47
5.1 Problem Statement.......................................47
5.2 Related Work..50
5.3 The DIAG Algorithm......................................51
5.4 Testing of DIAG...53
Chapter 6: Selection and Refinement of Metrics............63
6.1 Initial Selection of Metrics............................63
6.2 Improvements to Lack of Cohesion in Methods Metric.....64
6.3 Straw Man: the Random Clusterer........................67
6.4 Normalization of Metrics Against Number of Objects.....71
6.5 A Different Kind of Metric: Pairwise Goodness.........72
Chapter 7: Design of the Experimental System..............74
Chapter 8: Experimental Results...........................78
8.1 The Test Programs.......................................78
8.1.1 ARGOT...78
8.1.2 DIAG..78
8.1.3 RAY...79
8.1.4 RL..79
8.1.5 MMS...79
8.2 Results and Analysis....................................80
8.2.1 ARGOT...80
8.2.3 RAY...90
8.2.4 RL..91
8.2.5 MMS...92
Chapter 9: Conclusions....................................93
9.1 Conclusions from Experimental Results...................93
9.2 Future Work...97
9.3 Relation to Machine Learning............................98
9.4 The Last Word...100
References..101

Chapter 1
Introduction

1.1 Problem Statement

The object-oriented (OO) paradigm affords dramatic improvement on earlier programming paradigms in making software more reusable and more maintainable. In order to gain these advantages, software developers often expend great effort in migrating older programs (coded in a "procedural" paradigm) to OO form. The purpose of this research is to design a tool that will assist in the task of such migrations by identifying candidate objects in procedural code.

Our practical experimentation will be with procedural programs written in the C language. C was chosen because of the vast legacy of procedural code that exists in it, and because of the availability of the language C++, which is similar to C in nearly all respects except for the addition of OO facilities. However, the principles underlying the research are in no way specific to C and the results should be applicable to any choice of procedural and OO languages.

1.2 The Search for Objects

The key step in the design of an OO program is the creation of objects and classes. One of the major characteristics of OO is inheritance, and in a finished OO program the objects will be organized in an inheritance hierarchy. In our current research, however, we shall not attempt to construct the hierarchy, but merely to identify the objects. Once the objects have

been identified, either another tool or a human programmer could be used to seek common features among the objects and determine the inheritance hierarchy from those features.

An object consists of data members and of methods. If a procedural program and an OO program are algorithmically equivalent, then there must exist in the procedural program entities that provide the functionality of an object's components. Since methods consist of executable code and since all code in a procedural program must lie within some procedure (we regard the main program segment as a procedure), the methods must correspond to either procedures or parts of procedures. Although it is desirable to "slice" procedures to find methods, our current work looks only for methods corresponding to whole procedures. Since the data members of an object are typically used by more than one method and since procedural programming languages do not generally provide encapsulation at the same level as objects, the components that will become data members will likely be either global variables or parameters passed to the procedures. In our current work we consider only global variables.

Thus we view an object as a cluster of variables and procedures, and try to construct the object using clustering techniques. We cluster — or, more generally, re-organize — the components of the procedural program that will make up the objects in the OO program. A closely related task is that of trying to "normalize" a piece of code, i.e., to reduce it to some sort of "normal form". The idea is that there are optimal points in "code space", to each of which many suboptimal points get mapped. For reasons that we shall examine later, the clustering problem that we face is a somewhat unusual one, and we shall test a variety of clustering techniques, in part to see which is most applicable to our problem — to find the best "operators" to move us through code space.

There are three difficulties in searching this space: the space is vast; each node has a high branching factor (any component may be added to any cluster); and whatever scoring function is available is not likely to be "nice" enough for standard heuristics such as hill-

climbing to work well; for example, it is likely not to be continuous or monotonic, and to have many local maxima.

1.3 Legitimacy and the Schools of OO

Before one embarks on the quest for a tool like this, the first question to be asked is: is such a tool possible? As the OO paradigm is generally taught, OO coding must be preceded by OO analysis and OO design, and some OO partisans will argue that because OO analysis and OO design presumably did not occur before the creation of a procedural program. it is useless to seek objects in it.

There is certainly *some* degree of truth in this position. The whole claim that OO programming enhances reusability and reduces maintenance hinges on the fact that objects are meaningful units. In other words, if one does not create meaningful objects, then one will not gain the benefits of having an OO program.

Where one stands on this question is likely to depend on what OO "school" one subscribes to. [Kristensen 91] notes a difference between the "Scandinavian" and the "American" schools of OO: the Scandinavian school values OO inheritance chiefly for modelling of conceptual hierarchies, while the American school emphasizes code-sharing as the major benefit. [Yeh 95], clearly of the "American" school, defines "object" in a way that is removed from real-world modelling and can be seen as justifying our own research: "An object is an entity which has some persistent state (only directly accessible to that entity) and a behavior that is governed by that state and by the messages that object receives." As we shall see in Chapter 2, many researchers have concluded that a tool of the type we are trying to build is both possible and useful, and further that even procedural programmers untrained in the OO paradigm do often organize parts of their program along lines that can be considered OO.

For these reasons, and because a tool that identifies even *some* candidate objects in a legacy program may save a human programmer some time, we believe that it is worth while to try to build such a tool.

1.4 The Use of OO Metrics

As the quality indicator for the clusters that it builds, our tool uses OO software metrics. We shall be surveying OO metrics in Chapter 3. To the best of our knowledge, OO metrics have never yet been used for automatic or semi-automatic code improvement, and the idea of trying to transform a program so as to optimize some metric is the unique contribution of this thesis.

Four benefits to doing this sort of optimization were anticipated. First of all, if the metrics are good reflectors of code quality, then the optimization process should genuinely improve the code; in particular, it should make it more reusable. (It could also be used to optimize the quality of software, independent of any change in programming language or paradigm.) Secondly, if we optimize on the metric value and find (by some subjective or other independent measure) that the code is no more reusable than it was, then we shall have shown at least that current OO metrics are not useful for guiding clustering, and possibly that they are poor indicators of reusablity or other positive OO qualities. Thirdly, if the transformation tends to bring different (procedural or utterly chaotic) organizations of the components into a similar OO form, then this is a useful first step towards "normalizing" programs in order to compare them — this might be useful, for example, in situations where two different applications are to be merged into one. Fourthly, this sort of search is a novel application for machine learning.

Because software metrics are always imperfect, and particularly because one of the metrics that we shall use makes use of the one of the same features that we shall be clustering

on (use of shared variables), we expect our tool at times to outperform real-world OO programs on metric values. Hence we debated whether the tool should attempt to score as well as possible on the metric values, or should try to achieve proximity to metric scores for OO programs of acknowledged quality. The decision was to score as well as possible: manual translation is used as a target for "meaningfulness" (the subjective measure), but not for the metric values. Ideally, the metrics would perfectly reflect true meaningfulness, with the best possible score being the most meaningful cluster. If we find that there are meaningless clusters that score very well according to the metrics, then this will be an important result, because it will show that the metrics are not filling their intended role, and should be supplemented or repaired.

1.5 Notes on Terminology

In this thesis, "OO" stands for "object-oriented" or for "object-orientation".

Because the building of an OO inheritance hierarchy was excluded from this research, the organization of program elements that we are attempting to produce might often more strictly be called "object-based" than "object-oriented" (see [Cardelli 85]). However, we use the term "OO" without regard for this distinction.

We sometimes use the term "object" without specifying whether it signifies an object class or an object instance. [Liu 90] notes: "As a practical matter, in some cases it may be easier to first find the class and then its instances and in other cases to reverse this procedure. Thus it is more convenient to treat the two together."

We use the term "procedural" in contrast to "object-oriented", to describe such traditional programming languages as FORTRAN, COBOL, and C, and the methodologies associated with them. Often, "procedural language" is used as an antonym for "declarative language"; in this stricter sense, OO languages such as Smalltalk and C++ *are* procedural: they merely package procedural code in a new way (see [Taylor 93]). Since we are not concerned with declarative languages here, confusion will not arise.

There is much variation in what the data members of an object are called. In the literature on OO design they are often called "attributes". In Smalltalk terminology, one speaks of "instance variables", which have different values for every instance of a class, and "class variables", which have the same value for all members. C++ programmers often refer to the data members simply as "members" (adopting the meaning from the members of a `struct` in C), although, strictly speaking, the methods are members too. In this thesis, we shall generally use the term "members", but we shall speak of "data members" when greater precision is required.

Chapter 2

Survey of Approaches to Object-Oriented Re-engineering of Procedural Code

2.1 Introduction

[Arnold 93] says: "Of great interest to the reengineering community is extracting object-oriented classes and object-instances from non-object-oriented source code." In this chapter, we summarize the approaches that have been taken to this problem.

2.2 Automated and Semi-automated Approaches

[Reiss 91] divides the types of useful information that tools can provide for OO redesign into four categories: information needed to find potential classes and their interaction; information that assists understanding the original design; answers to specific questions that might arise during redesign, such as where variables are used or functions called; and answers to what-if questions such as what functions would have to be changed if the programmer combined two structures into a class.

[Liu 90], noting that programmers had been using some object-like features such as abstract data types and inheritance prior to the adoption of OO, proposes two algorithms to identify candidate objects in procedural languages. The first, "Globals Based Object Finder", groups procedures that share any global variables, and those global variables, into a candidate object. Global variables are used because they share with object members the property of being persistent data. The second, "Types Based Object Finder", groups procedures whose formal parameters share data types, or their subtypes, into an object. A data type is considered to be a subtype of another if the first type is used to define the second. The intent is that Types Based Object Finder will statically group together procedures that are dynamically passed the same data, even if no global variables are used. The authors admit that both these algorithms produce objects that are in many cases "too big". They propose that human intervention or heuristic search be used to refine the candidate objects by excluding irrelevant global variables and types from consideration.

[Lieberherr 90], [Lieberherr 91] investigated how to organize objects into an optimal inheritance hierarchy once they had been defined. Objects having members with the same name are grouped together. The transformation from an arbitrary inheritance graph to an optimal graph is in two steps: one to minimize the total number of edges in the graph, and the other to minimize multiple inheritance. The problem is shown to be NP-hard, and "an efficient approximation algorithm" is offered.

[Lano 92] describes a tool that attempts to assist reverse-engineering of COBOL programs. One object is created for each file opened by the COBOL program and also for each array. The code is divided into "phases", maximal contiguously executed statement sequences that do not open or close any files, or alternatively into slices using the data assigned to the object; the phases or slices are assigned to the object. The user then specifies which object he

wishes to examine, and data flow diagrams and entity life histories are produced. The user can direct the tool to merge multiple objects into one. The tool was applied to sections from a library database software package to assist restructuring them. (Although the authors describe the reverse-engineered design as object-oriented, the target language was apparently the same as the source language: COBOL with no OO extensions.) The restructured version "was determined to be much easier to understand by the user organization".

[Solnon 92a], [Solnon 92b], [Solnon 93] proposes an algorithm to infer data type inheritance hierarchies from Prolog code. The variable type in the head of a clause is considered to be a specialization of the conjunction of variable types in the body of the clause; thus, given the Prolog statement "`person(X) :- woman(X).`", it is inferred that "`woman`" is a subtype of "`person`". A separate hierarchy is built by observing inclusion relationships in the test data. The two hierarchies are compared to find inconsistencies.

[Silva-Lepe 93] reports on manually identifying objects in a C ray-tracing program in a quest to find heuristics for object identification, for use in a future tool for semi-automated OO reverse engineering. He found that union types could become abstract classes; that data structures with self-referential pointers, traversed by loops matching the iterator pattern, could become new collection classes; that C "`switch`" statements could suggest abstract classes with deferred polymorphic methods; that sets of variables often passed together as parameters could become classes; and that arrays of the same dimension could become objects in the same class. He found that methods should be attached to those objects whose data they modified most often. He observed that since methods in OO programs are typically smaller than procedures in procedural programs, "it cannot be expected that procedural functions and object-oriented methods be in one-to-one correspondence."

[Ong 93] describes a prototype tool that extracts candidate objects from FORTRAN 77 code and translates to C++. The data members for the candidate objects are decided on first; these are selected in three ways: whole FORTRAN common blocks become objects; arrays with the same dimensions become instances of the same object class; and different variables passed as same-position parameters in different invocations of the same subroutine become instances in the same object class. Dataflow analysis is then done to segregate code into slices that reference the data assigned to a candidate object; these slices then become the object's methods. Slices where the data are only read become methods that read the state of the object; slices where the data are only set become constructor methods that create the object; and other slices become methods that modify an object. No attempt is made to build an inheritance hierarchy. The prototype tool — consisting of about 10,000 lines of C, C++, YACC, LEX, AWK, and Make — was tested on an 18,000-line FORTRAN program that simulated the spread of viral infections. 70 classes were extracted. After manual merging to eliminate redundancies (which the authors attributed to "the lack of data-abstraction capability in FORTRAN"), 20 classes remained, subsuming one third of the original FORTRAN code. The authors believed that the amount of the redundancy found was a positive indication of the reusability of the objects constructed.

[McFall 93] describes an approach to identifying candidate objects in COBOL programs. COBOL file descriptions become objects' data, and COBOL procedures become methods. A Hopfield neural network is used to optimize a function that "describes the total Euclidean distance between the data and procedures" (not further explained), thus clustering the data and procedures into candidate objects. [McFall 92] cites a feasibility study in which a neural network had used "shape analysis" (seeking distinctive patterns in command sequences) to identify known algorithms in COBOL programs, and proposes that the neural networks be used to identify common methods such as object creation, a read operation, and data validation.

[Yeh 95] wrote a tool to identify candidate classes and candidate objects in C code. The candidate classes consisted of `structs` and procedures that referenced the `structs`' internal fields; the candidate objects consisted of global and external variables and procedures that referenced them. Apparently the algorithm allowed for no inter-class or inter-object references: the authors say that it "often finds connected components that are too large to be thought of as" OO artefacts. Their remedy was an "interactive exclusion facility" where a user directed the tool to ignore certain `structs` and variables as being "part of the underlying system". The tool was applied to a 30,000-line C program. After 40 of 62 files were excluded as being "part of the underlying system", the tool found 12 candidates deemed to be "of interest", comprising 122 procedures, 99 external variables, and 7 `structs`. The authors concluded that the tool "cannot find ADTs of object instances where none were intended by the programmers."

[Sneed 95] describes a tool that allows a user to select as a candidate object any subset of the data in a COBOL program, and then generates operation trees and collaboration graphs for those objects similar to those described in the OO analysis literature. No results are reported.

[Newcomb 95] describes a tool that analyses a suite of COBOL programs and generates what the authors assert is a complete abstract OO model. The tool groups together data structures whose fields match in length; these data structure become the classes. It slices basic blocks of the code (sequences of contiguous program statements that occur between control structures) into fragments that modify or read the data of the structures; these fragments are then used as "primitive methods", which are merged by parameterizing constants that occur in them. The tool then lists the classes and graphically represents their inter-behaviour. Applied to a suite of 65 programs totalling 168,183 lines of code, the tool found 303 classes; no attempt was made to assess the quality of the classes.

[Joiner 96] discusses how to derive OO designs from COBOL programs. Although one of the authors is the same as for [Ong 93], the technique of slicing code is here abandoned: whole COBOL paragraphs (i.e., procedures) become methods, since the authors believe they are small enough. Two algorithms are presented for clustering paragraphs: a "control-flow based algorithm" where paragraphs are clustered with either their calling paragraph or other paragraphs called by the same caller, provided those paragraphs refer to a common variable; and a "variable classification based algorithm" where a paragraph is assigned to the record variable that it references the most. The authors applied the algorithms to 10 industrial programs totalling 14,000 lines of code. The first algorithm gave 97 objects interacting with an average of 3 others; the second algorithm gave 80 objects interacting with an average of 2 others.

2.3 Manual Approaches

[Dietrich 89] describes the construction of a new, object-oriented programming interface for a pre-existing program for geometric solid modelling; the old program consisted of several hundred thousand lines of PL/1 code. He recommends communicating between the procedural and OO parts of the code with "object wrappers". An object wrapper is an interface created for a group of procedures; the wrapper will likely be written in the procedural language, but is designed to respond to OO messages. Dietrich considers it essential that there be destructor methods for storage freed on block exits in the procedural program. He concludes that one of the most difficult issues is "making the tradeoff between exposing the legacy's functions and isolating object-oriented applications from changes in the legacy."

[Duntemann 90] approves of incrementally converting procedural programs to OO, citing Borland's modifications to the TurboCalc spreadsheet program to take advantage of OO extensions added to Turbo Pascal. Duntemann's observations, reflecting an approach similar

to the approach taken in our own research, include the following: Programmers not thinking in OO terms nonetheless sometimes unwittingly create procedure libraries along OO lines, so that the procedural application has "near-objects" consisting of a data structure or family of structures and procedures that act on them; in effect, the programmers "have performed everything but encapsulation". Conversion to OO should begin with finding near-objects in the application and making them objects, followed by organizing the objects into a hierarchy. (Duntemann cautions that the hierarchy should not be designed to "accommodate the quirks" of the procedural program.) At the core of most applications is a single central object representing the work that the application does, and OO re-representation requires "drawing a line around the object", including the code that works directly with the data. The higher the coupling between components in the procedural program, the harder it will be to re-code for OO, but once libraries are re-cast in OO form, the benefits of looser coupling are immediate. Whole procedural applications can become objects in an OO system, e.g. a word-processing program can become a document object; and procedural systems can utilize OO subsystems simply by linking them in and calling their methods.

[Slabbinck 90] describes the manual conversion of an expert system written in a rule-based expert system shell ("Level 5", by Information Builders) to C++. The knowledge base was found to be structured in chunks, each treating a specific domain of knowledge in the application, that could easily be converted into OO methods.

[Jacobson 91] is the most-cited paper on re-engineering old systems to an OO architecture. Acknowledging that "a complete change of implementation technique will seldom occur for a large system", Jacobson approves of converting large procedural systems to OO gradually. He says that a non-OO system can be described by an OO model; that each object in the OO analysis, and every connection between two objects, "must be motivated by at least one primitive description element", where description elements are pieces of information from

the procedural source code or documentation which, "in the worst case, e.g. when we trust only the source code", will have a granularity on the level of what will become OO methods; that partial re-engineering should involve identifying which parts of the system will be converted to OO, mapping each object in the analysis of those parts to the procedural implementation, and then making the analysis objects serve as wrappers of the old system; that the cut between the OO and non-OO systems should be made so that only one of them is managing the state of every object instance and the other acts as a server; and that OO encapsulation will then ease moving the plane of division further and further into the old system. Based on three systems whose conversion he was involved in, he estimates that creating an OO analysis requires 5% to 10% of the total development time of the old system.

[Dunn 91] reports on building a C++ class library based on code scavenged from 4 naval communication programs totalling 285,000 lines of code; reusable components were found by manually scanning the source code for recurring processing patterns. The resulting library contained 20 classes and 133 methods. Citing another author who said "Old software is not reusable", Dunn found this statement "too general and defeatist".

[Grass 91] gives some observations based on her redesign of YACC, which had been written in C, as an OO program in C++. According to her, it may be advisable to avoid complete redesign and concentrate effort where it will be most effective: on components of the program that are either heavily used or likely candidates for reuse. The procedural programs easiest to convert to OO are those designed around critical data structures, in the spirit of Niklaus Wirth's stepwise refinement [Wirth 76]. The transition to OO begins with the discovery of objects in the old design, often in global data structures and the procedures that reference them, which she recommends locating with a browser or design analysis tool.

[Taylor 93] says that an OO designer who has discovered something of value in the problem domain can use the process of elimination to decide whether it is an object or not. Whereas values are candidates for attributes, and actions are candidates for methods, an entity that combines multiple actions and values "*must* be an object because only objects can bind these things together in a coherent unit."

[Tomic 94] advocates migrating COBOL programs to OO by modularizing parts of the old program around objects and moving them to separately compiled modules that are called from what remains of the old program. Tomic distinguishes "data-centered modularization", where small modules are formed by moving program statements to the individual data elements they operate on, from "object-centered modularization", where data elements found to be dependent on one another are clustered into a single object. Tomic notes: "requirements for an object-oriented reengineering process embody software engineering principles such as low coupling and high cohesion".

[Gall 95] describes an approach where candidate objects are identified in procedural code as either persistent data entities stored in files or structures in the program that reference the persistent entities. The candidate objects and the relations between them are used to reverse-engineer an application model which is then correlated by hand with an application model that is forward-generated by standard OO analysis of the problem domain. Applying the process to "two mid-size industrial case studies", the authors found the results "promising" in that they "could identify quite a large percentage of application-semantic objects that, in our experience, would not have been possible using fully automatic approaches."

[Graham 95] approves of evolutionary migration of a procedural application to OO while parts of the old system are still in use. Graham says that the wrappers might offer to the OO code facilities comparable to facilities offered to a user in a traditional software menu; he

says that the wrappers "tend to be of quite coarse granularity" because of "irreducibly large-grain 'objects' often present in the design of legacy systems".

[Enright 95], based on the experience of writing OO wrappers for three UNIX System V operating system primitives — RPC (Remote Procedure Call), TLI (Transport Layer Interface), and STREAMS —, asserts that wrappers allow access to such diverse OO features as "inheritance, function overloading, aggregation, genericity, encapsulation, and polymorphism", but cautions that "poor structured code that has either low cohesion properties or high coupling properties" "should not be object-oriented re-engineered using wrapping."

[Ihme 95] gives advice on OO re-engineering of real-time software. Noting that real-time programmers are often already using such OO-like features as information hiding, modularity, and hierarchy, the authors say that parts of a real-time system should be chosen for OO re-engineering: parts that are coherent, "not too tightly scheduled", and potentially most reusable. They say that candidate objects can be identified from the commonly used models called RTSA ("Real-Time Situation Awareness"); that "terminators, data stores, minispecifications and state transition diagrams" are often the most suitable candidates for reuse; that "terminators prove to be the best object candidates"; that finding other objects requires constructing scenario models from both RTSA artefacts and domain understanding; and that objects should be chosen to minimize inter-class connections as determined from an object communication model.

A number of books have been published intended to teach C++ to C programmers, and some of these address the issues of adapting existing C code. Cited here are some comments that might be construed as motivating our own work. [Ranade 92], listing the initial steps in OO analysis as consulting the users, identifying objects, identifying common attributes, and identifying common services, says: "... a class mechanism allows you to group together

variables and functions that can be performed on these variables as a single and unique type. It also allows you to localize problems quickly, through the access privileges specified within the class declaration." [Gurewich 94] acknowledges the advantage of C++'s upwards compatibility with C to be that "the existing body of good, working C programs could be salvaged when implementing applications with the new programming language", and (like [Jamsa 93]) compares C++ classes to C structures, saying that classes hold not only the data of structures but also the functions for working with them. [Dorfman 92] closely relates C++ member methods acting on member data to C functions acting on structure data, and says that the significant gain afforded by methods consists of many small advantages such as greater readability. [Murray 93] advises that programming projects using C "ease into" C++ by adopting it in three phases: taking advantages of some C++ features in a procedural paradigm; data abstraction to design and implement classes; and finally, adoption of inheritance and virtual functions. For existing C code not under active development, he advises writing C++ headers to declare the C functions rather than adapting the C code. [Pohl 93] describes ease of migration from C and the fact that one need not abandon the installed base of C code as a "crucial" advantage of C++, and says that the idea of member functions in classes is that "the functionality required by the `struct` datatype be directly included in the `struct` declaration." [Perry 92], calling data hiding "the biggest C++ advantage" over C, says: "Structures make data hiding even more important because so much data is local or global at any time". [Traister 93] says that even programmers with knowledge only of procedural programming tend to have objects unconsciously in mind when they design programs, calls the association of member functions `struct` data functions "the beginnings of an object-oriented approach", and says that whether a task lends itself to an OO approach depends on how well the task elements can be divided into separate entities.

2.4 Summary

Although we have found many researchers to be generally in philosophical agreement with us, experimental validation of their work is rare and there is certainly room for research that makes explicit use of OO metrics, as we propose to do.

Chapter 3

Survey of Object-Oriented Software Metrics

3.1 Definition of "Software Metric"

[Curtis 83] defines software metric as "a measure derived from the requirements, specification, design, code, or documentation of a computer program ... computed for the purpose of predicting some outcome of the software development process". Curtis calls this "its more limited definition"; his use of the words "measure" and "computed" implies that a metric must be an algorithmically evaluable function yielding a single numeric value for the object that it measures.

[Haaland 92] gives several ways of classifying metrics. Metrics can be classified by measurement scale, which may be nominal (where the only analysis permitted is that of group membership, e.g. a person's sex or nationality), ordinal (where observations can be ranked and tested for equality), interval (where measurements can be added and subtracted), or ratio (where observations can be divided). A metric can be subjective (affected by the person doing the measurement) or objective. It can be a product metric (measuring the finished result of a development process) or a process metric (capturing how the product was produced). A software metric can be static (measurable by analysing a program without running it), runtime

(measurable when running the program), or evolutionary (measuring how a program changes over time in the process of software development).

3.2 Traditional Software Metrics

The quest for good software metrics began in the late 1960s [Rubey 68], and was motivated by the successful use of metrics by engineers to predict product behaviour [Curtis 83].

Systematic research on software metrics began in 1972 with Halstead's development of his "Software Science" [Halstead 77]. Halstead used counts of tokens in programs, classified as operators and operands, in various equations to predict such criteria as the effort required to develop a program.

[McCabe 76] proposed a measure called cyclomatic complexity, which we shall utilize in our own research. The "cyclomatic number" measures the complexity of a software module based on its decision structure, using notions derived from graph theory. A program control graph is drawn, with nodes representing basic blocks (groups of statements that are always executed in sequence), edges representing branch instructions, and components representing subroutines. (The graph of a while loop shows three branches; the graph of an if-then-else block, four). The cyclomatic complexity is then given by:

$$e - n + 2p$$

where e is the number of edges, n is the number of nodes, and p is the number of components. McCabe suggested that any program whose cyclomatic complexity exceeded 10 should be reorganized into less complex modules in order to make testing and maintenance manageable.

[Albrecht 83] proposed a type of metric called "function points", a weighted sum of the inputs, outputs, and files used by an application. Function points were intended as a measure of productivity that would be independent of the programming language.

3.3 Limitations of Traditional Metrics for OO

[Morris 89] noted that classical software metrics, since they were developed for procedural programming, focus entirely on procedures and essentially ignore the complexity of the data model, and that the degree of code reuse in OO may significantly affect productivity even though it is not captured by the traditional metrics.

[Lake 92] noted that logic structure complexity metrics, such as McCabe's cyclomatics, are less applicable to OO than to procedural programming because true OO has no control flow issues. Applying metrics to 4 C++ applications, Lake found that one of the applications was subjectively much harder to understand because of the depth of the inheritance trees, but that two traditional metrics, lines-of-code and cyclomatic number, did not reveal it to be more complex than the others.

[Coppick 92] applied Halstead's and McCabe's metrics to the local methods (ignoring inherited methods) in objects coded in an OO derivative of LISP called LISP Flavors, and found the results to be "intuitively reasonable", but admitted that ignoring data complexity was a serious limitation.

[Tegarden 92a] considered the effects of applying three traditional metrics — lines of code, Albrecht's metric and cyclomatic number — to OO systems. For Albrecht's metric he used methods in place of operators and variables in place of operands. He applied the metrics to an OO General Ledger Account system in four ways: including and excluding

polymorphism, and including or excluding inheritance. When polymorphism was excluded, every method was given a unique name, and the required logic was inserted in the calling code to call the right method. When inheritance was excluded, all inherited code was replaced by duplicated local code. He found that the traditional metrics successfully captured the directionality of significant reductions in complexity afforded by polymorphism and inheritance, but found their order of magnitude to be insufficient.

[Barnes 93] applied three traditional software metrics (lines of code, Halstead's metric, and McCabe's metric) and four OO metrics (messages to self, messages to others, number of methods, and number of instance variables) to 55 classes implemented in the OO system Actor 3.0. He found high correlations between the traditional and the OO metrics. He proposed an extended OO language in which object metrics could be inspected by the user to help choose the higher-quality classes from class libraries.

[Kolewe 93] measured the cyclomatic complexity of a C++ program with roughly 100 classes and an average of 10 member functions per class, and found that it was never higher than 7, with the average rounding down to 0. The program was complex, but the cyclomatic numbers failed to capture this.

[Graham 95] reported a case study that found that function points correlated well with number of classes and number of methods, but did not correlate with number of messages.

3.4 The MIT OO Metrics Suite

The metrics selected for experimentation for this thesis were developed beginning with [Morris 89]. It should be noted that these metrics were not intended for automatic use or even for use by programmers; rather, they were intended to assist managers in assessing software

productivity. Morris proposed the following nine metrics, which were intended to correlate with maintainability, reusability, extensibility, testability, comprehensibility, reliability, and authorability. (Authorability was defined as the ability of the end-user to customize the product.) In particular, Morris suggested that all the proposed metrics except number 8 would enhance reusability.

1) *Number of methods per object class:* Morris suggested that the value of this metric is optimal when it is neither too high nor too low. Too many methods per class make the class too complex; too few methods per class make the application too complex in terms of methods.

2) *Maximum inheritance tree depth:* Again, Morris suggested that this value is optimal when it is neither too high nor too low. A deeper inheritance tree promotes reusability through method sharing, but may adversely affect testability and comprehensibility.

3) *Coupling between objects:* A couple exists between two objects when a method declared in one class uses a method or instance variable of another class. An "instance variable" is a variable that has a different value for every instance of the class — as opposed to a "class variable", which has the same value for all members. Couples should be minimized to maximize object encapsulation. Two forms of this metric are proposed: average number of couples per object, and maximum number of couples for any object in the application.

4) *Average Fan-in per Object:* An object's fan-in is the number of other objects that pass data to that object through messages, either directly or indirectly. The fan-in should be minimized to maximize the object's cohesion, i.e. the degree to which the data and methods encapsulated within the object are actually related.

5) *Object library effectiveness:* Morris gives two metrics: what proportion of an application's objects are library objects, and the average number of times a library object is reused. The correlation between object library effectiveness and reusability is clear.

6) *Factoring effectiveness:* A method implemented at only one location in the inheritance hierarchy is said to be unique. The factoring effectiveness is the number of unique

methods divided by the total number of methods. To maximize reusability, this quotient should be as low as possible and ideally should be 1.

7) *Degree of reuse of inheritable methods:* If a method is defined for a class, a subclass of that class may either not use the method or override the method's definition. Two metrics are proposed: the total number of actual method uses divided by the total number of potential method uses, and the total number of methods overridden divided by the total number of potential method uses. The former quotient should be maximized and the latter quotient minimized to maximize the effectiveness of the class hierarchy.

8) *Average method complexity:* To measure method complexity, Morris suggested that McCabe's formula $e - n + 2p$ be simplified to $e - n + 2$. This value should be minimized because more complex methods will probably be more difficult to test and to maintain.

9) *Application granularity:* This metric made use of Albrecht's function points, and was defined as the average number of objects per function point. This number should be maximized to minimize complexity of the objects.

[Chidamber 91] refined the above metrics to produce a suite of six "candidate metrics" for empirical study, elaborated on in [Chidamber 93]. Chidamber wanted his chosen metrics to satisfy certain theoretical properties, such as monotonicity (defined as the property that for all classes P and Q, $\mu(P) \leq \mu(P+Q)$ and $\mu(Q) \leq \mu(P+Q)$, where μ is the metric and + indicates union). Each of Chidamber's metrics is defined separately for each object class in an application; no technique, such as average or maximum, is suggested for providing a single summary metric value for the application.

1) *Weighted Methods per Class (WMC):* This is similar to Morris's Number of Methods per Class, but Chidamber suggests that the methods should be weighted by the cyclomatic complexity, so the value for each class is the sum of the cyclomatic complexities of its methods.

2) *Depth of Inheritance Tree (DIT):* This differs from Morris's inheritance tree metric in that it is defined for each class. It is the length (in the case of multiple inheritance. the maximum length) from the class's node to the root node; hence, it measures how many ancestors can potentially affect the class.

3) *Number of Children (NOC):* This is the number of immediate subclasses of a class. A greater number of children means a greater degree of reuse, since inheritance is a form of reuse, but may also indicate that the parent class was improperly abstracted.

4) *Coupling between objects (CBO):* This is the same as in Morris, except that it is defined for each class.

5) *Response set For a Class (RFC):* This is the cardinality of the set of methods that can potentially be executed as a result of a message received by the object — in other words, the transitive closure of the methods called. This number should be minimized to minimize object complexity.

6) *Lack of Cohesion in Methods (LCOM):* This is based on the intersection of instance variables used by a pair of methods. For any two methods in an object, the intersection is defined as the number of instance variables in the same object that both methods use; if there is no instance variable that they both use, the intersection is said to be null. The metric is defined as the number of null intersections minus the number of non-null intersections, or 0, whichever is greater. Lack of cohesion implies that a class should probably be split into subclasses.

[Graham 95] says that the MIT metrics "constitute one of the soundest and most practical suggestions to be found in the literature and have been widely adopted."

[Li 93] investigated whether these metrics can be used to predict maintainability. Li took all the above metrics except CBO, and added five other metrics: number of semicolons in a class, number of local methods in a class, number of local methods plus number of attributes in a class, number of send statements defined in a class, and number of abstract data types

defined in a class. Li took two commercial software products implemented in Classic-Ada™ (an OO programming language developed by Software Productivity, Inc.) and measured the maintenance effort for each class in the product as the number of lines of code changed over a three-year period. He then did a regression analysis to determine whether the metrics correlated with the maintenance effort. He found that at least 85% of the total variance in the maintenance effort was accounted for by the metrics in the population. Correlations with individual metrics are not given, but variable inflation factors show NOC, DIT, number of send statements, WMC, number of abstract data types, LCOM, and RFC to be in that order most nearly independent of the other metrics.

[Li 95] repeated this analysis looking at only the metrics that could be collected with reasonable effort from design documents: DIT, RFC, LCOM, number of abstract data types, and number of local methods. These metrics were found to account for 77% of the variability in the maintenance effort.

[Lake 92] empirically evaluated the DIT metric. He did an experiment in which programmers did program comprehension, debugging, and modification tasks in which the key class was either at the root or the leaf of an inheritance tree. The programmers working at the root performed their tasks more quickly, but there was a point of accumulated complexity beyond which this gain was lost.

[Kolewe 93] reported some experience applying the MIT metrics to moderate-sized C++ programs. He found that most problems could be accounted for by CBO values greater than 7, or by RFC values greater than 50.

3.5 Other OO metrics

[Bilow 93] regretted how little work had been done on OO metrics to date, but, on the grounds that OO analysis and OO design make critical information available earlier in the software life-cycle than earlier paradigms, predicted that OO metrics would eventually surpass traditioanl metrics in predicting software development time and quality.

[Coad 91], although not concerned with metrics, identified the following as criteria for good OO design: low interaction coupling; high inheritance coupling; cohesion of methods, attributes, and inheritance; reuse; clarity; an optimal depth of inheritance tree (considered to be about 7 for moderate-sized systems); simplicity; and minimum size of the total application. [Booch 91] suggested using coupling, cohesion, sufficiency, completeness, and primitiveness to evaluate the goodness of individual objects.

[Henderson-Sellers 91] proposed that the effort and cost of an OO software project be based on a weighted sum of the number of the object methods and object attributes in the application, numbers that could be gathered at the analysis stage. He speculated that the weighting required would be language-dependent, but did not attempt empirical validation.

[Bieman 91] devised a "call multigraph" for verbatim reuse of objects (through sending multiple messages to an object, or through multiple instantiations of a class) and an inheritance hierarchy graph for reuse with modifications (through subclassing). A call graph is a graph where the nodes are the callers and callees, and directed edges show what calls what; a call multigraph is an extension of this with one directed edge for each invocation. Bieman offered such possibilities as using edges, paths, or connected node pairs in the graph, without recommending one of them. [Karunanithi 93] proposed replacing the graphs with tables from which various types of reuse could be measured by summing the rows or columns.

[Haaland 92] proposed a very large number of metrics: method metrics (such as source code length, number of messages sent, number of temporary variables, and number of editions of the method in a shared repository); class metrics (such as number of descendants, number of instance variables, and number of refined methods); application metrics (such as number of defined classes); class edition metrics (measuring what classes a new application causes to be added to an existing class library); runtime metrics (such as method coverage, and method collaboration based on messages sent); and evolutionary metrics (number of classes added, removed, or reused between two editions of a system). Haaland applied these metrics to 13 OO projects written in Smalltalk. He found such close correlations between many of these metrics as to conclude that some of them were redundant. He made no formal attempt to validate the metrics externally, although he gave some informal discussion of correlation to reusability, programmer productivity, and cost estimation.

[Sheetz 91] proposed a large number of OO metrics to measure complexity at the variable, method, object, and class level, based on counts of local and inherited methods, variables, and properties; parameters passed and returned; polymorphism (the number of objects in which the same method name is defined); local redefinition of inherited properties and methods; fan-in and fan-out (for methods, the number of methods that respectively send messages to or receive messages from a method; also defined for objects); and fan-up and fan-down (respectively the number of superclasses or subclasses of the object).

[Tegarden 92b] opined that an accurate assessment of OO complexity should combine such structural measures with the developers' perception of the complexity of the system. [Tegarden 93] had 7 graduate students identify 148 concepts that they believed contributed to the complexity of OO systems, which they then grouped into 10 categories and ranked in order of importance. Class design issues, including encapsulation, intra-class complexity, proper

placement of methods and attributes, and number of attributes, was judged the most important category.

[Lorenz 94] proposed a large number of OO metrics, classified as project metrics and design metrics. Results of the metrics are given for 12 Smalltalk and 4 C++ applications, and "personal recommendations" among the metrics are given; but since no attempt is made to judge the relative quality of the applications, no objective basis is offered to justify the recommendations. The recommended project metrics are number of key classes (the classes subjectively judged essential to an application), person-days per class, classes per developer, and number of contracts completed. The recommended design metrics for methods are: number of message sends, method complexity (defined as a complex weighted sum based primarily on number of calls), number of methods overridden, specialization index (defined as number of overridden methods × class hierarchy nesting level ÷ total number of methods), number of friend functions, percentage of code outside any method, and number of problem reports per class. The recommended design metrics for classes are: numbers of instance and class methods and variables, depth in the inheritance hierarchy, uses of multiple inheritance, use of global variables, and class reuse.

[Graham 95] describes a tool that collects metrics based on the MIT metrics, with some additions whose automatic computation requires information from OO analysis and design. He considers the most important of these to be "task points", based on the concept of function points. The task points represent "atomic tasks"; a task is "atomic" if "further decomposition would introduce terms inappropriate to the domain".

Chapter 4
Clustering Techniques

4.1 Problem Statement

Our aim is to create a system that will transform procedural into OO code by identifying candidate objects. We regard this as a clustering problem, in which candidate objects are identified by clustering the procedures and global variables in the procedural code. We will use and compare three different algorithms, two pre-existing and one novel, for this purpose.

The various clustering algorithms define a "space" of possible clusterings. Conceptually, this space is the set of *all* partitions of procedures and variables, although in practice the space will be too large to search by brute force. Most of the points in this space are actually partial clusterings, in which some grouping has been done but some is still to be decided. From every partial clustering, there are many ways to proceed — many possible merges that can be done on the existing clusters to make the clustering less partial and more complete. Our OO metrics will be the evaluation function that we use for search control, to guide the search through the space of partial clusterings.

4.2 The Globals Based Object Finder

My first experiment was to code the "Globals Based Object Finder" given in [Liu 90]. My implementation was a 110-line AWK script, calling the standard utility `cxref` to analyse the input C program.

As an example of the algorithm, [Liu 90] gave headers for a toy program with six procedures, three of which manipulate a stack and the other three of which manipulate a queue. My implementation of the Globals Based Object Finder, given a fleshed-out version of this toy program as input, successfully disentangled into a "stack" candidate object and a "queue" candidate object. Here is the input and the output:

```
 1      typedef int element;
 2      typedef short boolean;
 3
 4      int stack[100], queue[100];
 5      int end_of_stack, beginning_of_queue, end_of_queue;
 6
 7      void init_s()
 8      {
 9      end_of_stack = -1;
10      }
11
12      void push_s (x)
13      element x;
14      {
15      end_of_stack++;
16      stack[end_of_stack] = x;
17      }
18
19      element pop_s()
20      {
21      end_of_stack--;
22      return stack[end_of_stack+1];
23      }
24
```

```
25      boolean is_empty_s()
26      {
27      return (end_of_stack < 0);
28      }
29
30      void init_q()
31      {
32      beginning_of_queue = 0;
33      end_of_queue = -1;
34      }
35
36      void push_q (x)
37      element x;
38      {
39      end_of_queue++;
40      queue[end_of_queue] = x;
41      }
42
43      element pop_q()
44      {
45      beginning_of_queue++;
46      return(queue[beginning_of_queue]-1);
47      }
48
49      boolean is_empty_q()
50      {
51      return(beginning_of_queue > end_of_queue);
52      }
53
54      main()
55      {
56      int i, j;
57
58      init_q();
59      init_s();
60      push_q(1);
61      push_q(2);
62      push_s(1);
63      push_s(2);
64      i = pop_q();
65      j = pop_s();
66      }
```

```
Candidate object 1
data members:
        queue
        end_of_queue
        beginning_of_queue
methods:
        is_empty_q
        init_q
        push_q
        pop_q

Candidate object 2
data members:
        stack
        end_of_stack
methods:
        is_empty_s
        init_s
        push_s
        pop_s
```

However, the authors' misgivings that their algorithm would generate candidate objects that were "too big" was confirmed: for all real-life programs input to the Object Finder, the output was always one large "candidate object" consisting of the entire input program.

The clustering of program elements into candidate objects can be viewed as analogous to gravitational clustering: initially, each program element stands by itself, but then an attractive force operates between the elements. Here, the attractive force is the sharing of references to global variables. Although this algorithm is a degenerate case where there is only one strength of attraction, which can be considered infinite because attraction guarantees that two elements will be in the same candidate object, it belongs to a general class of possible solutions in which the attraction can be considered finite and variable: the probability that

program elements attracted to one another will end up in the same object will not be a certainty, but will correlate positively with their attraction strength.

The over-large candidate objects produced by this algorithm motivated a quest for ways to weaken the forces that bound program elements together to form candidate objects. A very large body of clustering algorithms exist; it was decided initially to explore algorithms that had already been used on program elements. [Liu 90] cited [Hutchens 85] as work that had been done on clustering together routines that shared data, but [Liu 90] made no comment on the applicability of this work to candidate objects. Since it seemed applicable, I then proceeded to code the algorithms from [Hutchens 85].

4.3 BASILI Clustering

The algorithms described in [Hutchens 85] we here designate the BASILI algorithms, after the more famous co-author. The authors, who used as input four FORTRAN programs totalling 164,000 lines of code, were clustering procedural code based on data bindings between procedures, but they were not concerned with OO. Rather, they were trying to modularize the code in order to localize faults.

The BASILI clustering is bottom-up: the initial clusters are simple procedures, and every pair of procedures has a "binding strength", based on the number of variables the procedures share. (The authors distinguish between various types of data bindings, and say that the bindings that they used were "actual bindings": such a binding exists between two procedures if static analysis reveals that one procedure can assign it a value and that the other can use the value. The description of the kinds of data-sharing considered is not consistent. The first page of the paper defines bindings in terms of "a variable within the static scope" of

the procedures, implying that in FORTRAN only variables shared through COMMON blocks would be looked at. But the third-last page gives an example of a binding established through passing a variable as a parameter in a subroutine call.)

The most closely bound clusters are merged. The algorithms in the family differ from one another in how they re-compute the binding strength after a merge. It is not desirable simply to add the binding strengths of the merged components, since this makes merged clusters more likely to be merged again than unmerged clusters, creating what the authors call a "Black Hole Syndrome" where one cluster progressively gets bigger until it attracts everything else.

Initially, when each cluster contains one and only one procedure, for every pair of clusters i and j, the binding strength $b(i,j)$ is defined as the number of variables the two procedures share. When two clusters i and j are merged, then the binding strength of the new cluster to any outside cluster k is defined as $b(i,k) + b(j,k)$. (The clusters between which the binding strength is greatest are merged first. Initially, the binding strength between many pairs will be zero; these pairs are not yet candidates for merging.) From the binding strength, a dissimilarity d is computed, and it is the pair of clusters with the lowest dissimilarity that will be merged next. The initial dissimilarity between each pair of clusters i and j is given by

$$d = 1 / (1 - b(i,j))$$

In *single-link binding*, a traditional technique, the dissimilarity between the newly formed cluster and every other cluster is defined as the smaller of the two dissimilarities between either of the clusters that was merged and the outside cluster. In *recomputed binding*, which reflects the intuitive notion that connections of all components between two clusters should have lower dissimilarities than connections of just some of them, the dissimilarity between the merged clusters i and j is defined by:

$$d = (\mathrm{sum}i + \mathrm{sum}j - 2b(i,j)) / (\mathrm{sum}i + \mathrm{sum}j - b(i,j))$$

where sum$i$ is the number of data bindings involving i, and sum$j$ is the number of data bindings involving j. (The denominator of the above equation is never zero, because sum$i \geq b(i,j)$ and sum$j \geq b(i,j)$.) In *expected binding*, which attempts to compensate for the greater and greater proportion of the total bindings likely to lie between an arbitrary pair of clusters as merges decrease the total number of clusters, the dissimilarity is defined by:

$$d = ((\text{sum}i + \text{sum}j - b(i,j)) / (n - 1)) / b(i,j)$$

where n is the number of clusters remaining after the merge. (In my implementation, I simply set this to a very large number when $b(i,j)$ was zero.) *Weighted binding* was developed in an attempt to counteract the bias that causes Black Hole Syndrome. In weighted binding, the dissimilarity is computed as for recomputed binding, except that each occurrence of $b(i,j)$ in the equation is multiplied by a weighting function that depends on the size of the clusters merged. [Hutchens 85] does not reveal what weighting function the authors used; experimenting with various weighting functions, I found 2 divided by the square of the sum of the cluster sizes to be the best at avoiding Black Hole Syndrome.

[Hutchens 85] evaluated the algorithms presented based on how well known program faults were localized. The algorithms were run until all procedures had been merged and then the tree of clusters was examined; for each known fault, the smallest cluster in the tree containing all the procedures that the fault involved was found. The smaller that cluster was, the better the algorithm was considered to have done. The authors found that recomputed binding performed the best for the 3 large input programs and that single-link binding performed the best for one small program.

In my early experimentation when I attempted to use the BASILI algorithm without reference to metric values, I was not successful at avoiding Black Hole Syndrome with any of the first three algorithms in the family. However, reference to the metric values solved this problem, and I eventually decided on the use of single-link binding.

Because the BASILI algorithm clusters procedures only, I add an extra step in which the variables are assigned to the clusters to produce candidate objects. This assignment is done afresh after every merge; only the assignment done after the final merge will be seen by the user, but the assignments after the earlier merges are necessary so that the metrics can be computed. To determine the assignment, the candidate objects "bid" for global variables: each variable is assigned to the object with the greatest number of procedures using the variable, divided by the total number of procedures in the object. The bidding is performed and the metrics calculated before and after each merge is considered. The partition of all the program elements into candidate objects after the potential merge is compared with the partition before the merge according to the chosen metric value, adjusted to compensate for the differing number of objects as described in Chapter 6. Unlike the original authors, I do not allow the clustering to run to completion. Instead, I "hill-climb", accepting a merge only if it improves metrics values; otherwise the merge is rejected, the clusters that would have been merged remain separate, and the algorithm continues with the pair of clusters with the next-smallest dissimilarity being considered for merging.

Here is the algorithm expressed in pseudocode:

```
proc vote()
 uses_in_object[] := 0
 for v in variables do
  for p in procedures do
   if p is_used_in v then
    uses_in_object[v,object_of[p]] +:= 1
   fi
  od
 od
 for v in variables do
  u_max := 0
  for o in objects do
   u := uses_in_object[v,o] / number_of_procedures_in_object[o]
   if (u > u_max) {
    u_max := u
    object_of[v] := o
   }
  }
 }
corp

proc basili()
 merge_rejected[] := FALSE
 did_a_merge := TRUE
 vote()
 old_score := compute_metric()
 while did_a_merge do
  dissimilarity[] := compute_dissimilarities()
  min_dis := INFINITY
  for i in objects do
   for j in objects do
    if dissimilarity[i,j] < min_dis AND NOT merge_rejected[i,j] then
     min_dis = dissimilarity[i,j]
     i_to_merge := i
     j_to_merge := j
    fi
   od
  od
  i := i_to_merge
  j := j_to_merge
  merge(i,j)
  vote()
  new_score := compute_metric()
  if new_score < old_score then
   old_score := new_score
   did_a_merge := TRUE
  else
   unmerge(i,j)
   merge_rejected[i,j] := TRUE
  fi
 od
corp
```

4.4 AGGLOM Clustering

AGGLOM is a clustering program for machine learning applications, developed by Douglas Fisher and Douglas Talbert at the University of Vanderbilt, and kindly supplied by them for use in this research. It is described in [Fisher 92]. A successor to Fisher's earlier program COBWEB, it uses the same clustering criterion function, known as "category utility". It differs from COBWEB in being *non-incremental*, meaning that the output is independent of the order of the input.

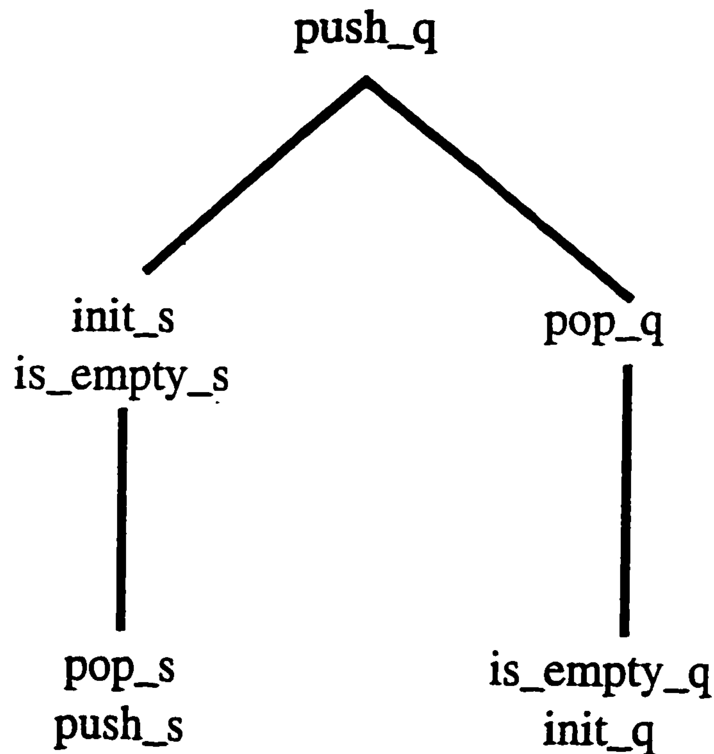
The entities here submitted to AGGLOM for clustering are the procedures from the input C program, and the attributes submitted are binary numbers corresponding to variables used and called or calling procedures: 1 if the corresponding reference exists in the input program, and 0 if not.

Here is the input to AGGLOM for the stacks-and-queues example:

```
* * * * * * * * * * * * * * 
is_empty_q 0 0 1 0 1 0is_empty_q 0is_empty_s 0init_q 0init_s
           0push_q 0push_s 0pop_q 0pop_s
init_q 0 0 1 0 1 0is_empty_q 0is_empty_s 0init_q 0init_s 0push_q
           0push_s 0pop_q 0pop_s
push_q 1 0 1 0 0 0is_empty_q 0is_empty_s 0init_q 0init_s 0push_q
           0push_s 0pop_q 0pop_s
push_s 0 1 0 1 0 0is_empty_q 0is_empty_s 0init_q 0init_s 0push_q
           0push_s 0pop_q 0pop_s
pop_q 1 0 0 0 1 0is_empty_q 0is_empty_s 0init_q 0init_s 0push_q
           0push_s 0pop_q 0pop_s
pop_s 0 1 0 1 0 0is_empty_q 0is_empty_s 0init_q 0init_s 0push_q
           0push_s 0pop_q 0pop_s
is_empty_s 0 0 0 1 0 0is_empty_q 0is_empty_s 0init_q 0init_s
           0push_q 0push_s 0pop_q 0pop_s
init_s 0 0 0 1 0 0is_empty_q 0is_empty_s 0init_q 0init_s 0push_q
           0push_s 0pop_q 0pop_s
```

For our purposes, the first input line to AGGLOM will always consist of asterisks, one for each attribute; this tells AGGLOM how many attributes there are, and indicates that they will all be nominal rather than numeric. On subsequent lines, the first element is the entity name, the plain numbers correspond to the global variables, and the numbers tagged with strings represent the procedures.

AGGLOM's output is a similarity tree. Here is the tree produced for the above input:



My post-processing of this tree inspects it from the leaves upwards. At each level, an object is considered to contain the procedures clustered there. In the first iteration, every procedure is considered to be a separate object (so, for example, in the above tree, `pop_s()` and `push_s()` would initially be regarded as being at daughter nodes of the node at which they are shown); in subsequent iterations, every pair of objects having the same parent in the tree is considered for merging. The objects "bid" for global variables, as described above

under BASILI. The partition after the potential merge is compared with the partition before the merge according to the chosen metric value; if the partition after the merge scores worse on the metrics, the merge is rejected and the tree is "cut" at that point, excluding those objects from further consideration for merging. (So, in the above tree, if a merge between `pop_s()` and `push_s()` were rejected, we would not consider merging `pop_s()` or `push_s()` with any other procedure). After a merge is accepted, the resulting object is assigned the node of the common parent in the tree, so that the grandparent node becomes the parent. The algorithm terminates when no merging of any sister nodes will improve the metric score.

Chapter 5
Clustering in Two Dimensions Simultaneously

5.1 Problem Statement

The clustering problem that we here face is unusual in that, whereas in most clustering problems there are entities to be clustered together based on their attributes and it is clear which are the entities and which are the attributes, in our problem the role of entities and attributes is reversible: it is as desirable to cluster together global variables based on the procedures that use them as it is to cluster together procedures based on the global variables they use.

The use of global variables in procedures can be represented as a matrix in which the rows represent procedures, the columns represent variables, and each entry in the matrix is 1 if the given procedure references the given variable and 0 otherwise. Here is this matrix for the stacks-and-queues example:

```
      b
      e
      g
      i
      n
      n
    e i e
    n n n
    d g d
    _ _ _
    o o o
    f f f
    _ _ _
s q s q q
t u t u u
a e a e e
c u c u u
k e k e e

0 0 1 0 1  init_q

0 0 1 0 1  is_empty_q

1 0 1 0 0  push_q

1 0 0 0 1  pop_q

0 1 0 1 0  pop_s

0 0 0 1 0  is_empty_s

0 0 0 1 0  init_s

0 1 0 1 0  push_s
```

The problem is to rearrange the matrix by row and column exchanges so that as many 1s as possible are as near to the diagonal as possible, and then to partition the matrix into submatrices that will become our candidate objects:

stack	end_of_stack	queue	end_of_queue	beginning_of_queue	
1	1	0	0	0	push_s
1	1	0	0	0	pop_s
0	1	0	0	0	init_s
0	1	0	0	0	is_empty_s
0	0	1	0	1	pop_q
0	0	1	1	0	push_q
0	0	0	1	1	is_empty_q
0	0	0	1	1	init_q

For every row and column in the original matrix, there must be one and only one submatrix that uses elements from that row or column. And we wish to maximize the number of 1s inside the submatrices and minimize the number of 1s outside; 1s outside the submatrices represent coupling between one object's methods and another object's data members, which is undesirable because if an object is coherent its methods should tend to use mostly its own data members. Thus, we are attempting to bring the matrix as close as possible to block diagonal

form, although it is unlikely that pure block diagonal form will ever be attainable if a real program is used as input.

5.2 Related Work

[Gertsberg 87], studying how to partition a database into sub-databases so as to minimize the number of transactions between the subdatabases, showed that the general partitioning problem, and even the bipartite partitioning problem (where every transaction accesses exactly two entities) is NP-complete. If, as has been widely conjectured [Illingworth 90], no NP-complete problem is polynomially solvable, any algorithm for optimal partitioning would have an exponential time complexity function, making it useless for most practical purposes.

[Hill 74] summarized work on two-dimensional scaling, called "principal components analysis" for discrete variates and "correspondence analysis" for continuous variates. He recommended an iterative process of calibrating one dimension according to the other, which he called "reciprocal averaging" because new column scores are averages of old row scores and reciprocally new row scores are averages of old row scores. He showed that if a matrix can be permuted so that all the 1s in every row and column come together, there is a unique ordering with maximum correspondence and that reciprocal averaging will eventually stabilize to it. No method for automatic partitioning is offered, although one of the applications discussed is "noda", natural groupings sought by plant ecologists between species and types of habitat.

[King 80] was concerned with the problem of grouping machines with product components in a manufacturing system to maximize the extent to which components and the machines needed to produce them would be in the same group. In this version of the problem, the matrix must always eventually be reduced to a perfect block diagonal form, since groups

must contain all the machines needed for a component; however this will not be so for the initial input, so there are what King calls "exceptional elements" which must be dealt with. King reviews two earlier algorithms that had been applied to the problem. The first is a single linkage cluster analysis algorithm, which is not two-dimensional, but groups machines according to components. The second is known as a "bond energy algorithm". The bond energy is defined as the sum of the number of pairs of 1s in adjacent rows or columns. The algorithm iterates over the columns, tries placing every other column to the left or right of the current column, and accepts the placement that maximizes the bond energy. When all the columns have been placed, the procedure is repeated on the rows. King offers a new algorithm, which he calls "rank order clustering". Each row, and later each column, is treated as a single binary number. The rows and columns are alternately sorted according to the binary number values, until the latest sort has not caused anything to move. Since sorting by binary number values favours the left columns over the right columns, clearly a large number of "exceptional elements" will degrade the utility of this algorithm, but King does not seem to address this. [King 82] shows how to make this algorithm more efficient by using hash tables, doubly linked lists, and a radix sort.

5.3 The DIAG Algorithm

Here is the algorithm that I developed:

1. A matrix is constructed with procedures as rows and variables as columns. An entry is 1 if a procedure uses the variable; 0 if not.

2. The rows in the matrix are sorted according to the position of the average 1 in each row ("leftmost to the top": the further left the average 1, the higher in the matrix the row will be after sorting); the columns are sorted according to the position of the average 1 in each column ("topmost to the left"). These alternating sorts are repeated until all 1s are as near as possible to the diagonal (experimentally found to be true after 10 iterations).

Here is an example of a perfectly sorted matrix ("perfectly" in the sense that both the rows and the columns are ordered as described above):

```
1 1 1 1 0
1 1 0 1 1
0 1 0 0 1
0 0 1 1 1
0 0 0 1 1
```

In the first row, the position of the average 1 is (0+1+2+3)/4 = 1.5. In the second row, it is (0+1+3+4)/4 = 2. In the third row, it is (1+4)/2 = 2.5.

3. A rectangular block is then cut from the top left of the matrix, its size and shape chosen so as to include as many 1s as and exclude as many 0s as possible. First, a 1×1 block is considered. If a subrow or subcolumn adjacent to and the same length as the block contains more 1s than 0s, it is added to the block.

4. A similar block is cut from the bottom right of the matrix.

5. All the rows and columns containing the blocks that we have found are stripped away, and we then continue from step 2, using the submatrix that remains as our new matrix.

6. The algorithm terminates either when no rows or columns remain, or when the remaining matrix consists entirely of 0s. It is quite common to be left with a fair size "hole", a residual of one or more meaningless blocks in the centre, consisting entirely of 0s. For the above example, we would get a 1×1 hole:

```
 ---
|1 1|1 1 0
|   |
|1 1|0 1 1
 --- -
 0 1|0|0 1
     - ---
 0 0 1|1 1|

 0 0 0|1 1|
       ---
```

This hole will correspond to procedures whose only global variable references, and variables whose only procedure uses, are in candidate objects that have already been decided upon. It would be desirable to post-process, merging these blocks or assigning the corresponding variables and procedures to other blocks; however, this was not implemented in the current research, and the program as currently constructed returns one or two "hole" objects in such cases.

7. The blocks then become our candidate objects.

(Obviously there are various ways in which one could cut the blocks from the matrix. I initially assessed the goodness of the block using the quotient of the number of 1s in the block over the number of 1s in the L-shaped area obtained by including the complete rows and columns containing the block. One cannot simply maximize this quotient, since larger blocks will tend to score better on it than smaller blocks. I therefore had to penalize large blocks by dividing the quotient by some power of the block size. Experimentally, I found the above, simpler algorithm to work better.)

5.4 Testing of DIAG

As an initial sanity check on this algorithm, a perfect block diagonal matrix:

```
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
```

was given as input to DIAG with the rows and columns randomly scrambled:

```
0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1
0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0
0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0
0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0
1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0
0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0
0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0
1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0
0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1
0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0
1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1
0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0
0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1
0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0
```

DIAG was consistently able to find the blocks in such scrambled matrices (verified over 10 runs using the same initial matrix but differently scrambled). Here is its output for the above input:

```
         f f f f f f f h h h h h f f f f f f
 t t t t i i i i i i i r r r r r o o o o o o o o o
 w w w w v v v v v v v e e e e e u u u u u u n n n
 o o o o e e e e e e e e e e e e r r r r r r e e e
 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
 v v v v v v v v v v v v v v v v v v v v v v v v v
 2 1 3 4 5 4 6 7 3 1 2 5 3 4 2 1 2 1 6 3 4 5 2 3 1
 -------                                              1
|1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  two_p4
|       |
|1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  two_p3
|       |
|1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  two_p2
|       |
|1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  two_p1
 ------- -------------                                3
 0 0 0 0|1 1 1 1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0  five_p5
        |             |
 0 0 0 0|1 1 1 1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0  five_p1
        |             |
 0 0 0 0|1 1 1 1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0  five_p6
        |             |
 0 0 0 0|1 1 1 1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0  five_p3
        |             |
 0 0 0 0|1 1 1 1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0  five_p4
        |             |
 0 0 0 0|1 1 1 1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0  five_p2
        |             |
 0 0 0 0|1 1 1 1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0  five_p7
         ------------- ---------                      5
 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1|0 0 0 0 0 0 0 0 0  three_p3
                      |         |
 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1|0 0 0 0 0 0 0 0 0  three_p1
                      |         |
 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1|0 0 0 0 0 0 0 0 0  three_p2
                      |         |
 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1|0 0 0 0 0 0 0 0 0  three_p4
                      |         |
 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1|0 0 0 0 0 0 0 0 0  three_p5
                       --------- -----------          4
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1 1|0 0 0  four_p5
                                |           |
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1 1|0 0 0  four_p1
                                |           |
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1 1|0 0 0  four_p6
                                |           |
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1 1|0 0 0  four_p3
                                |           |
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1 1|0 0 0  four_p2
                                |           |
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1 1|0 0 0  four_p4
                                 ----------- -----    2
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1| one_p2
                                            |     |
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1| one_p3
                                            |     |
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1| one_p1
                                             -----
```

The numbers 1, 2, 3, etc. represent the order in which DIAG found the blocks. These numbers will serve to remind us that DIAG does not proceed sequentially from top to bottom. The designations `one_p1`, `one_p2`, etc. refer to the first and second "procedures" respectively in the first block of the input matrix; likewise, `one_v1`, `one_v2`, etc. refer to the first and second "variables".

Next, noise was introduced by changing the value of some of the entries from 1 to 0 or 0 to 1 according to a chosen probability. It was found that DIAG could consistently recover the blocks as this probability ranged up to 10%. Here is the block-diagonal matrix with 10% noise:

```
1 1 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1 1 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1
0 0 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 1 1 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 0 1 0 0 1 1 1 1 0 1 0 0 0 0 0 0 0
0 1 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 1 1 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0 1 1 0 1
1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 0 1 1 1 1 1
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 1
0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 1
0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
```

Given the above matrix with the rows and columns randomly scrambled, DIAG produced the following output:

```
               t t t t t
 f f f f f f f h h h h h               f f f f f f
 i i i i i i i r r r r r t t t t o o o o o o o o o
 v v v v v v v e e e e e w w w w n n n u u u u u u
 e e e e e e e e e e e e o o o o e e e r r r r r r
 - - - - - - - - - - - - - - - - - - - - - - - - -
 v v v v v v v v v v v v v v v v v v v v v v v v v
 4 3 2 6 5 1 7 3 1 5 4 2 4 3 1 2 3 1 2 6 4 2 3 1 5

 -------------                                        1
|1 1 1 1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   five_p1
|             |
|1 1 1 1 1 1 1|0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0   five_p2
|             |
|1 1 1 1 1 1 1|0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0   five_p7
|             |
|1 1 0 1 0 1 1|0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0   five_p6
|             |
|1 1 1 1 1 0 1|0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0   five_p5
|             |
|1 1 0 1 1 1 1|1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0   five_p4
|             |
|1 0 1 0 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1   five_p3
 ------------- ---------                              3
 0 0 1 0 0 0 0|1 1 1 1 1|0 0 0 0 0 0 0 0 0 0 0 0 0   three_p3
              |         |
 0 0 0 0 0 0 0|1 1 1 1 1|0 0 0 0 0 0 1 0 1 0 0 0 0   three_p2
              |         |
 0 0 0 0 1 0 0|1 1 1 1 1|0 0 0 0 0 0 0 1 0 0 0 0 0   three_p5
              |         |
 0 0 0 1 0 0 1|1 1 1 1 1|0 0 0 0 1 0 0 0 0 0 0 0 0   three_p4
              |         |
 0 0 0 0 0 0 0|1 1 1 1 1|0 0 1 1 0 0 0 0 0 0 0 0 0   three_p1
               --------- -------                      5
 0 0 0 0 0 0 0 0 1 0 0 1|1 1 1 1|0 0 0 0 0 0 0 1 0   two_p1
                        |       |
 0 0 0 0 0 0 0 0 0 0 0 1|1 1 1 1|0 0 0 0 0 0 0 0 0   two_p3
                        |       |
 0 0 0 0 0 0 0 0 0 1 0 0|0 0 1 1|1 0 0 1 0 0 0 0 0   two_p4
                        |       |
 0 0 0 1 0 0 0 0 0 0 0 1|0 1 1 1|1 0 0 0 0 0 0 0 0   two_p2
                         ------- -----                4
 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1|1 1 1|0 0 0 0 0 0   one_p2
                                |     |
 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0|1 1 1|0 0 0 1 0 0   one_p1
                                |     |
 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0|1 1 1|0 0 0 0 0 1   one_p3
                                 ----- -----------    2
 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0|1 1 1 1 0 1|  four_p1
                                      |           |
 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0 0 1|1 0 1 0 1 1|  four_p4
                                      |           |
 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0|1 1 1 1 1 1|  four_p6
                                      |           |
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0|0 1 1 1 1 0|  four_p2
                                      |           |
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0|1 1 1 0 1 1|  four_p5
                                      |           |
 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0|1 1 1 1 1 0|  four_p3
                                       -----------
```

When we invert 20% of the entries in a block-diagonal matrix:

```
1 1 1 0 0 0 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1 0
1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
1 1 1 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0
0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1
0 0 0 1 1 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1 0 0 0 1 1 0 0 1
0 1 0 0 0 0 0 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 0 1 0 1 0 0 0 0 1
0 0 0 1 0 0 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 1 1 0 0 0 1 1 0
0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 1 0 1 0 1 1 0 0 0 1
0 0 1 0 0 0 0 1 1 0 0 1 1 1 0 1 1 1 0 0 0 0 1 1 1
0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1
0 0 1 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 1 1 0 1
0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 1 1 1 1 0 1
1 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 1 0 1 0
```

scramble the rows and columns, and give it to DIAG, we no longer recover the exact blocks, but the result is intuitively reasonable:

```
t t                                       t   t   t
h h   f   f       f f   f f h   h f h   f f f f f f
r r t o o o o o i o t o o r t r o r t i i i i i i
e e w u n u n n v u w u u e w e u e w v v v v v v
e e o r e r e e e r o r r e o e r e o e e e e e e
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
v v v v v v v v v v v v v v v v v v v v v v v v v
2 5 3 1 1 4 3 2 7 5 1 2 3 4 2 1 6 3 4 3 5 2 1 4 6

 -----                                                1
|1 1 0|0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0   three_p4
|     |
|1 0 1|0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0   two_p2
|     |
|1 0 1|1 0 0 0 0 0 1 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0   four_p6
|     |
|1 1 0|0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 0   three_p1
|     |
|1 1 0|1 0 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0   three_p5
 ----- -----------                                    3
 0 0 0|1 1 0 1 0 1|0 0 0 0 0 1 1 0 0 0 1 1 0 0 0 0   five_p7
      |           |
 1 1 0|1 1 1 1 1 1|0 0 0 0 1 1 1 0 1 0 0 1 0 0 0 0   three_p3
      |           |
 0 1 0|1 1 0 1 1 0|0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 1   one_p2
      |           |
 0 0 0|0 1 1 0 1 1|0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0   one_p1
      |           |
 0 0 0|1 0 1 0 1 0|1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0   four_p3
      |           |
 0 0 0|0 1 0 0 1 1|1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0   one_p3
      |           |
 0 0 0|1 0 1 1 0 1|1 1 1 1 0 0 0 0 0 0 1 0 0 1 1 0   four_p4
       ----------- -----------                        5
 0 0 0 0 0 0 0 0 1|1 1 1 1 1 0|1 1 0 0 0 0 1 1 1 0   four_p5
                  |           |
 0 0 1 0 0 0 0 0 0|1 1 0 1 0 1|0 0 0 1 1 1 0 0 0 0   two_p3
                   ----------- -------                4
 0 0 1 0 0 0 0 1 1 1 0 1 0 0 0|1 1 1 0|0 0 0 0 0 0   four_p1
                              |       |
 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1|1 1 1 1|0 0 0 0 1 1   four_p2
                              |       |
 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0|1 0 1 0|1 1 0 0 1 1   five_p1
                              |       |
 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0|1 1 0 1|0 0 1 0 0 0   two_p4
                              |       |
 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0|1 1 1 1|0 0 0 1 0 0   three_p2
                              |       |
 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0|0 1 0 1|0 1 0 0 1 0   two_p1
                               ------- -----------    2
 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 0|1 1 0 1 0 1|  five_p5
                                      |           |
 1 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1|0 1 0 1 1 1|  five_p2
                                      |           |
 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 1|1 1 1 1 1 1|  five_p3
                                      |           |
 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0|0 1 1 1 1 1|  five_p6
                                      |           |
 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1 0|  five_p4
                                       -----------
```

To show DIAG's ability to produce order out of utter chaos, here is a randomly generated matrix of 0s and 1s, and the same matrix after being organized by DIAG:

```
0 0 0 0 0 0 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1
0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1 1 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 1
0 0 1 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0
1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0
1 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 0 1
```

```
|1 1|0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
|0 1|0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0
|1 0|0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0
 0 0|1 0|1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0|1 1|0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0
 0 0|0 1|0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
 0 0|0 1|0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0
 0 0 0 0|1 1 1 1|0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
 0 0 0 0|0 1 0 0|0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 1 0|1 1 1 0 1 0|0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 1|0 1 0 1 1 1|0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 1|1 0|0 0 0 0 0 0 1 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 0|1 1|0 0 0 0 0 0 0 0 1
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0|1 0|0 0 0 1 0 0
 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0|1 0|0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0|1 1|1 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1|0 1|0 0 0 1 0 0
 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0|1 1 1|0 0 0
 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0|1 0 1|1 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0|1|0 0
 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0 0|1 0|
 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0|0 1|
 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0|0 1|
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0|1 1|
 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0|1 0|
```

Here is the output of DIAG for the stacks-and-queues example:

```
          b
          e
          g
          i
          n
          n
  e   e i
  n   n n
  d   d g
  _   _ _
  o   o o
  f   f f
  _   _ _
s s q q q
t t u u u
a a e e e
c c u u u
k k e e e

---        1
|1 1|0 0 0  push_s
|           
|1 1|0 0 0  pop_s
|           
|0 1|0 0 0  init_s
|           
|0 1|0 0 0  is_empty_s
--- -----   2
0 0|1 0 1|  pop_q
   |     |
0 0|1 1 0|  push_q
   |     |
0 0|0 1 1|  is_empty_q
   |     |
0 0|0 1 1|  init_q
    -----
```

One advantage of this algorithm is that it not only gives us candidate objects, but ranks them qualitatively: the outer blocks are likely to be the best candidate objects.

Chapter 6
Selection and Refinement of Metrics

6.1 Initial Selection of Metrics

The metrics here implemented for experimental work are based on the MIT metrics suite as described in [Chidamber 91] and summarized in Chapter 3. These metrics were selected because they seem to be the most widely respected OO metrics among researchers [Roberts 92] and because some empirical validation exists for them (the work of [Li 93], showing that they can be used to predict maintenance effort).

Of the six metrics in that suite, two (Depth on Inheritance and Number of Inheritance Tree) were discarded because they required building an inheritance hierarchy, deemed to be beyond the scope of this research; and one (Weighted Methods per Class) was discarded because attempts to minimize its value would inevitably lead to one method per class and it gives no feedback on class data members. This left:

Coupling between objects (CBO): A couple exists between two objects when a method declared in one class uses a method or instance variable of another class. In our own research, no attempt is made to distinguish instance variables from class variables, so for Chidamber's instance variable we substitute any global variable assigned as à member of the object.

Response set For a Class (RFC): the cardinality of the set of methods that can potentially be executed as a result of a message received by the object — in other words, the transitive closure of the methods called.

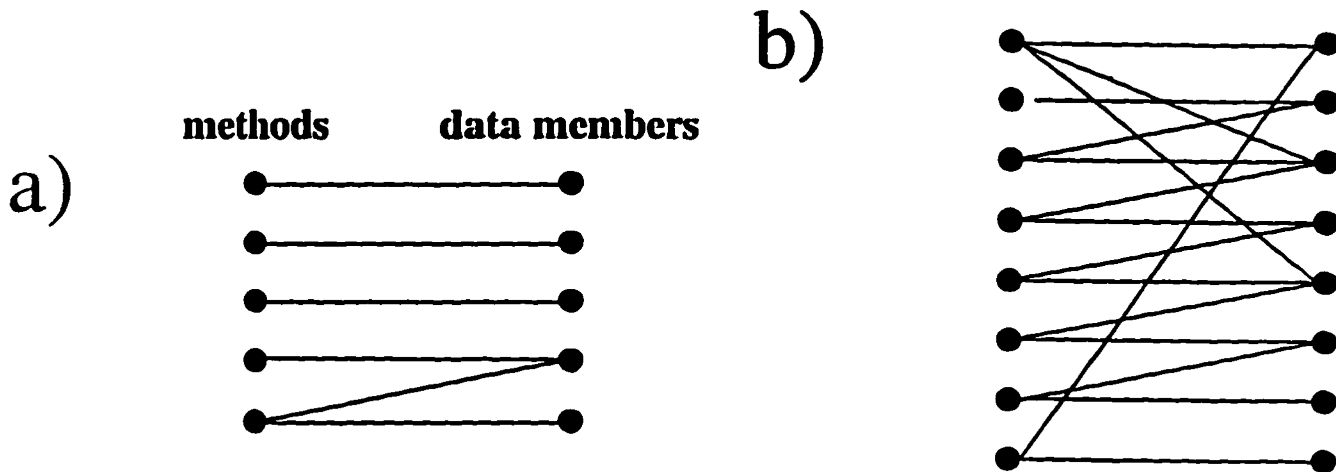
Lack of Cohesion in Methods (LCOM): considering all possible pairs of the methods in an object, the number of pairs sharing no instance variables (null intersections) minus the number of pairs sharing at least one instance variable (non-null intersections), or 0, whichever is greater. Again, in our case, we substitute "global variables assigned as members" for "instance variables".

6.2 Improvements to Lack of Cohesion in Methods Metric

In this research, many of the artefacts to which we will be applying the metrics will be pathological in comparison with objects in real OO programs: if we are to hill-climb on clusterings, then the metrics must make sense for *any* possible clusterings. To ensure that the metrics were robust enough for this purpose, and to implement the random clusterer described later in this chapter, early experimental work involved applying the metrics to randomly constructed objects.

Applied to randomly constructed objects, LCOM was found to be 0 in the vast majority of cases. This led to a quest for ways to make this metric more sensitive. The first attempt was simply to eliminate the zero cutoff. This led to the metric that we shall call NMN, for Nulls Minus Non-nulls. The value of NMN will often be negative, but we decided that even below zero, lower values might be indicative of better OO design, because even if the number of null intersections is zero, more non-null intersections suggest that the methods are more closely related.

[Henderson-Sellers 96] criticized Chidamber's LCOM metric, noting that LCOM=0 can occur "even for cases of obvious dissimilarity"; "this measure is not very discriminating .... for low cohesive structures." Henderson-Sellers also offered the following pair of examples:



Intuitively, (a) is much less cohesive than (b): we might suggest that (a) be divided into four separate classes, but it is difficult to see any way to divide (b) into smaller cohesive classes. Yet (a) has Nulls=9 and Non-nulls=1, and (b) Nulls=18 and Non-nulls=10, so according to Chidamber's metric they are equally cohesive, both having LCOM=8.

Henderson-Sellers proposes his own metric for Lack of Cohesion. For his metric, one iterates over the members of a class and computes the mean number of methods accessing a data member. If this mean value is M and the total number of methods in the class is m, then the metric is defined as $(M - m) / (1 - m)$. We shall call this metric LCOMH, for Lack of Cohesion in Methods as defined by Henderson-Sellers.

Attempting to implement this metric, I encountered two difficulties: what to do if there is only one method in the class ($m = 1$ gives us a zero denominator), and what to do if there are no data members in the class (M is undefined). In each of these cases, the value of the metric would be infinite according to the definition given. After seeking advice from Professor Henderson-Sellers, who admitted that he had overlooked these possibilities but suggested that

these were two types of non-OO artefacts (respectively a single method accessing one or more variables, and solo functions accessing no variables) that his metric could be used to identify, I decided to reflect their undesirability by having the metric return the value 1, a very unfavourable score, in both these cases.

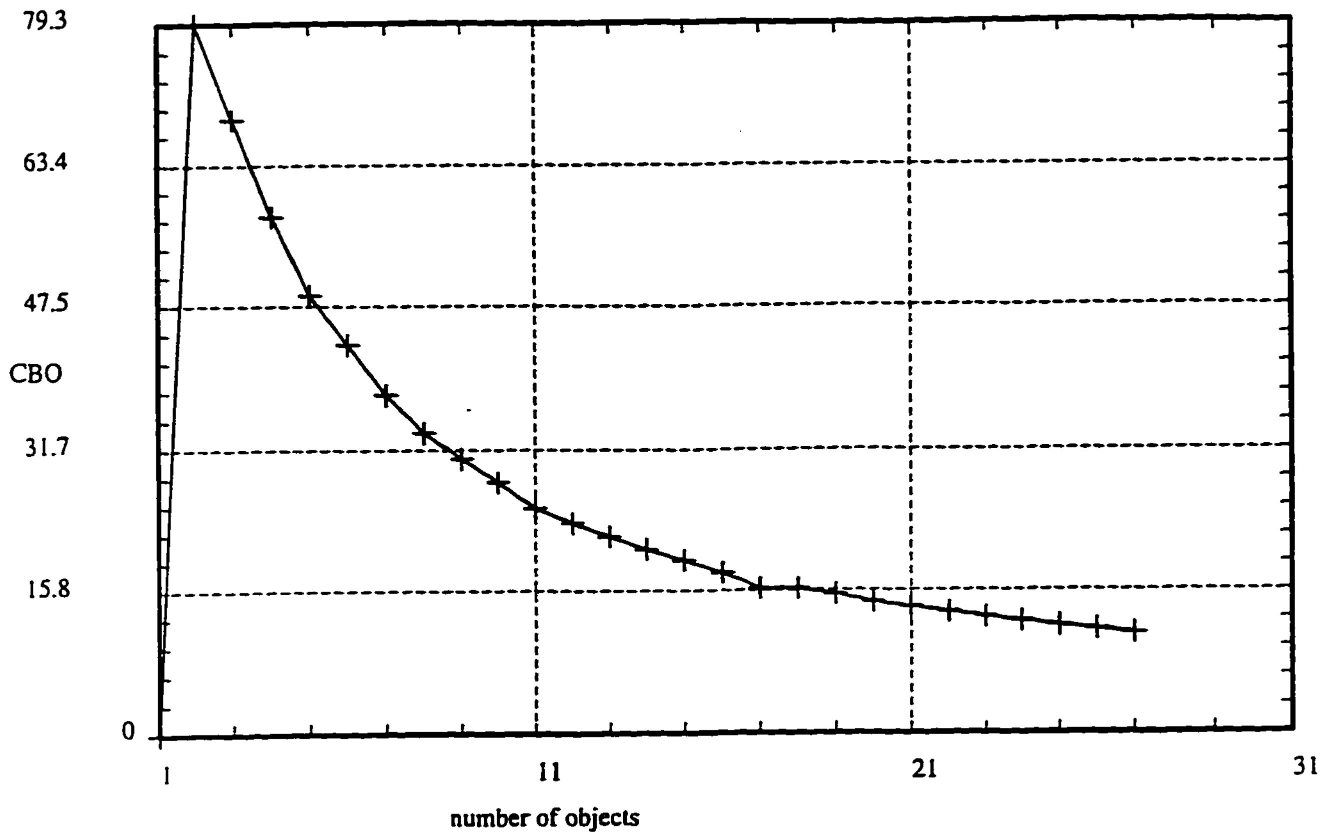
In the course of experimentation I also found that, although the subtraction and the division in the formula were reminiscent of an attempt to scale the value to the range 0 to 1, frequently values greater than 1 were being returned. For example, if an object consists of two data members and a method that accesses only one of them, then LCOMH will be (.5 - 2) / (1 - 2) = 1.5. Professor Henderson-Sellers told me: "Yes, the values of LCOM*" — what we are calling LCOMH — "were designed to run from 0 (high cohesion and thus low 'lack of cohesion') and 1. The case you propose is again an indicator of poorly designed classes since you have methods accessing attributes but also some attributes not accessed at all.... Thus again you have discovered an interesting application of LCOM*, that is, if the value appears to be greater than unity, there is a design flaw in the class itself" (e-mail message, 11 July 1996). Values greater than 1 were therefore retained.

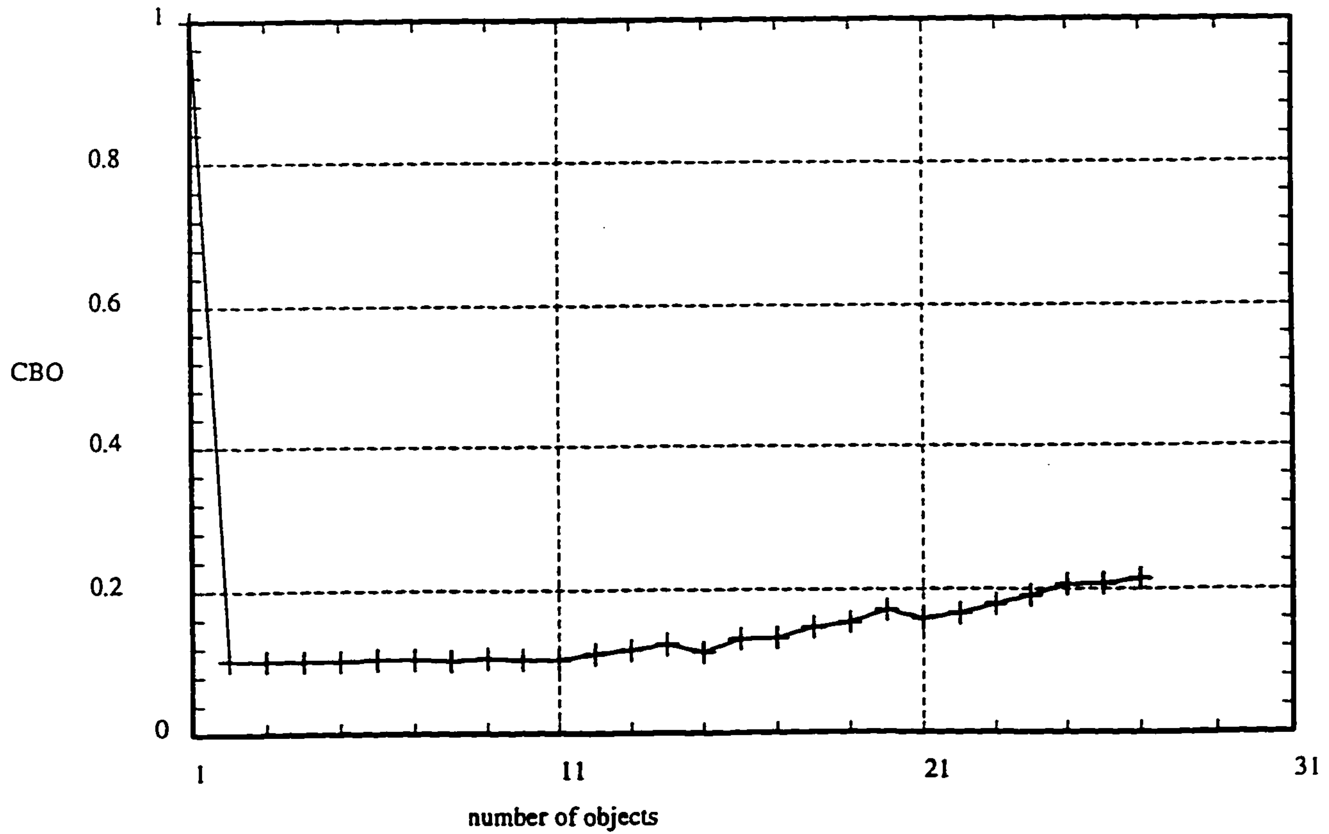
Since these scores are greater than 1, they are worse scores than for the two types of non-OO artefacts identified above. This can be justified since although the elements in the non-OO artefacts may actually belong together, it is implausible that unaccessed attributes actually belong in the candidate object.

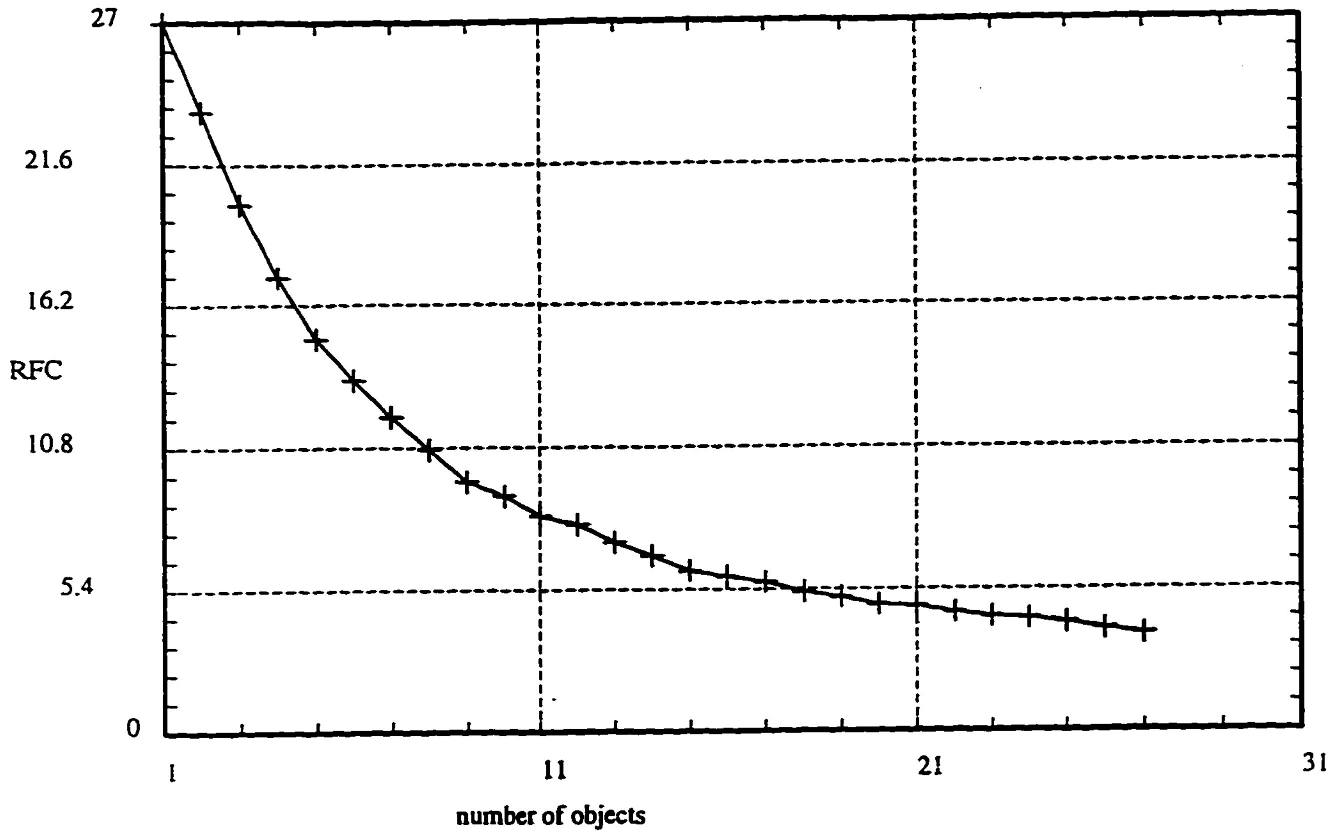
6.3 Straw Man: the Random Clusterer

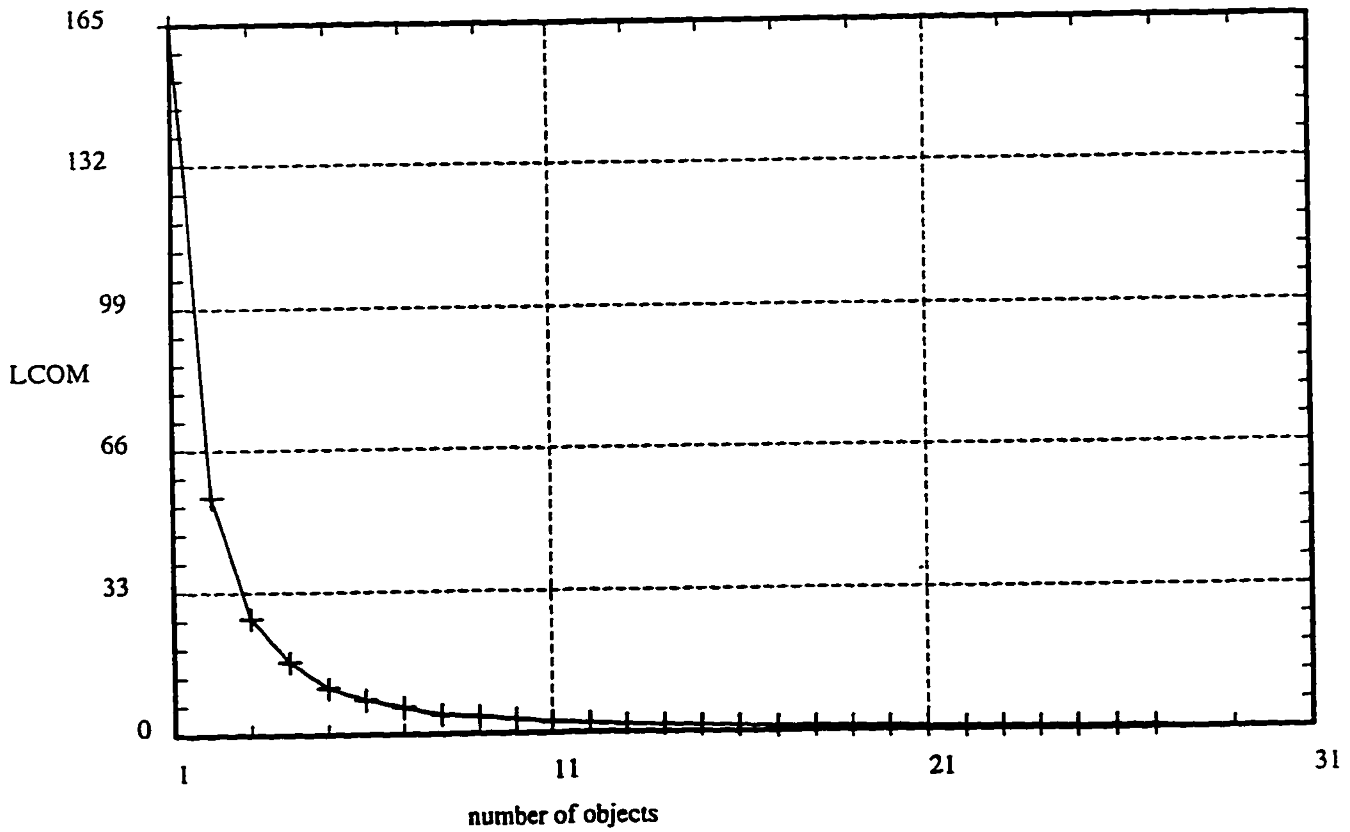
When building such a system as this, for which there is no "rival" system with which it can be readily be compared, it is always good practice to compare it with some sort of straw man system. The aim of this comparison is to show that one's system does something non-trivial.

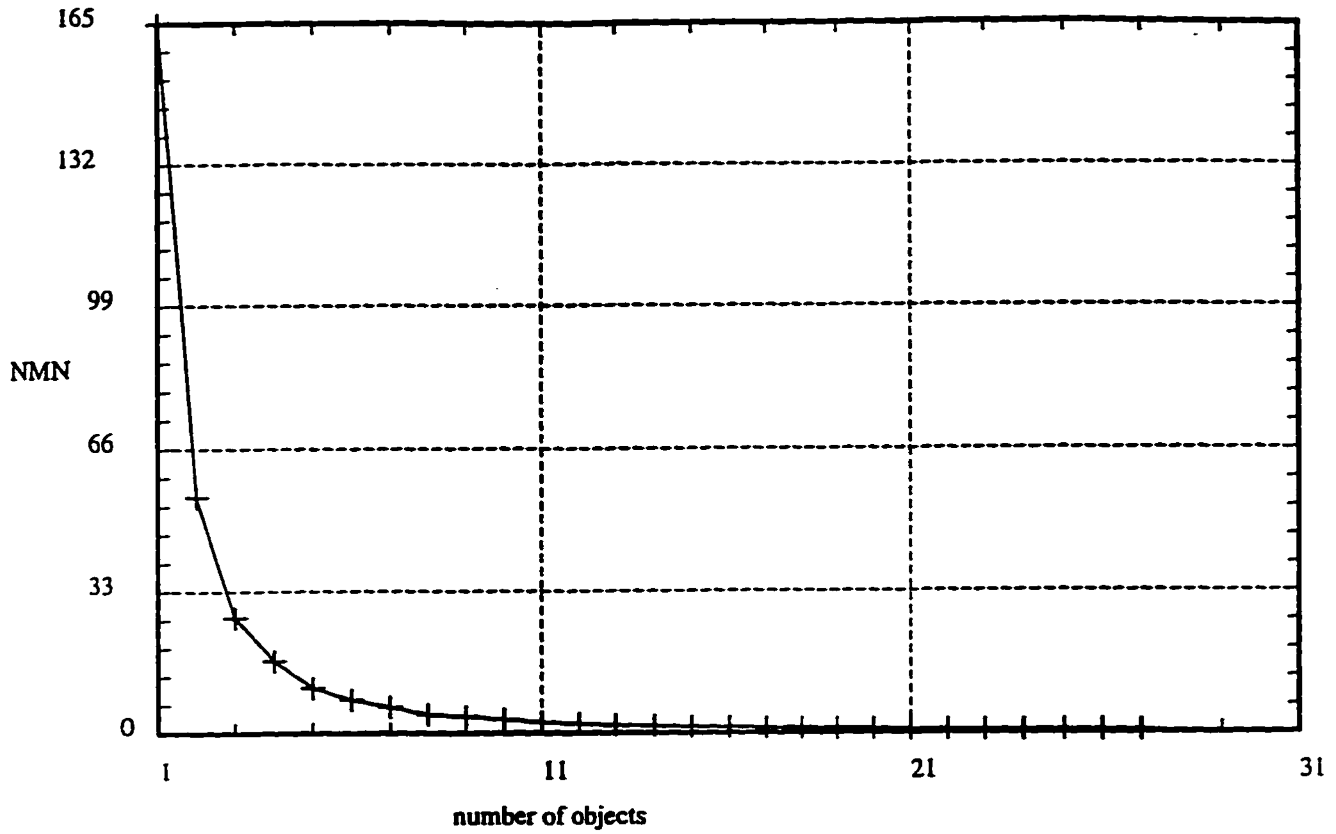
The straw man used here is a random clusterer. It randomly assigns variables and functions to objects: each is assigned to an object independently based on a uniformly distributed random number between 1 and N, the number of objects. The following graphs show metric scores for randomly constructed partitionings of one of the test programs into a given number of objects, averaged over 20 trials, each with a different random seed. The *x* axis represents the number of objects; the *y* axis represents the metric score.

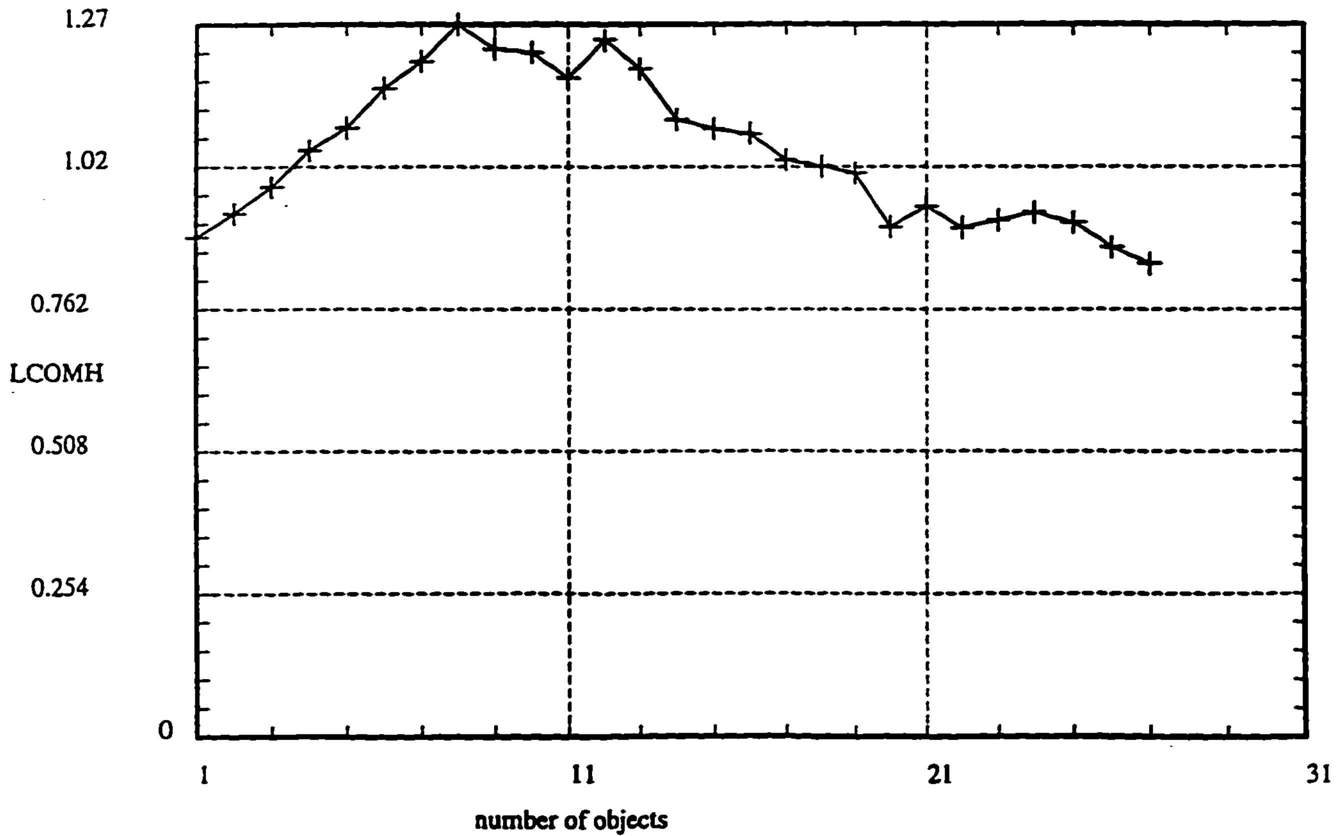

Coupling Between Objects (CBO), unnormalized
79.3
63.4
47.5
CBO
31.7
15.8
0
1
11
21
31
number of objects


Coupling Between Objects (CBO), normalized
1
0.8
0.6
CBO
0.4
0.2
0
1
11
21
31
number of objects


Response Set for Class (RFC)
27
21.6
16.2
RFC
10.8
5.4
0
1
11
21
31
number of objects


Lack of Cohesion in Methods, Chidamber (LCOM)
165
132
99
LCOM
66
33
0
1
11
21
31
number of objects


Nulls Minus Non-nulls (NMN)
165
132
99
NMN
66
33
0
1
11
21
31
number of objects


Lack of COhesion in Methods, Henderson-Sellers (LCOMH)
1.27
1.02
0.762
LCOMH
0.508
0.254
0
1
11
21
31
number of objects

Although the graphs for NMN and for LCOM look identical to the naked eye, the NMN values are in fact uniformly slightly lower. The non-monotonicity of LCOMH exhibited here suggests that NMN may be a superior metric to it.

6.4 Normalization of Metrics Against Number of Objects

A basic difficulty in applying these metrics in our research is that the metrics are for individual classes, whereas we had to compare, not one object with another, but one partitioning of material from the procedural program into objects with another. Furthermore, the number of objects in the partitioning changes (gets smaller) as we iterate. But it is not correct to compare, for example, the arithmetic mean of two partitionings consisting of a different number of objects, because (as can be readily seen from the graphs above) changes in the number of objects will cause changes in this mean value independent of *how* the objects have been partitioned. Because of the dependence of the mean metric value of the number of objects in the partition, a way must be found to normalize it if partitionings with different numbers of objects are to be compared in a meaningful way.

For the CBO metric, there is an intuitively obvious way of doing this: take the number of actual couples divided by the number of *possible* couples. For the purposes of our research, CBO was therefore redefined in this way.

For the other metrics, no such intuitive adjustment suggested itself. The other metrics were therefore artificially normalized by actually using the 20-trial mean score for randomly partitioning the relevant program into the relevant number of objects. RFC was normalized by dividing by this random score. Because NMN is sometimes positive and sometimes negative,

division did not seem advisable, so NMN and LCOMH were normalized by subtracting the random score.

6.5 A Different Kind of Metric: Pairwise Goodness

Some of the C programs used as input for experimentation were also hand-translated to C++, or at least the programmer kindly undertook on my behalf to indicate how he would proceed if he were to undertake such a translation. Hence the need arose to measure how similar a given partitioning by the experimental system was to the manual mapping. For this purpose, an original similarity metric which we shall call *Pairwise Goodness* was devised.

The Pairwise Goodness (PG) of two partitionings of the same set of entities consists of two numbers, lying between 0 and 1. For any entity i, let $o_1(i)$ be a function returning a number identifying the object to which i has been assigned in the first partitioning, and let $o_2(i)$ be a function returning a number identifying the object to which i has been assigned in the second partitioning. Let d_1 be the number of pairs of entities i and j for which $o_1(i) = o_1(j)$, and let d_2 be the number of pairs of entities for which $o_2(i) = o_2(j)$. Let n be the number of pairs of entities for which both $o_1(i) = o_1(j)$ and $o_2(i) = o_2(j)$. Then $PG_1 = n / d1$ and $PG_2 = n / d2$.

For example, if we compare the partitionings (A B) (C D) and (A B C) (D), then d_1 is 2 (for the pairs (A B) and (C D)), d_2 is 3 (for the pairs (A B), (A C), and (C D)), n is 1 (for the pair (A B)), and PG is (1/2, 1/3).

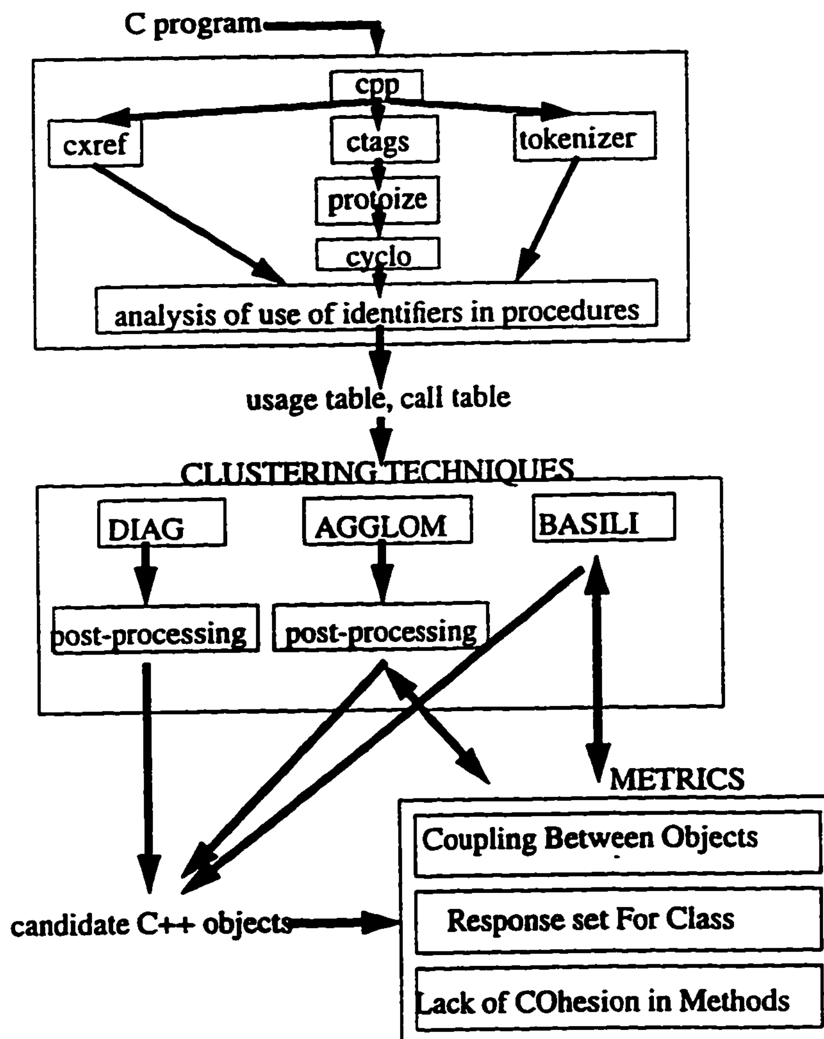
In our case, the first partitioning is the computer-generated one and the second partitioning is the hand-constructed one. Clearly, an increase in n means that the partitionings are more similar, because if two entities are assigned to the same object in one partitioning they

will be more likely to have been assigned to the same object in the other partitioning. The denominators d_1 and d_2 provide scaling so that the quotient approaches 1 as the partitionings become identical. The reason two numbers are needed is that it is not significant to score well on only one of them; for example, if in the hand-constructed partitioning we assign all entities to one object, then o_2 will always return the same value, so d_1 will equal n and PG_1 is guaranteed to be 1. When two pairwise goodnesses must be compared, the most mathematically elegant way might be to take the geometric mean of the two numbers; but on visual inspection, comparison is easy if one merely uses the smaller of the two numbers.

It is not to be expected that the numbers in the Pairwise Goodness will ever approach the optimal value, 1, in our work. When transforming a program to OO form, many entities of the original program may be discarded altogether. (In the computer representation of the hand mapping, discarded entities have been assigned to a single "junk" object.) New entities may be added. The DIAG algorithm generates residual "candidate objects" in the centre that cannot realistically be hoped to have OO significance. Different OO programmers may transform the same procedural program into different (but perhaps equally valid) OO organizations. And the ability to identify just *some* objects is not evidence of failure. For all these reasons, the Pairwise Goodness should be looked on as a relative and not an absolute measure.

Chapter 7
Design of the Experimental System

The software that converts and evaluates test programs consists of 1335 original lines of AWK code and 997 original lines of C code, and several pre-existing utilities. Its organization is summarized in the figure below.



First, a test C program is run through the standard Unix C pre-processor, **cpp**, so that subsequent parsing stages need not concern themselves with include files and macros. The output from cpp goes in three directions. First, it is run through **cxref**, the standard Unix C cross-reference utility. Second, it is decomposed into standalone ANSI C functions with the help of **ctags**, the standard Unix utility for locating definitions in C programs, and **protoize**, a standard Unix utility for Kernighan-and-Ritchie C function headers to conform to ANSI C function prototypes; the resulting ANSI C functions are then presented to **cyclo**, a program written by Roger Binns of Brunel University for analysing cyclomatic complexity. (In the early stages of experimentation, it was believed that cyclomatic complexity would play a role in the computation of the Weighted Methods per Class metric and in similar scaling of other metrics. But in the final version of the tool, no use is made of cyclomatic complexity; the **cyclo** program is retained merely for its abilities to parse C.) Third, it is run through original tokenization software, so that C structure members may be distinguished from other global variables. From the output of these utilities, the master AWK script constructs a usage table (indicating which procedures use which global variables) and a call table (indicating which procedures call which other procedures). The reason so many different utilities were used is that the standard Unix utilities were found not to provide sufficient information to narrow down the information that was needed — for example, to distinguish procedures actually defined in the input program from procedures in external libraries. The motive for making use of pre-existing parsing tools was to make the original code in the tool as independent as possible to the choice of source language.

The random trials then begin. For every number of objects from 1 to the number of procedures in the input program, there are 20 trials in each of which all the procedures and global variables in the input program are randomly assigned to one object and the metrics are evaluated. For every set of 20 trials, the mean metrics result is saved so that the normalization

described in Chapter 6 can be performed. For each metric, the best random result for all trials is also output to the user.

If there is a manual mapping to objects, the metrics are now evaluated for it.

Information from the tables is now passed to each of the three clustering techniques.

DIAG is passed a matrix in which the rows represent procedures and the columns represent variables. The candidate objects constructed by DIAG are adopted unaltered, and the metrics (and, if applicable, the pairwise goodness) are evaluated for them.

AGGLOM is passed a list of input lines corresponding to the procedures, and its output is then post-processed as described in Chapter 4. The implementation of BASILI is entirely within the AWK script. (In the figure, a "post-processing" step is shown for DIAG and for AGGLOM because these are implemented partly as external programs and partly within the AWK script. This does not refer to the "bidding" of candidate objects for variables, which is done by AGGLOM and BASILI, but not by DIAG.)

For both AGGLOM and BASILI, hill-climbing is done successively on each of the metrics CBO, RFC, LCOMH, and NMN, and then simultaneously on a combination of CBO, RFC, and LCOMH, designated as "combined" below. When the combination is used, a merge is accepted if it improves more of the metrics scores than it makes worse.

The output offered to the users is the set of candidate objects produced by each of the above techniques, with the component variables and procedures displayed graphically in a manner suggested by the DIAG algorithm, although for the techniques other than DIAG the order of the candidate objects is arbitrary; the scores for each of the final partitionings on each

of the metrics values; and, if applicable, the Pairwise Goodness of each of the partitionings as compared to the manual assignment of objects.

Chapter 8
Experimental Results

8.1 The Test Programs

I used the following C programs as test input for my tool.

8.1.1 ARGOT

The program ARGOT is an implementation by Christopher Drummond of the Department of Computer Science, University of Ottawa, of Craig Shaefer's genetic algorithm of the same name, described in [Shaefer 87]. It was written in 1991 and consists of 900 lines of C code. In 1994, Mr. Drummond at my request identified what objects he would construct if he were to redesign the program on OO lines, and which global variables and procedures in the C program would correspond to them. But the actual conversion of the code to OO was not performed.

8.1.2 DIAG

DIAG is my own matrix block diagonalization program, described in Chapter 6. It consists of 200 lines of C code and was written in 1994. I decided not to manually convert it to OO, for fear of being biased in favour of the methods used by my own tool.

8.1.3 RAY

RAY is a ray-tracing program written in 1991 by Steven Collins of the Department of Computer Science, Trinity College, Dublin, consisting of 3400 lines of C code. In 1992 it was translated into OO form consisting of roughly the same number of lines of C++ code by Andrew Condon of the same department. Ray-tracing is a brute-force technique for generating highly realistic computer-generated images including perspective projection, hidden surface removal, reflections, and shadows; it is described in [Harrington 1987]. The conversion to OO also involved a change in functionality: the program had to conform to a particular paradigm of parallel computation that was being studied.

8.1.4 RL

RL is an artificial intelligence program for inductive rule learning, written by Foster John Provost of the Department of Computer Science, University of Pittsburgh, and consisting of 2000 lines of C code. It was translated into OO form consisting of 4200 lines of C++ code by Dan Henessy of the same department; the C++ headers were automatically generated by OOTher, an OO documentation tool that stored higher-level descriptions of the objects. The conversion to OO also involved various changes in functionality. The work is described in [Provost 94].

8.1.5 MMS

MMS is a program for interactive molecular modelling, written in the early 1980s by Steve Dempsey of the Department of Chemistry, University of California at San Diego. It consists of 21,000 lines of C code.

No OO version of MMS was available.

8.2 Results and Analysis

These are the abbreviations used in stating the results:

CBO = Coupling Between Objects
RFC = Response set For a Class
LCOMH = Lack of COhesion in Methods, as redefined by Henderson-Sellers
NMN = Nulls Minus Non-nulls
PG = Pairwise Goodness (a pair of values)

It should be remembered that the values deemed desirable for all the metrics are the smallest possible values, except PG, for which the largest value is desirable.

8.2.1 ARGOT

	CBO	RFC	LCOMH	NMN	PG	
By hand	0.077	8.857	0.747	4.714	1.000	1.000
AGGLOM(CBO)	0.058	7.364	0.762	-0.364	0.323	0.259
AGGLOM(RFC)	0.061	7.538	0.724	-0.462	0.362	0.240
AGGLOM(LCOMH)	0.061	7.538	0.724	-0.462	0.362	0.240
AGGLOM(NMN)	0.053	7.333	0.633	-1.000	0.458	0.310
AGGLOM(combined)	0.061	7.538	0.724	-0.462	0.362	0.240
DIAG	0.082	5.588	0.868	0.941	0.373	0.121
BASILI(CBO)	0.047	5.467	0.695	-0.733	0.306	0.192
BASILI(RFC)	0.076	4.000	1.000	0.000	0.463	0.198
BASILI(LCOMH)	0.047	5.312	0.721	-1.125	0.422	0.313
BASILI(NMN)	0.044	3.375	0.969	-2.750	0.448	0.441
BASILI(combined)	0.051	4.941	0.658	-0.824	0.389	0.201
BestRand(CBO)	0.097	7.917	1.214	2.167	0.175	0.077
BestRand(RFC)	0.270	3.593	0.981	0.333	0.226	0.038
BestRand(LCOMH)	0.253	3.833	0.707	1.444	0.187	0.064
BestRand(NMN)	0.270	3.593	0.981	0.333	0.226	0.038
BestRand(PG)	0.113	20.333	0.992	34.333	0.216	0.380

First of all, it is encouraging to note that for all the metrics with the exception of the RFC, we have outperformed the best of the random partitionings: since the `argot` program had 27 procedures that access global variables, and since there were 20 random trials for each possible number of objects, there were $20 \times 27 = 540$ random trials altogether.

A surprise is how often we outperform the metrics on the hand-partitioning. Here is the hand-partitioning:

Block		oldpop	val	newpop	CrossoverRate	maxgen	gen	no_cities	argot	MutationRate	pmutation	pcross	lchrom	old_parms	new_parms	lastAvg	variance	sig_sum	popsize	best_chrom	nmutation	fitnessDifference	min	ultra_best	ncross	sumfitness	old_min	max	_iobuf	u	_iob
1	main	1	0	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	1	0	0	0	1	0	0	1	1	1	1	0	0
	ultra	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
	generation	1	0	1	0	0	1	1	1	0	1	1	1	0	0	0	0	1	1	1	1	0	0	1	1	1	0	1	0	0	0
	initialise	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	1	1	0	0	0	1	1	0	1	1	1	0	0	0
	do_argot	0	1	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
2	crossover	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
	mutation	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
3	bubble_sort_tours	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	tlen	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	distx	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	value	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	dist	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	quick_sort_tours	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	decode	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
	tlenx	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	power	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	swap	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	select	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0	0
	pop_stats	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	1	1	1	0	1	1	1	0	0	0
	initpop	0	0	1	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
	parm_stats	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
5	repchar	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	report	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	1	1	1	0	0	1

```
0 0 0 1 1 0 1 1 1 0 0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0|0 0 1  initdata
                                                   |
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 1  writechrom
                                                             6
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0  rnd
                                                   |
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0  flip
                                                    -----    7
```

(In these diagrams, vertical lines that are not edges of boxes represent candidate objects, such as 5 and 6 above, to which procedures have been assigned, but no global variables; horizontal lines that are not edges of boxes represent candidate objects, such as 7 above, to which global variables have been assigned, but no procedures.)

And here is BASILI(NMN), which performs significantly better than the hand-partitioning on every metric:

```
        f
        i
        t
        n
        e                 C
        s                 r             M
        s                 o             u
      b D u   s           s             t
      e i l o u           s     n     n a     n p
      s f t l m           o v   o     e t     m m
l s p t f r d f o         v a   _     w i     u u
a i o _ e a _ i l   o   n e r m c _   _ o     t t n p l
s g p c r _ p t d   l   e r i a i i   p n     a a c c c   a
t _ s h e b a n _   d   w R a x t o   a R _   t t r r h   r
A s i r n e r e m m p v p a n g i b m r a i   i i o o r g g
v u z o c s m s i a o a o t c e e u i m t o   o o s s o e o
g m e m e t s s n x p l p e e n s f n s e b u n n s s m n t

 -------------------                                         1
|0 1 1 0 0 0 0 1 0 1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  select
|                   |
|1 1 1 1 1 1 0 1 1 1|0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0  pop_stats
|                   |
|0 0 1 0 0 0 1 0 0 0|0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0  decode
 ------------------- -------------------                     2
0 1 1 0 0 0 0 1 1 1|1 0 1 1 0 1 1 1 1 1|0 0 0 1 0 1 0 1 1  main
                   |                   |
0 0 0 0 0 1 0 0 0 0|0 0 0 0 0 1 0 0 1 0|0 0 0 0 0 0 0 0 0  ultra
                   |                   |
0 0 1 0 0 0 0 0 0 0|0 0 1 0 0 0 1 0 0 1 0|0 0 0 0 0 0 1 0 0  initpop
                   |                   |
0 0 0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0 1 0|0 0 0 0 0 0 0 0 0  value
                   |                   |
0 0 0 0 0 0 0 1 1 1|0 0 0 0 0 0 1 0 1 1 0|1 0 1 0 1 0 0 1 0  report
                   |                   |
0 1 1 1 0 1 0 1 0 1|1 0 1 0 0 0 1 0 0 0 0|0 0 1 1 1 1 1 1 1  generation
                   |                   |
0 0 1 0 0 0 0 0 0 0|0 0 0 1 0 1 1 0 0 0 1|1 0 1 0 1 0 1 0 1  initdata
                   |                   |
0 0 1 0 0 0 1 0 0 0|0 0 0 0 1 0 1 0 0 1 0|0 0 0 0 0 0 0 0 0  parm_stats
```

```
                   |                     |
0 1 1 0 0 1 0 1 1 1|0 0 1 0 0 0 1 0 1 1 0|0 0 0 0 0 0 0 0 0   initialise
                   |                     |
0 0 0 0 0 0 0 0 0 1|0 1 0 0 0 0 1 0 1 1 0|0 0 0 0 0 0 0 1 0   do_argot
                    ---------------------                      3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0 0   bubble_sort_tours
                                          -                    4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1|0 0 0 0 0 0 0 0   repchar
                                          -                    5
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 0   tlen
                                            -                  6
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1|0 0 0 0 0 0 0   distx
                                            -                  7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0 0 0 0 0   rnd
                                                               8
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1|0 0 0 0 0 0 0   dist
                                                               9
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0 0 0 0 0   flip
                                                               10
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0 0 0 0 0 0   quick_sort_tours
                                              ---------        11
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1 1 1 1 1|0 0   crossover
                                              ---------        12
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0   tlenx
                                                               13
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0   power
                                                               14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|0 0   swap
                                                        ---    15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0|1 1|  mutation
                                                        ---    16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0|  writechrom
```

On inspection of the code, several reasons emerge for the discrepancy between what the human programmer favoured and what the metrics favour. One is that, although the procedures assigned to the same object in the hand-partitioning do share data, they do so mostly through pointers passed as parameters, which is something that none of the metrics try to measure. Thus, `init_pop()`, which initializes a population at random, uses very much the same data as `pop_stats()`, which reports statistics on a population; but the data are shared through a parameter `newpop` which is passed to each routine, and which the metrics have no way of detecting. Another is that in the hand-partitioning, the driver routines for the program are placed in a separate object from the utility routines, but the utility routines have not been cleanly coded, so that they share some global variables with the drivers. This is the case, for example, with the routine `decode()`, and its use of the global variable `popsize`. In the tool's clustering, coherence is achieved by grouping the utilities and the drivers into a single object; but this does not actually represent good OO design.

It is encouraging that BASILI(NMN), which scores best on the target metrics, also scores best on the pairwise goodness.

We note that the best result with a metric is often not achieved by hill-climbing on that particular metric. NMN seems to be a good metric to hill-climb on: AGGLOM(NMN) scores better than the other AGGLOM results on every metric, and BASILI(NMN) scores better than the other BASILIs on every metric except LCOMH. LCOMH is the only metric on which any AGGLOM result outperforms every BASILI result. DIAG outperforms all the AGGLOM results on RFC, but in all other respects does worse than the other two techniques.

8.2.2 DIAG

	CBO	RFC	LCOMH	NMN
AGGLOM(CBO)	0.107	8.250	0.500	-9.250
AGGLOM(RFC)	0.114	4.200	0.900	0.800
AGGLOM(LCOMH)	0.109	4.125	0.385	-2.000
AGGLOM(NMN)	0.110	4.714	0.433	-2.857
AGGLOM(combined)	0.109	4.125	0.385	-2.000
Using DIAG:	0.111	4.571	0.629	-4.714
BASILI(CBO)	0.087	6.250	0.730	-3.500
BASILI(RFC)	0.112	2.316	1.000	0.000
BASILI(LCOMH)	0.088	4.571	0.471	-3.714
BASILI(NMN)	0.127	8.250	0.560	-20.500
BASILI(combined)	0.092	3.727	0.670	-1.364
BestRand(CBO):	0.122	13.000	0.750	-10.500
BestRand(RFC):	0.241	2.000	0.737	0.158
BestRand(LCOMH):	0.241	2.000	0.737	0.158
BestRand(NMN):	1.000	19.000	0.759	-69.000

BASILI(CBO) does best on CBO; BASILI(RFC) does best on RFC: AGGLOM(combined) and AGGLOM(LCOMH), which have found the same partitioning, do best on LCOMH; BASILI(NMN) does best on NMN. We got a better random result on NMN, but this was greatly at the expense of the other metrics. Here is the partitioning found by AGGLOM(combined) and AGGLOM(LCOMH):

matlft	size_penalty	_iobuf	matbot	matrgt	mattop	maxvarlen	n_columns	candfile	_iob	a	procname	n_rows	p	varname	Group	
1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	main
1	0	0	1	1	1	0	0	0	0	0	0	0	0	0		botbox
1	1	0	1	1	1	0	0	0	0	1	0	0	0	0		getrat
1	0	0	1	1	1	0	0	0	0	0	0	0	0	0		topbox
1	0	0	1	1	1	0	0	0	0	1	0	1	0	1		sort_columns
0	0	0	0	0	0	1	1	0	1	0	0	1	1	1	2	print_picture
1	0	0	1	1	1	0	1	0	0	1	1	0	0	0		sort_rows
0	0	0	0	0	0	0	1	1	0	0	1	0	1	1		put_box_in_picture
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	3	more_1s_than_0s
0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	4	csd_rows
0	0	0	0	0	0	0	1	0	0	1	1	0	0	0		csu_rows
0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	5	print_matrix
0	0	0	0	0	0	0	1	0	0	1	0	1	0	0		random_matrix
0	0	0	0	0	0	0	1	0	0	1	1	1	1	0	6	put_nums_in_picture
0	0	0	0	0	0	0	1	0	0	0	0	1	1	0		init_picture
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	7	all_0s
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		sort
0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	8	csl_columns
0	0	0	0	0	0	0	0	0	0	1	0	1	0	1		csr_columns

Intuitively, there are four possible objects in the program: the entire matrix (which is the global variable a); a box that we cut off from the top left or bottom right of the matrix (the procedures topbox() and botbox() find the box, and getrat() determines the ratio of 0s to 1s that we use to determine how good the box is); the matrix remaining in the centre when the boxes have been cut away (delimited by mattop, matbot, matlft, and matrgt); and the picture (the global variable p) including matrix, lines, and procedure and variable names presented to the user.

It is encouraging that the three procedures that operate on the box have been assigned to one candidate object. And the procedures that operate on the picture have been grouped into two objects with only one extraneous procedure; if there were a way the tool could sense that the variable p was "important" (perhaps by the amount of storage space allocated for it), these two candidate objects would no doubt have become one. The two procedures related to the entire matrix are also the only procedures (init_matrix() and rand_matrix()) in a candidate object, with one of the relevant variables (n_rows but not n_columns), although the code in main() that was assigned to the box object (it initializes mattop, matbot, matlft, and matrgt, and causes termination when mattop has reached matbot and matlft has reached matrgt) would more properly be assigned to this object. But there is no trace of a candidate object corresponding to the matrix remaining in the centre; its procedures (sort_rows(), sort_columns(), csd_rows(), csu_rows(), csl_columns(), and csu_columns(); the last four do circular shifts) are scattered everywhere. Part of the problem seems to be that the tool cannot detect that n_rows and n_columns are attributes of a single potential object; this seems to be because AGGLOM does not cluster variables based on procedures, only procedures based on variables. DIAG, which does both, does better in this respect:

p	matlft	matbot	matrgt	mattop	procname	candfile	_iobuf	maxvarlen	_iob	size_penalty	a	n_columns	varname	n_rows		
1	0	0	0	0	0	0	0	1	1	0	0	1	1	1	print_picture	1
0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	botbox	3
0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	topbox	
0	1	1	1	1	0	0	0	0	0	1	1	0	0	0	getrat	
0	1	1	1	1	1	0	0	0	0	0	1	1	0	0	sort_rows	
0	1	1	1	1	0	0	0	0	0	0	1	0	1	1	sort_columns	
0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	main	
0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	csd_rows	5
0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	csu_rows	
1	0	0	0	0	1	0	0	0	0	0	1	1	0	1	put_nums_in_picture	
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	more_1s_than_0s	7
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	sort	6
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	all_0s	4
0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	print_matrix	2
1	0	0	0	0	1	1	0	0	0	0	0	1	1	0	put_box_in_picture	
0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	csr_columns	
0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	csl_columns	
1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	init_picture	
0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	random_matrix	

But even with DIAG, we see a similar problem: if the variables `procname` and `varname` had been clustered together, we would not see the misgroupings that we do. This suggests the need for a stronger method of clustering variables than merely their being shared by procedures; but there is a certain OO sense to their separation: intuitively, each row might be considered a separate object and each procedure name an attribute of it, and similarly for columns and variable names, although it is difficult to see how to implement this. If the names are not to be considered attributes of rows and columns, they should be in a separate object, and this is an argument for "slicing" procedures so that the parts of the sort and circular shift procedures that swap rows and columns and the parts that swap the corresponding procedure and variable names could become methods of separate objects.

BASILI(NMN) produced what is intuitively the best solution, which scored much better on NMN than any other non-random technique, although it did not score best on any other of the metrics:

Group	Function	p	maxvarlen	n_columns	varname	n_rows	candfile	_iob	size_penalty	a	_iobuf	procname	matlft	matbot	matrgt	mattop
1	print_picture	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0
	put_box_in_picture	1	0	1	1	0	1	0	0	0	0	1	0	0	0	0
	init_picture	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
2	main	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1
	all_0s	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	sort_rows	0	0	1	0	0	0	0	0	1	0	1	1	1	1	1
	put_nums_in_picture	1	0	1	0	1	0	0	0	1	0	1	0	0	0	0
	csd_rows	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0
	csu_rows	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0
	more_1s_than_0s	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	getrat	0	0	0	0	0	0	0	1	1	0	0	1	1	1	1
	print_matrix	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0
	sort_columns	0	0	0	1	1	0	0	0	1	0	0	1	1	1	1
	csl_columns	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0
	csr_columns	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0
	random_matrix	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0
3	sort	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	botbox	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
	topbox	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Our confidence in the NMN metric increases.

The remaining programs are too large for candidate objects to be to be shown here.

8.2.3 RAY

	CBO	RFC	LCOMH	NMN	PG	
By hand	0.025	6.889	0.969	8.500	1.000	1.000
AGGLOM(CBO)	0.016	6.417	0.881	1.875	0.241	0.168
AGGLOM(RFC)	0.019	5.759	0.850	0.828	0.334	0.123
AGGLOM(LCOMH)	0.018	6.185	0.826	0.667	0.381	0.150
AGGLOM(NMN)	0.016	6.680	0.864	-0.120	0.371	0.210
AGGLOM(combined)	0.018	6.185	0.826	0.667	0.381	0.150
DIAG	0.032	9.667	0.772	18.810	0.134	0.166
BASILI(CBO)	0.010	3.500	0.972	8.731	0.190	0.643
BASILI(RFC)	0.023	4.186	1.000	0.000	0.462	0.103
BASILI(LCOMH)	0.014	4.385	0.874	-0.744	0.381	0.205
BASILI(NMN)	0.014	4.974	0.919	-1.526	0.364	0.283
BASILI(combined)	0.015	4.429	0.852	-0.690	0.355	0.173
BestRand(CBO)	0.032	6.889	1.057	2.333	0.191	0.053
BestRand(RFC)	0.215	3.981	1.056	0.481	0.117	0.016
BestRand(LCOMH)	0.154	4.000	0.820	0.569	0.179	0.021
BestRand(NMN)	0.135	4.085	0.887	0.339	0.180	0.018
BestRand(PG)	0.035	24.500	1.069	44.000	0.146	0.167

BASILI(CBO) does best on CBO and RFC; BASILI(NMN) does best on NMN, outperforms the hand-partitioning on all target metrics, and gives the best pairwise goodness; DIAG does best on LCOMH. None of the techniques produced any candidate objects resembling a whole object in the hand-partitioning, although there were often candidates that one could combine to produce a whole object. Failure to recognise many of the objects can be attributed to the fact that the C program did most data-sharing through parameter-passing rather than global variables. The C++ program has objects corresponding to various geometric primitives: plane, polygon, sphere. The tool has a tendency to cluster similar messages to these objects, for example, it clusters the procedures `plane_intersect()`, `polygon_intersect()`, and `sphere_intersect()`, rather than find the objects themselves. Each of the geometric objects is in a separate file in the C program; this suggests

that if two procedures share a file, we should use this fact itself as one of the clues that they should be in the same object. The fact that we have so easily out-performed the hand-partitioning, often even with random clustering, suggests that the OO metrics should be made sensitive to the fact that if the same data or variables are passed in multiple messages, it is better that they should be multiple messages to one object than messages to different objects.

8.2.4 RL

	CBO	RFC	LCOMH	NMN	PG	
By hand	0.085	9.308	0.775	21.538	1.000	1.000
AGGLOM(CBO)	0.029	7.500	0.792	2.222	0.393	0.181
AGGLOM(RFC)	0.034	4.567	0.886	0.367	0.503	0.122
AGGLOM(LCOMH)	0.034	4.567	0.886	0.367	0.503	0.122
AGGLOM(NMN)	0.032	4.607	0.799	-0.107	0.465	0.116
AGGLOM(combined)	0.034	4.567	0.886	0.367	0.503	0.122
DIAG	0.052	7.789	0.755	-7.000	0.273	0.141
BASILI(CBO)	0.021	6.769	0.933	-0.154	0.250	0.375
BASILI(RFC)	0.037	3.055	1.000	0.000	0.439	0.084
BASILI(LCOMH)	0.026	3.917	0.796	-0.861	0.376	0.106
BASILI(NMN)	0.026	4.281	0.901	-2.406	0.372	0.116
BASILI(combined)	0.027	3.564	0.888	-0.462	0.380	0.101
BestRand(CBO)	0.045	28.250	0.963	34.750	0.190	0.231
BestRand(RFC)	0.205	2.891	0.770	0.618	0.230	0.017
BestRand(LCOMH)	0.205	2.891	0.770	0.618	0.230	0.017
BestRand(NMN)	0.214	3.077	0.994	0.346	0.229	0.023
BestRand(PG)	0.046	32.333	0.944	99.000	0.214	0.342

BASILI(CBO) does best on CBO and on the pairwise goodness; a random partitioning, followed by BASILI(RFC), does best on RFC; DIAG does best on LCOMH and NMN. All the non-random techniques outperform the hand-partitioning on three of the metrics, but only DIAG outperforms the hand-partitioning on LCOMH. In the hand-partitioning, most of the objects contain many procedures and hardly any variables, so the difficulty facing a tool that clusters by variables in coming close to the hand-partitioning is understandable. In this regard it is surprising that DIAG, which does not use information about which procedures call which, does as well as it does. For none of the techniques are most of the candidate objects recognisable as objects in the hand-partitioning; the major success of BASILI(CBO) seems to be its identification of the "rule" object. Many of the techniques produce an object clustered

based on a global variable `i`, which in fact is used only as a loop counter; this suggests the desirability of a tool's inspecting global variables to make sure they actually represent global data.

8.2.5 MMS

For MMS, only results using DIAG are described, since MMS's size proved beyond the limits of practical computability for the other two techniques as implemented: AGGLOM and Basili's method both ran for several hours with MMS as input without getting anywhere; it seems that AGGLOM was not designed to handle such a large number of attributes (170). An attempt was made to apply the software to an even larger X-windows-based application, but this caused the maximum number of fields allowed by AWK to be exceeded, and would have required a complete rewrite of the software. Nor are metric values given, since it was not feasible in AWK to do the random trials with the requisite number of objects required for normalization.

Although DIAG produced many candidate objects consisting of just one or two procedures and one or two variables, several larger candidate objects deemed to be of OO value were found, including one representing the whole canvas on the workstation screen, one representing a character input/output device, one representing an atom, one representing a distance between two atoms, and one representing an atom surface.

Chapter 9

Conclusions

9.1 Conclusions from Experimental Results

In Chapter 6, we expressed the desire for surety that our tool was actually doing something purposeful. To ascertain whether it is, we here summarize its performance versus that of the random clustering. In the following table, "yes" indicates a higher score on that metric than the best random score by at least one technique; "no" indicates that the random score was best:

	CBO	RFC	LCOMH	NMN	PG
ARGOT	yes	yes	yes	yes	yes
DIAG	yes	no	yes	no	
RAY	yes	yes	yes	yes	yes
RL	yes	no	no	yes	yes

Doing better than the random clusterer in 79% of these combinations is not poor performance, considering the hundreds of random trials involved.

If, on the other hand, we ask whether the *majority* of techniques did better than the best random score on each metric, the performance of our tool looks less impressive:

	CBO	RFC	LCOMH	NMN	PG
ARGOT	yes	no	no	yes	yes
DIAG	yes	no	yes	no	
RAY	yes	no	no	no	yes
RL	yes	no	no	yes	no

Of course, this may merely show that *some* of our techniques are poor, while others may not be appropriate for all data. But the patterns here point to some possible new approaches. For one, since the random partitionings seem quite good at scoring well on a *particular* metric (usually at the expense of other metrics, although sometimes, as with BestRand(RFC) for RL,

the random partitioning "gets lucky" and does well on multiple metrics), we might look at the best random score on combinations of metrics, or even have a random clusterer *guided* by combinations of metrics — this might do well across the board. For another, since we outperform the random clusterings on CBO with far more consistency than on the other metrics, we might try first to optimize CBO and then to investigate only transformations that do not increase coupling but that score better on the other metrics.

Our tool usually outperforms the hand-partitioning. The following table summarizes whether the majority of techniques outperformed the hand-partitioning on that metric:

	CBO	RFC	LCOMH	NMN
ARGOT	yes	yes	yes	yes
RAY	yes	yes	yes	yes
RL	yes	yes	no	yes

For all three programs, there was at least one technique that outperformed the hand-partitioning on *all* metrics. If the goal is to duplicate the hand-partitioning, then we can conclude that no combination of the metrics that we have used will do the job, and that metrics incorporating other information are needed. From the manual inspection of the code described in Chapter 8, we *have* found that outperforming the hand-partitioning on the metrics usually points to inadequacies in the metrics rather than to having produced a mapping that is "better" from an OO point of view.

And we have seen with ARGOT, DIAG, and MMS that our tool can indeed find candidate objects that are intuitively OO objects, although at present it does not always do so. In Chapter 1, we said that we would either find good objects or find deficiencies in the OO metrics, and we have done both.

Although our random clusterings show that it is fairly easy to optimize the value of a particular metric, the random results do not begin to exhaust the space of candidate objects and

do not duplicate the results of the intelligent strategies. The largeness of the space of candidate objects is itself justification of our quest for an intelligent search strategy rather than a brute-force technique.

In general, BASILI has proved the best of our three clustering techniques; but the other two have shown themselves not to be completely useless, and we might look at ways of combining their advantages.

We have achieved generally better results with the NMN metric than with the other metrics, and even than with the other three metrics combined. As we saw in Chapter 6, NMN is an improvement on LCOM because of the latter's frequent zero values for objects of interest; from our experimental results, we tentatively suggest to the OO metrics community that NMN is a better refinement of LCOM than LCOMH is.

We find that although our general idea of using global data to find candidate objects was sound, our implementation assuming that global data corresponded to global variables was too restrictive. Global data need not be in global variables: they can be passed in parameters, as we have seen more than once in the C test programs. And global variables need not contain global data: as we saw with RL, they may in fact be scratch variables. Using them in this way may not be the best programming practice; but if our tool is to have practical value, then we cannot assume that the best programming strategy has been adhered to.

The coarsest way to extending the tool to use parameters would consider only the datatypes of parameters and their subtypes, as in the Types Based Object Finder of [Liu 90] that we described in Section 2.2. A more sophisticated approach would be to dynamically monitor the C code in execution and track the `malloc`'d areas the way C memory-leak-detectors do; each `malloc`'d area could be treated as one data item.

We suggest that OO metrics be refined better to take use of global data into account. As we saw with DIAG, not all global data items are equally important, and some way to weight

the influence of an item by its importance, perhaps by the amount of storage allocated for it, should be looked at.

9.2 Future Work

If further research were to be done on this project, the first priority would be to re-implement the software currently written in AWK in a compiled programming language. This would greatly speed testing, and the tool could be applied to much larger test programs and more informative results obtained. (The bottleneck in speed is not in the parsing; it is in the iteration through the possible candidate objects, whose number grows exponentially with the size of the input program.) The greater the size of the procedural program, the more likely a programmer undertaking its conversion to OO would feel the need for such a tool.

Slicing of procedures to form smaller methods would be a high priority; this could be implemented as it was in the CAESAR system [Fouqué 93].

It would be desirable to hill-climb on the metrics when we use DIAG, as we do when using the other two clustering techniques. To do this, we could exploit the fact that DIAG leaves its "best" cluster in the top-left corner. Once the boundaries of this cluster have been determined, its rows and columns would be replaced by a single row and column representing the aggregate. The replacement of aggregates by single rows and columns would be repeated until no improvement was possible according to the metric score.

To return to the analogy used when describing BASILI in Chapter 4, we have been clustering with two main "forces of attraction": functions are attracted to one another if they share variables, and variables are attracted to one another if they share procedures. We should consider as many different forms of attractive force as possible. For example, if two procedures or variables have similar names, that is a weak form of evidence that they should be together. If two procedures have the arguments with the same datatypes, that is weak evidence that they should be together. And as we found with RAY in Chapter 8, in a multi-file program,

entities' being in the same original source file should be considered an attractive force. We should be able to add new forms of evidence without having to change the algorithm that clusters (although that is not the case now with DIAG). We have been exclusively using features derived from the source code. Other possible sources are: documentation, prior knowledge about the domain of application or algorithm, an existing OO library into which the new classes should be placed, or a human expert.

Many of these attractive forces will also tend to find similar messages passed to disparate objects; but when building an OO hierarchy is added to our work, these disparate objects will tend to inherit from a single ancestor.

9.3 Relation to Machine Learning

Clustering of software components has been a topic of interest to researchers in machine learning; for example, [Jetzelsperger 93] applied conceptual clustering to organize Smalltalk methods in a library, based on Smalltalk syntactic properties such as selection, assignment, recursion, and statement structure, so that programmers could more easily locate the methods in a large library. Software metrics have also been of interest to machine learning researchers; for example, [Srinavasan 92] applied decision tree algorithms and an expert system to estimate software development times from such features as the number of lines of code, the experience of the development team, the required reliability of the software, and the choice of programming language.

The problem we have been moving towards addressing in our research is a *bias selection* problem. There are two aspects to the bias selection; one is choice of algorithm. In pursuit of our task of identifying candidate objects, we have utilized two existing clustering algorithms and invented a new one. Although we have found BASILI generally to do the best

job, this is not the case for all our programs; and if we refined the other techniques further, the instances in which the other two techniques outperform BASILI might well increase. Hence we would want to investigate ways of combining the clustering algorithms, or of choosing between them for optimum effect. The other aspect of bias selection is the choice of which metric, or which combination of metrics, to use to guide clustering. Again there was no all-out winner, so we would want to investigate ways of combining them or dynamically choosing among them.

There has been a recent surge of research on inductive systems that aims to choose the best bias for a given set of training data. In this work on bias-shifting, there are three basic approaches. One of them (exemplified by [Rendell 87]) measures various features of a training set and selects a bias. In its crudest form it selects between whole learning systems. [Brodley 95] takes a second approach, which is really just a recursive application of the first: a bias chosen based on the training set is applied to construct *part* of the final rule-set — which has the side effect of partitioning the training data into two or more subsets. With the remaining subsets of the training data and partial rule-set built up so far, another bias selection is made, which is then applied to extend the partial rule and further partition training data. The third approach, comparable to the "wrapper" approach promoted by [Kohavi 95], is not tied at all to "features" of the training data. It tries all the possibilities, using cross-validation, or something similar, to estimate which gives the best performance. The advantage of Rendell's approach is that it is the only one of the three that can give an explanation of the rules learned in terms of features of the training data; the other two build rules based on intermediate stages of computation — which in our case would be partially clustered candidate objects. However, for the bias selection we need, all three approaches would be worth investigating, since the metrics *are* applicable to partially clustered objects, and since a dynamically biased system might well outperform a fixed-bias system.

9.4 The Last Word

We have prototyped a tool that may assist programmers in converting procedural programs to OO form; developed a novel clustering technique; highlighted the applicability of other clustering techniques to the problem; and supplied evidence that currently used OO metrics could be improved by refinements that we have proposed. OO metrics and software conversion tools are both fields in which there is considerable room for further research. It is hoped that the research presented in this thesis will provide a foundation for future work in both these fields and in their interrelation.

References

[Albrecht 83] Allan J. Albrecht and John E. Gaffney, Jr. "Software function, source lines of code, and development effort prediction: a software science validation" *IEEE Transactions on Software Engineering,* vol. SE-9, no. 6, Nov. 1983, pp. 639-648

[Arnold 93] Robert S. Arnold *Software Reengineering* IEEE, 1993

[Barnes 93] G. Michael Barnes and Bradley R. Swim "Inheriting Software Metrics" *Journal of Object-Oriented Programming* Nov.-Dec. 1993, pp. 27-34

[Bieman 91] James M. Bieman Deriving measures of software reuse in object oriented systems" Colorado State University, Computer Science, technical report CS-91-112

[Bilow 93] Steven C. Bilow "Software entropy and the need for object-oriented software metrics", *Journal of Object-Oriented Programming* vol. 5. no. 8, Jan. 1993, p. 6

[Booch 91] Grady Booch *Object Oriented Design with Applications* Benjamin/Cummings. 1991

[Brodley 95] Carla E. Brodley "Recursive automatic bias selection for classifier construction" *Machine Learning*, vol. 20, no. 1-2, July-Aug. 1995, pp. 63-94

[Cardelli 85] Luca Cardelli and Peter Wegner "On Understanding Types, Data Abstraction, and Polymorphism" *ACM Computing Surveys*, vol. 17, no. 4, Dec. 1985, pp. 471-522

[Chidamber 91] Shyam R. Chidamber and Chris F. Kemerer "Towards a metrics suite for object-oriented design" *Conference proceedings: OOPSLA '91, Phoenix, Arizona, October 6-11, 1991* ACM, pp. 197-211

[Chidamber 93] Shyam R. Chidamber and Chris F. Kemerer "A metrics suite for object-oriented design", MIT Center for Information Systems Research Working Paper #249, Jan. 1993, revised July 1993

[Coad 91] Peter Coad and Edward Yourdon *Object-Oriented Design* Prentice-Hall, 1991

[Coppick 92] J. Chris Coppick and Thomas J. Cheatham "Software metrics for object-oriented systems" *Proceedings of the 20th Annual Computer Science Conference, March 3-5, 1992, Kansas, Missouri,* ACM, pp. 317-322

[Curtis 83] Bill Curtis "Software Metrics: Guest Editor's Introduction" *IEEE Transactions on Software Engineering* vol. SE-9, No. 6, November 1983, pp. 637-638

[Dietrich 89] K. W. Dietrich, Jr., L. R. Nackman, and F. Gracer "Saving a Legacy with Objects" *Object-oriented programming systems, languages, and applications: OOPSLA '89 conference proceedings, October 1-6, 1989, New Orleans, Louisiana* pp. 77-83

[Dorfman 92] Len Dorfman *Converting C to Turbo C++* Windcrest/McGraw-Hill, 1992

[Dunn 91] Michael F. Dunn and John C. Knight "Software Reuse in an Industrial Setting: A Case Study" *13th International Conference on Software Engineering, Austin, Texas, May 13-16, 1991* IEEE, pp. 329-338

[Duntemann 90] Jeff Duntemann and Chris Marinacci "New Objects for Old Structures" *Byte*, April 1990, pp. 261-266

[Enright 95] C. Enright and M. Barbeau "An Object-Oriented Re-Engineering of the Remote Procedure Call, STREAMS and Transport Layer Interface" *Canadian Conference on Electrical and Computer Engineering (8th: 1995: Montreal)*, 1995, pp. 602-605

[Fisher 92] Douglas Fisher, Ling Xu, and Nazih Zard "Ordering Effects in Clustering" *Machine Learning: Proceedings of the Ninth International Workshop (ML92)* Kaufman, 1992, pp. 163-168

[Fouqué 93] Gilles Fouqué and Stan Matwin "A case-based approach to software reuse" University of Ottawa, Computer Science, technical report TR-93-04, Feb 1993

[Gall 95] Harald Gall and René Klösch "Finding Objects in Procedural Programs: An Alternative Approach" *Proceedings: Second Working Conference on Reverse Engineering* IEEE, 1995, pp. 208-216

[Gertsberg 87] Vladimir Gertsberg *Partitioning a Logical Model in an Enterprise into Subject Databases* M.Sc. thesis (supervisor: William Armstrong), University of Alberta Department of Computing Science, 1987

[Graham 95] Ian Graham *Migrating to Object Technology* Addison-Wesley, 1995

[Grass 91] Judith E. Grass "Design archaeology for object-oriented redesign in C++" *Technology of Object-Oriented Languages and Systems, TOOLS 5: Proceedings of the Fifth International Conference, TOOLS, Santa Barbara, 1991* Prentice Hall, pp. 365-368

[Gurewich 94] Nathan Gurewich and Ori Gurewich *Mastering C++: From C to C++ in 2 Weeks* Sybex, 1994

[Haaland 92] Kevin G. Haaland *Towards Metrics for Object Oriented Languages* M.Sc. thesis (supervisor: John Pugh), Carleton University School of Computer Science, April 1992

[Halstead 77] Maurice H. Halstead *Elements of Software Science* Elsevier, 1977

[Harrington 87] Steven Harrington *Computer Graphics: A Programming Approach* McGraw-Hill, 2nd ed., 1987

[Henderson-Sellers 91] Brian Henderson-Sellers "Some metrics for object-oriented software engineering" *Technology of Object-Oriented Languages and Systems TOOLS 6 : Proceedings of the Sixth International Conference* Prentice Hall, 1991, pp. 131-139

[Henderson-Sellers 96] Brian Henderson-Sellers *Object-Oriented Metrics: Measures of Complexity* Prentice-Hall , 1996

[Hill 74] M. O. Hill "Correspondence Analysis: A Neglected Multivariate Method" *Applied Statistics*, 1974, vol. 23, no. 3, pp. 340-354

[Hutchens 85] David H. Hutchens and Victor R. Basili "System structure analysis: clustering with data bindings" *IEEE Transactions on Software Engineering*, vol. SE-11, no. 8., Aug. 1985, pp. 749-757

[Ihme 95] Tuomas Ihme, Eila Niemelä, Marko Salmela, and Veikko Seppänen "Object-oriented re-engineering of embedded software" *Mechatronics*, vol. 5. no. 1, Feb. 1995, pp. 73-86

[Illingworth 90] Valerie Illingworth, Edward L. Glaser, and I. C. Pyle *Dictionary of Computing* Oxford, 3rd ed., 1990

[Jacobson 91] Ivar Jacobson and Fredrik Lindström "Re-engineering of old systems to an object-oriented architecture" *Conference proceedings: OOPSLA '91, Phoenix, Arizona, October 6-11, 1991*, pp. 340-354

[Jamsa 93] Kris Jamsa *Rescued by C++* Jamsa Press, 1993

[Jetzelsperger 93] Rudi Jetzelsperger, Stan Matwin, and Franz Oppacher "Enhancing reuse of Smalltalk methods by conceptual clustering", *Fifth International Conference on Tools with Artificial Intelligence: TAI '93 : Proceedings, November 8-11, 1993, Boston, Massachusetts* IEEE, pp. 108-11

[Joiner 96] Jay K. Joiner and Wai-tek Tsai "Re-engineering Legacy Cobol Programs", to appear in *Communications of the ACM*; <http://kirk.usafa.af.mil/papers/joiner/cbol-doc.mcw>

[Karunanithi 93] Santhi Karunanithi and James M. Bieman "Candidate reuse metrics for object oriented and Ada software" *Proceedings / First International Software Metrics Symposium, May 21-22, 1993, Baltimore, Maryland*, pp. 120-128

[King 80] John Russell King "Machine-component grouping in production flow analysis: an approach using a rank order clustering algorithm" *International Journal of Production Research*, vol. 18, no. 2, 1980, pp. 213-232

[King 82] John Russell King and Vizes Nakornchai "Machine-component group formation in group technology: review and extension" *International Journal of Production Research*, vol. 20, no. 2, 1982, pp. 117-133

[Kohavi 95] Ron Kohavi and Dan Sommerfield "Feature subset selection using the wrapper method: overfitting and dynamic search space topology" *Proceedings of First International Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Que., Canada, 20-21 Aug. 1995*, AAAI, pp. 192-197

[Kolewe 93] Ralph Kolewe "Metrics in Object-Oriented Design and Programming" *Software Development*, vol. 1, no. 4., Oct. 1993, pp. 53-62

[Kristensen 91] Bent Bruun Kristensen, Ole Lehrman *Object Oriented Programming in the Beta Programming Language* draft, 22 September 1991, Trykkested Matematisk Institut, Aarhus Universitet

[Lake 92] Al Lake and Curtis Cook "A software complexity metric for C++", Oregon State University, Computer Science Dept., technical report 92-60-03, 1992

[Lano 92] K. Lano and H. Haughton "Extracting design and functionality from code", *Fifth International Workshop on Computer-Aided Software Engineering : proceedings, Montreal, Quebec, Canada, July 6-10, 1992*, IEEE, pp. 74-82

[Li 93] Wei Li and Sally Henry "Object-oriented metrics that predict maintainability" *Journal of Systems and Software* 23(2), pp. 117-122, 1993

[Li 95] Wei Li, Sally Henry, Dennis Kafura, and Robert Schulman "Measuring object-oriented design" *Journal of Object-Oriented Programming*, vol. 8, no. 4, July/August 1995, pp. 48-55

[Lieberherr 90] Karl J. Lieberherr, Paul Bergstein and Ignacio Silva Lepe "Abstraction of object-oriented data models" *Proceedings of International Conference on Entity-Relationship* (Lausanne, Switzerland), 1990, pp. 81-94

[Lieberherr 91] Karl J. Lieberherr, Paul Bergstein and Ignacio Silva-Lepe "From objects to classes: algorithms for optimal object-oriented design" *Software Engineering Journal*, vol. 6, no. 4, July 1991, pp. 205-228

[Liu 90] Sying-Syang Liu and Norman Wilde "Identifying objects in a conventional procedural language: an example of data design recovery" *Proceedings / Conference on Software Maintenance 1990, November 26-29, 1990, San Diego, CA*, IEEE, pp. 266-271

[Lorenz 94] Mark Lorenz and Jeff Kidd *Object-Oriented Software Metrics: A Practical Guide* Prentice-Hall, 1994

[McCabe 76] Thomas J. McCabe "A complexity measure" *IEEE Transactions on software engineering*, vol. SE-2, no. 4, Dec. 1976, pp. 308-320

[McFall 92] Donald McFall, Gillian Sleith, and John MacRae "Towards an Object Oriented Representation of Structured Code" *Colloquium on Software Engineering and AI (Artificial Intelligence)* London: Institution of Electrical Engineers, 1992, pp. 86-92.

[McFall 93] Donald McFall, Gillian Sleith, and John Hughes "Reverse Engineering Structure Code to an Object-Oriented Representation" *Proceedings, Fifth International Conference on Software Engineering and Knowledge Engineering* IEEE, 1993, pp. 86-93.

[Morris 89] Kenneth L. Morris *Metrics for Object-Oriented Software Development Environments* M.Sc. thesis (supervisor: Chris Kemerer), MIT Sloan School of Management, May 1989

[Murray 93] Robert B. Murray "Moving your project to C++" Chapter 11 in Robert B. Murray *C++ Strategies and Tactics* Addison-Wesley, 1993, pp. 265-273

[Newcomb 95] Philip Newcomb and Gordon Kotik "Reengineering Procedural into Object-Oriented Systems" *Proceedings: Second Working Conference on Reverse Engineering* IEEE, 1995, pp. 237-249

[Ong 93] Chap-liong Ong and Wai-tek Tsai "Class and object extraction from imperative code" *Journal of Object-Oriented Programming* vol. 6, no. 1, March-April 1993, pp. 58-68

[Perry 92] Greg Perry *Moving From C to C++: The Ins and Outs of Object-Oriented Programming* Sams Publishing, 1992

[Pohl 93] Ira Pohl *C++ for C Programmers* 2nd ed., Benjamin/Cummings, 1993

[Provost 94] Foster John Provost and Dan Hennessy "Distributed Learning: Scaling Up with Coarse-grained Parallelism" *ISMB-94: Proceedings, Second International Conference on Intelligent Systems for Molecular Biology* AAAI Press, 1994, pp. 340-347

[Ranade 92] Jay Ranade and Saba Zamir *C++ Primer for C Programmers* McGraw-Hill, 1992

[Reiss 91] Steven P. Reiss "Tools for object-oriented redesign" *Technology of Object-Oriented Languages and Systems, TOOLS 5: Proceedings of the Fifth International Conference, TOOLS, Santa Barbara, 1991* Prentice Hall, pp. 361-363

[Rendell 87] Larry Rendell, Raj Seshu, and David Tcheng "More robust concept learning using dynamically-variable bias" *Proceedings of the Fourth International Workshop on Machine Learning, Irvine, CA, USA, 22-25 June 1987*, Kaufmann, pp. 66-78

[Roberts 92] Teri Roberts "Workshop report — Metrics for object-oriented software development" *OOPSLA '92 Addendum to the Proceedings* (Vancouver, Canada), Oct. 1992, pp. 97-100

[Rubey 68] Raymond J. Rubey and R. D. Hartwick "Quantitative measures of program quality" *Proceedings of the ACM National Conference*, 1968, pp. 671-677

[Shaefer 87] Craig G. Shaefer "The ARGOT strategy: adaptive representation genetic optimizer technique" *Genetic algorithms and their applications : proceedings of the second International Conference on Genetic Algorithms : July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*, Lawrence Erlbaum Associates, pp. 50-58

[Sheetz 91] Steven D Sheetz, David P. Tegarden, and David E. Monarchi "Measuring Object-Oriented System Complexity" *Proceedings of the First Workshop on Information Technologies and Systems*, Dec. 1991, pp. 285-307

[Silva-Lepe 93] Ignacio Silva-Lepe "An empirical method for identifying objects and their responsibilities in a procedural program" *Technology of Object-Oriented Languages and Systems, TOOLS 10: Proceedings of the Tenth International Conference TOOLS Europe '93, Versailles, France*, Prentice-Hall, 1993, pp. 136-149

[Slabbinck 90] Eric Slabbinck, Steven Warmoes and Peter Van Damme "Porting from specialised tools, a case study: conversion of an expert system written in a rule-based shell to an Object-Oriented program in C++" *Expert Systems Integration. Proceedings of the BANKAI Workshop, Brussels, Belgium, 12-14 Sept. 1990* North-Holland, 1991, pp. 123-131

[Sneed 95] Harry M. Sneed and Erika Nyáry "Extracting Object-Oriented Specification from Procedurally Oriented Programs" *Proceedings: Second Working Conference on Reverse Engineering* IEEE, 1995, pp. 217-226

[Solnon 92a] Christine Solnon and Michel Rueher "Inference of inheritance relationships from Prolog programs: a system developed with PrologIII" *Programming Language Implementation and Logic Programming: 4th International Symposium, PLILP '92, Leuven, Belgium, August 26-28, 1992: Proceedings (Leuven, Belgium)*, Springer-Verlag, pp. 489-490

[Solnon 92b] Christine Solnon and Michel Rueher "From a Prolog prototype to an object oriented model: an approach using relationships between types" University of Nice Sophia Antipolis - CNTS, research report RR 92-21

[Solnon 93] Christine Solnon and Michel Rueher "Extracting inheritance hierarchies from Prolog programs: a system based on the inference of type relations" *Logic Programming and Automated Reasoning: 4th International Conference, LPAR '93, St. Petersburg, Russia, July 13-20, 1993: Proceedings* Springer-Verlag, pp. 309-320

[Srinivasan 92] Krishnamoorthy Srinivasan and Douglas Fisher "Machine learning approaches to estimating software development time" Vanderbilt University, Computer Science, technical report CS-92-09, 1992

[Taylor 93] David A. Taylor "Finding Good Objects" *Object Magazine* vol. 3. no. 3 Sep.-Oct. 1993, p. 16

[Tegarden 92a] David P. Tegarden and Steven D. Sheetz "Effectiveness of Traditional Software Metrics for Object-Oriented Systems" *Proceedings of the Twenty-fifth Hawaii International Conference on System Sciences*, Jan. 1992, pp. 359-368

[Tegarden 92b] David P. Tegarden and Steven D. Sheetz "Object-oriented system complexity: an integrated model of structure and perceptions" presented at OOPSLA '92 Workshop for Metrics on Object-Oriented Software Development

[Tegarden 93] David P. Tegarden, Steven D. Sheetz and David E. Monarchi "A Software Complexity Model of Object-Oriented Systems" *Decision Support Systems: The International Journal*, Jan. 1993, pp. 241-262

[Tomic 94] Marijana Tomic "A Possible Approach to Object-Oriented Reengineering of Cobol Programs" *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 2, Apr. 1994, pp. 29-34

[Traister 93] Robert J. Traister *Going from C to C++* Academic Press, 1993

[Wirth 76] Niklaus Wirth *Algorithms + Data Structures=Programs* Prentice-Hall, 1976

[Yeh 95] Alexander S. Yeh, David R. Harris, and Howard B. Reubenstein "Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language" *Proceedings: Second Working Conference on Reverse Engineering* IEEE, 1995, pp. 227-236