Using Clustering Techniques to Guide Refactoring of Object-Oriented Classes

by

Keith Cassell

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2012

// Abstract

Much of the cost of software development is maintenance. Well structured software tends to be cheaper to maintain than poorly structured software, because it is easier to analyze and modify. The research described in this thesis concentrates on determining how to improve the structure of object-oriented classes, the fundamental unit of organization for object-oriented programs.

Some refactoring tools can mechanically restructure object-oriented classes, given the appropriate inputs regarding what attributes and methods belong in the revised classes. We address the research question of determining what belongs in those classes, i.e., determining which methods and attributes most belong together and how those methods and attributes can be organized into classes. Clustering techniques can be useful for grouping entities that belong together; however, doing so requires matching an appropriate algorithm to the domain task and choosing appropriate inputs.

This thesis identifies clustering techniques suitable for determining the redistribution of existing attributes and methods among object-oriented classes, and discusses the strengths and weaknesses of these techniques. It then describes experiments using these techniques as the basis for refactoring open source Java classes and the changes in the class quality metrics that resulted. Based on these results and on others reported in the literature, it recommends particular clustering techniques for particular refactoring problems.

These clustering techniques have been incorporated into an open source refactoring tool that provides low-cost assistance to programmers maintaining object-oriented classes. Such maintenance can reduce the total cost of software development.

Acknowledgments

I thank all of the people who have been supportive during my PhD study. I especially want to thank my advisors, Peter Andreae and Lindsay Groves, who have been extremely helpful in getting me to think and write like an academic. They have altered my way of looking at things, for the better. Similarly, I appreciate James Noble's assistance in helping me to learn to see past peripheral issues to get at the essence of the problem.

Finally, I would like to thank Sally. Not too many people are lucky enough to have a partner who is willing to sell her house, leave her family, travel across the world, and get a job, so her husband can be student. I am blessed.

Contents

1 Introduction **1**
1.1 Motivation and context 1
1.2 Objective 4
1.3 Approach 5
1.3.1 Identifying problematic classes 6
1.3.2 Proposing refactorings 8
1.3.3 Performing refactorings 9
1.3.4 Evaluating results 9
1.3.5 Visualizing refactoring 9
1.4 Contributions 10
1.5 Organization of the thesis 10

2 Evaluating the Quality of Classes **12**
2.1 Background – object-oriented software metrics 13
2.1.1 Class size 14
2.1.2 Intraclass complexity – WMC 14
2.1.3 Intraclass complexity – cohesion 14
2.1.4 Interclass complexity – coupling 24
2.2 Related work – detecting bad smells 25
2.2.1 Detecting feature envy 26
2.2.2 Identifying large (god) classes 26
2.2.3 Determining thresholds of smell detection 27
2.3 Analysis of existing cohesion metrics – sources of failure 28
2.3.1 Example noncohesive classes – PersonCars 29
2.3.2 Illusory cohesion 31
2.3.3 Hidden cohesion 33
2.4 A technique for improving structural cohesion metrics 40

2.4.1 Approach – restructuring graphs 40
2.4.2 Cohesion metrics – modified implementations 41
2.4.3 Cohesion metrics – input restructuring experiments 43
2.4.4 Cohesion experiments – conclusions 47
2.5 Metrics and refactoring 48
2.5.1 Metrics as quality measurements 48
2.5.2 Metrics as hint providers for clustering 49
2.6 Summary of contributions 50

3 Background – Refactoring and Clustering 51
3.1 Refactoring 51
3.1.1 Terminology 52
3.1.2 Refactoring using automated tools 53
3.1.3 Refactoring class structures 54
3.2 Clustering 59
3.2.1 Refactoring based on clustering 62
3.2.2 Evaluating the results of refactoring based on clustering . . 64
3.3 Summary 66

4 The Refactoring Environment 67
4.1 Identifying problematic classes 69
4.2 Visualizing classes 70
4.3 Proposing refactorings 71
4.4 Performing refactorings 72
4.5 Contribution Summary 74

5 Refactoring Using Distance-Based Clustering Techniques 75
5.1 Background – distance-based clustering 75
5.1.1 Entity representation 76
5.1.2 Similarity and distance functions 77
5.1.3 Agglomerative clustering 78
5.1.4 Partitional clustering 82
5.2 Related work – applying distance-based clustering to software . . . 84
5.2.1 Modularization 85
5.2.2 Moving attributes and methods 87
5.2.3 Extracting classes 91
5.2.4 Property set summary 94

5.3 Test suite experiments . . . 96
5.3.1 Refactoring based on structure . . . 98
5.3.2 Refactoring based on semantics . . . 103
5.4 Open source studies . . . 107
5.5 Visualizing agglomerative clustering . . . 114
5.6 Evaluation of distance-based techniques . . . 118
5.6.1 Evaluation of agglomerative techniques . . . 119
5.6.2 Evaluation of k-means and k-medoids . . . 122
5.7 Contribution summary . . . 122

6 Refactoring Using Graph-Based Clustering Techniques 124
6.1 Background – graph-based clustering . . . 125
6.1.1 Splitting a minimum spanning tree . . . 125
6.1.2 Betweenness clustering . . . 126
6.1.3 Max flow/min cut . . . 128
6.2 Related work – applying graph-based clustering to software . . . 129
6.2.1 Modularization . . . 129
6.2.2 Extracting classes . . . 130
6.3 Experiments – applying betweenness clustering to Java classes . . . 131
6.3.1 Visualizing betweenness clustering . . . 132
6.3.2 Refactoring god classes using betweenness clustering . . . 137
6.3.3 Using betweenness clustering on directed graphs . . . 146
6.4 Evaluation of graph-based clustering techniques . . . 146
6.5 Contribution summary . . . 148

7 Refactoring Using Multiple Clustering Techniques 150
7.1 Related work . . . 150
7.2 Refactoring god classes using dual clustering . . . 152
7.2.1 Dual clustering approach . . . 152
7.2.2 Case study . . . 153
7.2.3 Experiments on open source god classes . . . 158
7.3 Evaluation of refactoring using dual clustering . . . 162
7.4 Contribution summary . . . 162

8 Conclusions and Future Work 163
8.1 Contribution review . . . 163
8.2 Contribution summary . . . 167

8.3 Future work . . . 167
8.3.1 Cohesion metrics – comparative study and sensitivity analysis 168
8.3.2 Handling inheritance . . . 168
8.3.3 Additional applications of clustering algorithms . . . 169
8.3.4 Adding domain knowledge . . . 172
8.4 Conclusions . . . 174

Appendices **176**

A Test Suite Source Code **177**
A.1 AnonymousPersistence . . . 177
A.2 PersonCarDisjoint . . . 182
A.3 PersonCarDirect . . . 185
A.4 PersonCarIndirect . . . 188
A.5 PersonCarSpecial . . . 191

B Open Source Repositories **196**
B.1 Refactoring environment . . . 196
B.1.1 Plug-ins . . . 196
B.1.2 Libraries . . . 197
B.2 Open source test classes . . . 198
B.2.1 Qualitas Corpus . . . 198
B.2.2 FreeCol . . . 198
B.2.3 Heritrix . . . 199
B.2.4 Jena . . . 200
B.2.5 JHotDraw . . . 200
B.2.6 Weka . . . 200

C Experimental Data **202**
C.1 Preferences . . . 202
C.2 Agglomerative clustering . . . 202
C.2.1 Cluster counts . . . 203
C.2.2 Cluster sizes . . . 206
C.3 Betweenness clustering . . . 220
C.3.1 Cluster sizes . . . 220
C.3.2 Non-cohesion metrics . . . 220
C.3.3 Cohesion metrics . . . 224

C.4 Dual clustering 228
C.4.1 Preferences 228
C.4.2 Non-cohesion metrics 228
C.4.3 Cohesion metrics 232

D List of Cohesion Metrics 237

List of Figures

1.1 Extracting a class using an IDE 4
1.2 Locating classes using metrics 6
1.3 Visualizing a class using a graph 7
1.4 Identifying clusters of members 8

2.1 MetricDiscriminator source code 16
2.2 MetricDiscriminator intraclass relationships 16
2.3 Cohesion graph variations 20
2.4 PersonCar variations 30
2.5 Constructor removal in XMLElement 32
2.6 Example factory method 32
2.7 Example use of an event listener 37
2.8 Example use of reflection 38
2.9 Reflection in DataGenerator (*TipText methods) 39

3.1 Client with original PersonCarDirect 55
3.2 Original AutoRegistration source code 56
3.3 AutoRegistration.registerCar after a loose Move Method 56
3.4 PersonCar.setVin after a strict Move Method 57
3.5 AutoRegistration source code after loose refactoring 58
3.6 Client with PersonCarDirect after loose refactoring 58
3.7 Client with PersonCarDirect after strict refactoring 60
3.8 PersonCar.setVin after strict refactoring 60
3.9 Clustering of mammals 61

4.1 The ExtC refactoring environment 68
4.2 ExtC metrics view 69
4.3 ExtC graph view 71

4.4 ExtC agglomeration view 72

5.1 Agglomerative clustering 80
5.2 Spatial clusters 84
5.3 saveToDB method 86
5.4 AnonymousPersistence intraclass dependency graph 97
5.5 PersonCarDisjoint intraclass dependency graph 98
5.6 Clusters produced – JDeodorant's 2011 distance function (single link) 100
5.7 Agglomerative Clustering – SIM01 102
5.8 Single link agglomerative clustering of PersonCarSpecial (structure) 104
5.9 Agglomerative clustering of PersonCarDisjoint (semantics) 106
5.10 Agglomerative clustering of AnonymousPersistence (semantics) . . 107
5.11 CommandLine intraclass dependency graph 109
5.12 Single link clustering of CommandLine (Sim01) 110
5.13 Single link clustering of CommandLine (Nhood) 111
5.14 Agglomerative clustering view 115
5.15 Agglomerative clustering visualization 117
5.16 RegOptimizer dendrogram 120

6.1 Minimal spanning tree – mammals 126
6.2 Betweenness clustering – edge betweenness values 127
6.3 Minimum cuts 128
6.4 Betweenness view 132
6.5 Betweenness clustering – PersonCarDirect before start 133
6.6 Betweenness clustering – PersonCarDirect iterations 1 and 2 134
6.7 Betweenness clustering views 136
6.8 Betweenness clustering – Weka's Script class 138
6.9 Betweenness clustering with filtering – Weka's Script class 139
6.10 Cohesion distributions 144
6.11 Betweenness clustering – XMLDocument 145
6.12 Betweenness clustering on directed graphs 147

7.1 RegOptimizer class after betweenness clustering 154
7.2 RegOptimizer class after agglomerative clustering 155
7.3 RegOptimizer class after dual clustering 156
7.4 Source code for epsilonParameterTipText and listOptions 157
7.5 Cohesion distributions 160

7.6 Average cohesion values 161

C.1 Number of clusters – agglomerative clustering, Sim01 204
C.2 Number of clusters – agglomerative clustering, Nhood 205

List of Tables

2.1 Original, non-transformed cohesion measurements 44
2.2 Weka cohesion measurement changes 46
2.3 JHotDraw cohesion measurement changes 47

5.1 Property sets used for extracting classes 95
5.2 Test classes – clustering results 99
5.3 Open source classes - median number of clusters 108
5.4 Number of open source classes with 2 clusters of over 6 members . 108
5.5 Cohesion metrics - refactored CommandLine 113

6.1 Average metric values (excluding cohesion) 142
6.2 Cohesion measurements 142

7.1 Average metric values (excluding cohesion) 159
7.2 Cohesion measurements 159

B.1 FreeCol test files 198
B.2 Heritrix test files 199
B.3 Jena test files 200
B.4 Weka test files 201

C.1 Sim01 – single link 206
C.2 Sim01 – average link 208
C.3 Sim01 – complete link 211
C.4 Nhood – single link 213
C.5 Nhood – average link 215
C.6 Nhood – complete link 218
C.7 Betweenness cluster sizes 221
C.8 Betweenness clustering – non-cohesion metric results 222

C.9 Betweenness clustering – cohesion results 225
C.10 Dual clustering – non-cohesion metric results 229
C.11 Dual clustering – cohesion results 233

D.1 Object-oriented cohesion metrics 237

Chapter 1

Introduction

1.1 Motivation and context

Software development is big business. While there is significant cost and effort spent preparing code for its release, several studies indicate that 40% - 65% of the total effort is spent maintaining software after delivery [Han93, Pre97, Som96, YL94], and the maintenance is expensive. An estimate from the mid-1990s [Sut95] put the total cost of maintaining code in the United States at approximately 70 billion dollars per year. Numbers like these motivate us to try to decrease maintenance costs.

Maintainability refers to the ease by which software can be modified [ISO06], where potential software modifications include bug fixes, feature additions, and adaptations of the software for different environments. Maintainability is affected by the software's structure. Software that is modular and simple is generally easier to analyze and maintain than software that is poorly structured or complex. A programmer should be able to analyze well-structured code, understand its purpose, and improve it in a short time.

However, not all software "improvements" are worth doing. Every software change has a cost. In addition to the cost of programming the change, there are potential additional costs when the external behavior of a software component changes, e.g., updating test suites and documentation. Furthermore, any change to software entails risks – new bugs may be introduced, or the code may become less maintainable. Due to these issues, many software businesses prefer to be conservative when making code changes [Cor03]. The perceived benefit of a change must outweigh the perceived cost and risk.

Refactoring techniques are useful for making software more maintainable, while limiting the scope of potentially costly changes. *Refactoring* refers to changes that preserve the external behavior of code while improving its internal structure [Opd92, FBB+99]. Because refactoring maintains external behavior, it lessens some of the associated costs of a code change, for example, the costs of changing client code, test suites, and documentation.

Modern integrated development environments (IDEs) [SDF+03, SFB06] generally provide tool support that helps programmers to quickly refactor their code once they have noticed a potential maintenance problem. The IDE can often determine whether a proposed refactoring is legal, determine additional code that might be affected by the change, and reliably change the specified code and its dependent code in a matter of seconds. This automated refactoring support generally reduces cost, because the programmer is freed from much of the required analysis, and reduces risk, because the automated refactoring tools have presumably undergone ample quality assurance testing prior to release.

Refactoring typically makes the code easier to understand, which facilitates future coding changes. Many people [FBB+99, Bec00, MB08a, Mar08] advocate refactoring as part of a programmer's daily coding routine. (Some go as far as to recommend that programmers "refactor mercilessly" [ACP+06]). The underlying idea is to make incremental design improvements in the form of refactoring part of an evolutionary development process. By doing so, everyday development can proceed more smoothly, and the designated maintenance phase of development can be shortened. This underlying idea is often not realized, and software maintainability problems persist into the maintenance phase of the software life cycle.

Refactoring is also useful in the maintenance phase. As with other designated maintenance activities, there are typically several steps involved – a programmer must identify a potential problem, determine how to fix it, and then implement a solution. The problem of locating maintenance problems amenable to refactoring is an ongoing effort, and numerous researchers [FBB+99, DBDV04, LM06, MB08b] have suggested ways of identifying problematic code (a.k.a. "bad smells" [FBB+99]). How to perform the second step is even less clear-cut. For some refactorings, determining how to fix the problem is trivial; for others, only high-level advice exists. The third step, implementing the solution, has been aided by a number of integrated development environments whose "automated refactoring" tools provide a mechanism for mechanically altering source code, given the

appropriate inputs. Here too, the state of the practice is varied – some refactorings are widely implemented and robust, while more complicated refactorings may not have robust implementations (or any implementation at all).

Because object-oriented programming is one of the most popular programming paradigms of today [Lan11], this thesis concentrates on refactoring techniques for object-oriented programs. One of the less mature areas in refactoring is how to reorganize the class structure of object-oriented programs. For object-oriented languages like Java, C++, and C#, the fundamental organizational units are classes, and one of the most important things for creating and maintaining code written in object-oriented languages is getting the classes "right". Yet, although there is high-level guidance on how to design good classes [CY91, Rum96, Mar08], problematic classes persist.

Much of the high-level guidance for creating good classes pertains to having clear-cut divisions of responsibilities between classes. "One class, one responsibility" is a common refrain [Mar08], and there are a variety of pejorative terms that refer to classes where the division of responsibilities is not clear-cut, e.g., "blobs" [MHVG08], "god classes" [LM06, WL08], and "spaghetti classes" [BCM+96]. All of the bad smells below indicate problems with dividing responsibilities between classes. For each of them, Fowler's book [FBB+99] provides some high level strategic advice about how class structures should be modified by refactoring, e.g.:

1. Large class – Large classes that perform too many functions can be divided into smaller ones, e.g., via the *Extract Class* refactoring (described in Section 3.1.3).
2. Data Clumps – Groups of attributes (a.k.a. fields) that repeatedly show up together in multiple classes can be grouped as the basis for a class, e.g., via the *Extract Class* refactoring.
3. Feature envy – Classes with inappropriate functionality can have that functionality moved to other classes, e.g., via the *Move Method* and *Move Field* refactorings (described in Section 3.1.3).

The common theme for these refactorings is to put closely related class members (attributes and methods) together in the same class and less closely related members[1] in different classes.

[1] This thesis uses the shorter term "members" instead of "class members" to refer to the attributes and methods of a class, when the shorter term is unlikely to cause confusion.

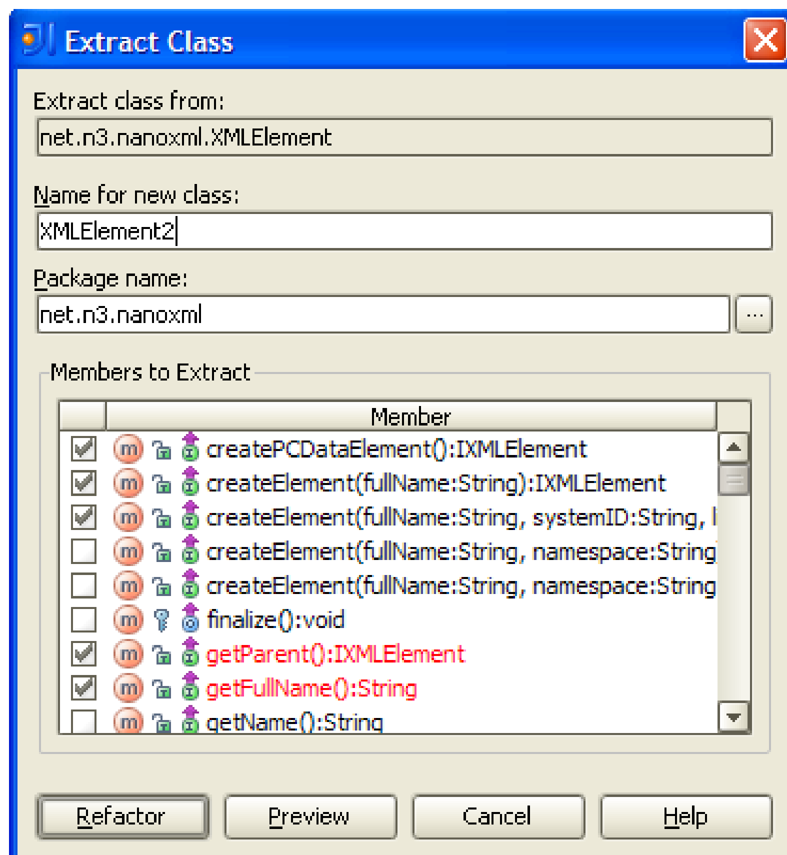


Figure 1.1: Extracting a class using an IDE

Unfortunately, such strategic refactoring advice generally lacks detail about what needs to be done to effect a solution. For example, Figure 1.1 shows how one IDE determines how to perform the *Extract Class* refactoring – by the user entering the class members that should be moved to the new class. However, it is up to the user to determine which members are to comprise the revised classes. This is difficult when a class can consist of over a hundred members.

1.2 Objective

The objective of this research is to develop techniques that can improve the maintainability of object-oriented classes by supplying the necessary detail on how the class members should be reorganized. Given such input, programmers can restructure code either using "automated" refactoring tools provided by popular IDEs or manually. From our point of view, it would be ideal if programmers

could easily identify difficult-to-maintain classes, and based on input supplied by our techniques, easily refactor those classes into ones that were less expensive to maintain.

1.3 Approach

If there were sufficient knowledge of what constitutes good class structure, restructuring classes would be relatively easy. For example, one could construct a "class structure evaluation function" and then use "generate and test" techniques to create potential class structures to maximize the result of the evaluation function [SSB06]. Unfortunately, such comprehensive knowledge about class structure does not exist, so a more exploratory approach is indicated. An exploratory refactoring environment helps researchers determine how to refactor classes based on existing ideas of what constitutes good classes, and also helps them discover additional characteristics of classes and their relationships that may be useful for refactoring.

An interactive class refactoring environment should help with each of the following tasks:

1. Identifying classes that violate guidelines concerning class structure.
2. Proposing potential refactorings that address these violations.
3. Performing the refactorings.
4. Evaluating the results of the refactorings.
5. Visualizing the classes before, during, and after refactoring.

The refactoring environment I created for my research concentrates on (2), but makes contributions to (1), (4), and (5) as well. In principle, the first four tasks could be completely automated, with the output of one task serving as the input to the next one. In practice, however, we prefer an interactive approach, where the environment provides helpful input to the programmer, rather than programmatically making decisions and executing them.

There are many reasons to prefer an assistive, interactive approach to an automated one [Cor03]. In our case, there are two main ones:

1. The state of the art for refactoring class structure is not sufficiently mature to trust automated results.

2. There are many potential programming styles, each of which has advantages in certain circumstances. A completely automated approach is unlikely to be sufficiently flexible to satisfy users.

Chapter 4 discusses the *ExtC* (*Ext*ract Class) refactoring environment I created, including details of user interfaces. The following subsections describe our approach to the five refactoring subtasks above.

1.3.1 Identifying problematic classes

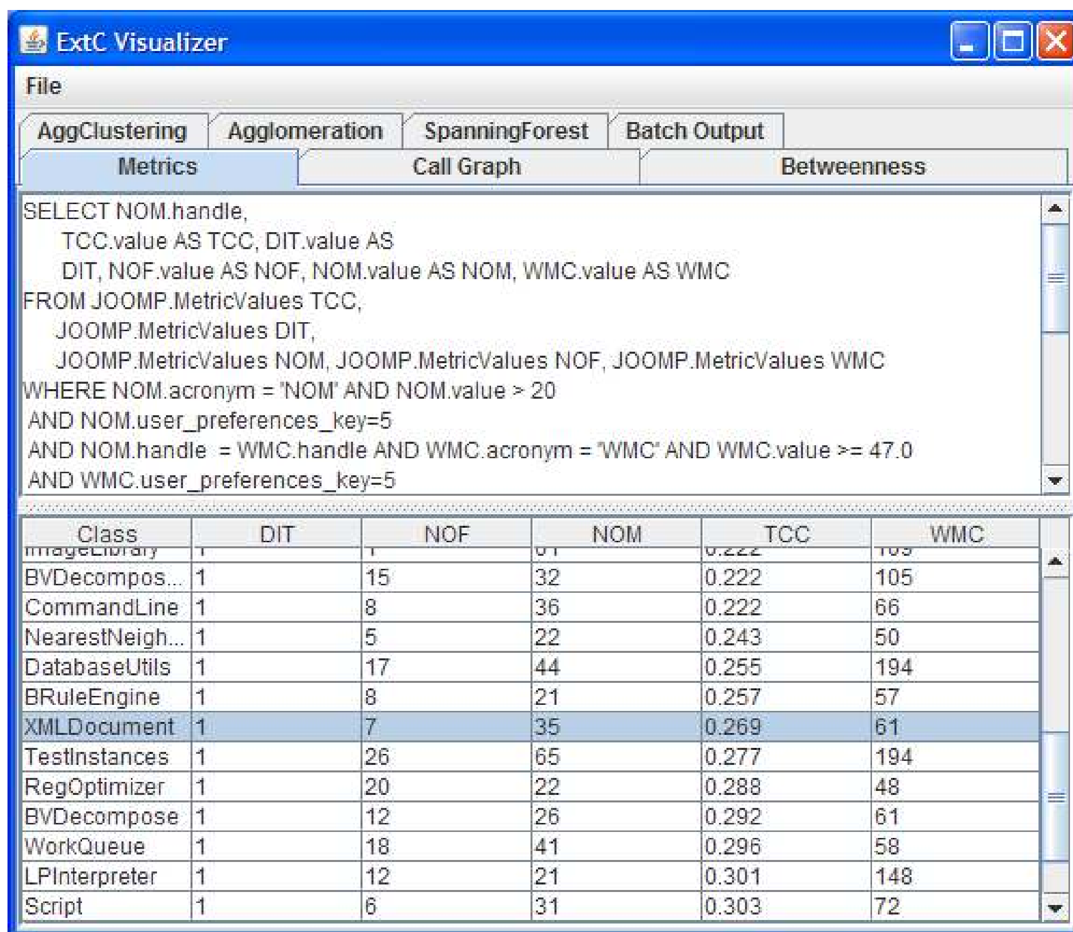


Figure 1.2: Locating classes using metrics

Object-oriented software metrics can help programmers detect poorly structured classes that are too large, too small, or have poor distribution of methods and attributes across the class hierarchy [LM06, CK94, Hen96, CLM06]. Of particular

interest are metrics that measure a class's cohesion (how well the members of a class work together) and those that measure coupling between classes (the degree to which classes rely on other classes). Both cohesion and coupling have been shown to correlate with maintainability [ASKM07, DJ03, CDK98]. In general, one wants to maximize cohesion and minimize coupling [CK94].

We provide programmers the capability of constructing metric-based queries to locate potentially troublesome classes. Figure 1.2 shows a user interface containing a user-composed query and a table of matching classes and their metric values. These classes can then be examined in a source editor or with a graph-based visualization that shows their internal structural dependencies. Figure 1.3 shows a visualization of one of the identified large classes, which shows the methods, attributes, and certain relationships between them. In this thesis, we will generally illustrate relevant points using graphs instead of the actual code. This tends to be both clearer and more concise.

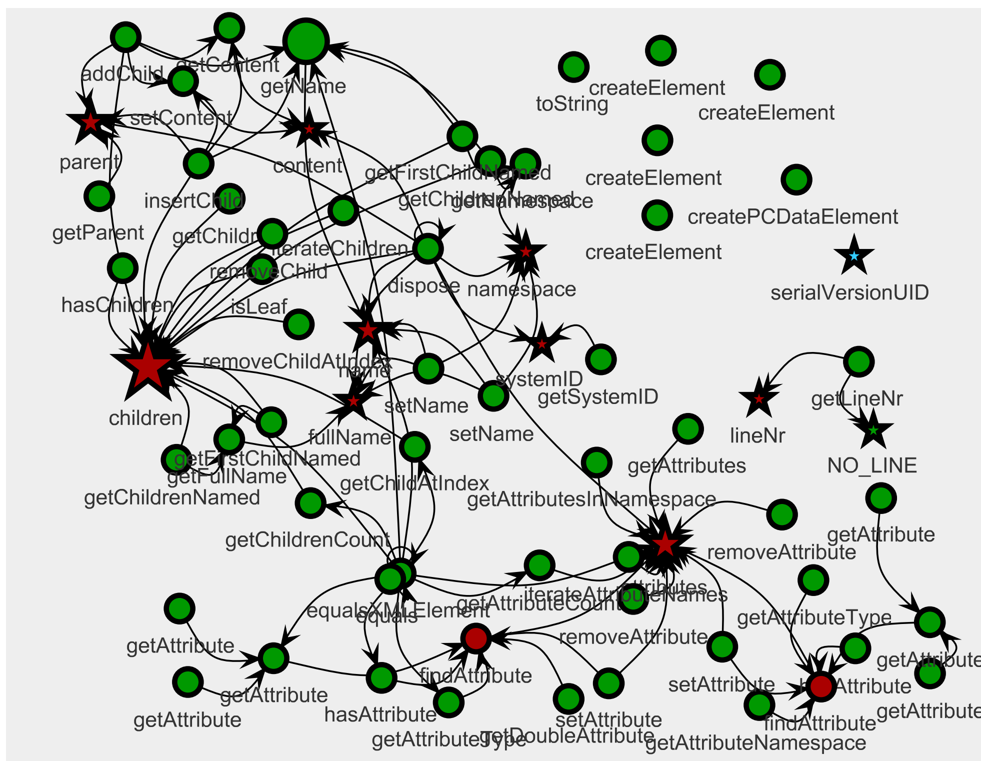


Figure 1.3: Visualizing a class using a graph

1.3.2 Proposing refactorings

This thesis explores how a variety of clustering techniques can be applied to creating higher quality classes. Clustering techniques [JMF99, Ber02, Sch07] group together related entities into clusters. In the context of correcting poorly designed classes, the entities to be clustered consist of the members of the classes. The clustering algorithms use information about the affinity of the members to determine which members most belong together in classes.

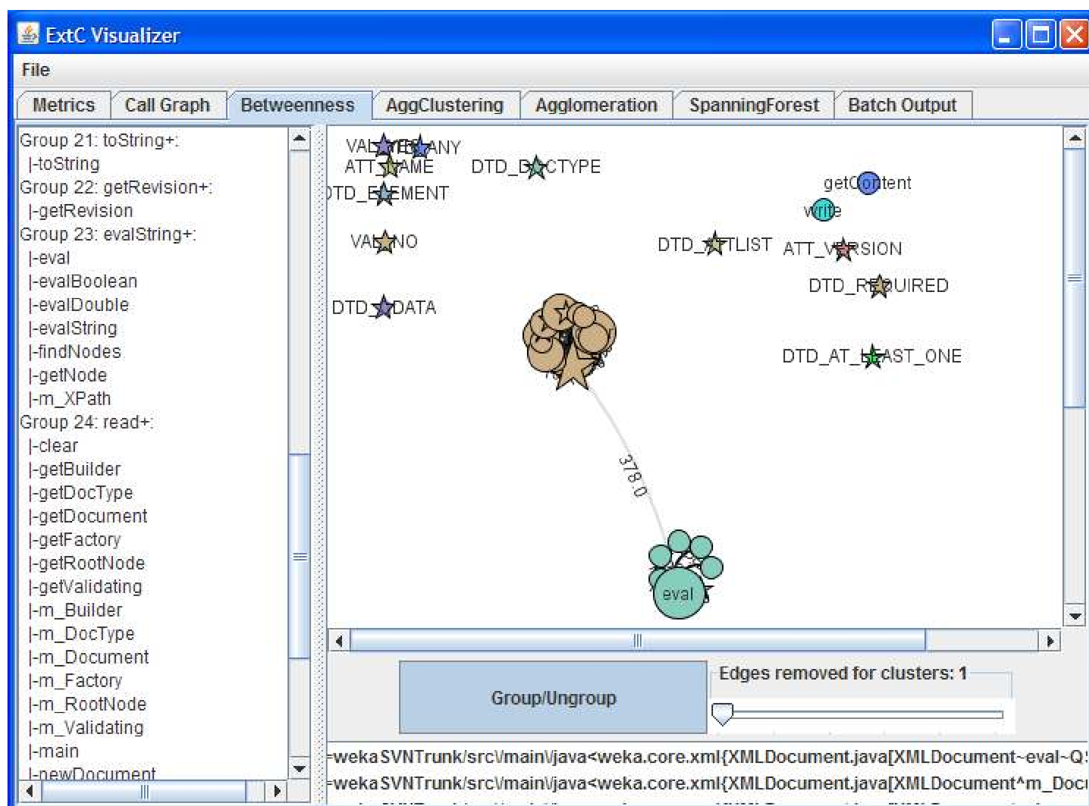


Figure 1.4: Identifying clusters of members

Some of these clustering techniques work by determining similarities between entities. Others, derived from social network analysis, form clusters based on patterns of communications between entities as shown in Figure 1.4. (Textual identification of the group members is on the left, and a graph-based visualization is on the right.) Still others work by evaluating the "semantic" content contained in the classes. The class members within the clusters produced from these techniques can serve as inputs to automated refactoring tools as shown in Figure 1.1.

1.3.3 Performing refactorings

Until relatively recently, almost all code changes were performed manually by programmers. Beginning in the 1990s, integrated development environments have been providing an increasing amount of semi-automated refactoring capability, and some of these IDEs offer application programmer interfaces (APIs) for refactoring [Wid06, Jem08]. Our research makes heavy use of existing capabilities to restructure classes, but does not implement any new refactorings.

1.3.4 Evaluating results

Evaluating the effects of refactoring on maintainability can be difficult and expensive. Because this is impractical for the scope of this thesis, we use several software metrics that may correlate with maintainability [LH93, BBD01] to evaluate the effectiveness of the refactorings.

There are some difficulties with using software metrics to measure the quality of software. While there is general agreement on some high-level principles (e.g., high cohesion and low coupling are desirable [HM95]), there are seldom specifics (e.g., what cohesion values indicate acceptable class structure). Furthermore, there are no studies that we know of that indicate acceptable trade-offs between various metric values. For example, what should happen when one metric shows improvement (e.g., cohesion increases) while another metric shows degradation (e.g., coupling also increases)?

Another possibility is to have programmers evaluate the results. This, too, is problematic. Many of the characteristics of programs that make them hard to maintain also make them hard to evaluate for improvement.

Given these difficulties, we use a multi-pronged approach to evaluating results. First, we collect and compare metric values from before and after any restructuring. Secondly, we evaluate our techniques relative to a test suite with known preferred outcomes. Finally, we provide visualizations that help programmers see the relationships between the members of classes, so that they can better compare classes before and after refactoring.

1.3.5 Visualizing refactoring

Complex object-oriented classes are difficult to analyze and understand, and therefore difficult to maintain. This same complexity also makes it difficult to see

how best to refactor them. We help with this problem by providing visualizations of the static structure of classes (as seen in Figure 1.3 and discussed in Section 4.2) and by providing visualizations of the clustering technique results (as discussed in Sections 5.5 and 6.3.1).

1.4 Contributions

The primary contribution of this research is the identification of the relative strengths and weaknesses of various clustering techniques for identifying groups of class members that can be used as the basis for refactoring object-oriented classes. To address some of the weaknesses of current approaches, we introduce two new techniques. The first, based on betweenness clustering [GN02, Bra01], determines clusters based on the communication patterns within a class. The second, dual clustering, determines clusters by sequentially using two complementary techniques – betweenness clustering based on structure and agglomerative clustering [Ber02, JMF99, WFH11] based on semantics.

In order to evaluate the use of clustering techniques in support of refactoring, it was helpful to build an exploratory environment in which problematic classes could be identified, examined, and repaired. This effort led to some additional contributions. In order to better identify problematic classes, we analyzed many cohesion metrics and their limitations. This led to the development of a technique to make certain structural cohesion metrics more accurate by eliminating specified class members and relationships from consideration, and including additional relationships in the calculations. This same technique is also useful for reducing noise in the inputs to clustering.

To help determine whether clustering was producing meaningful results, we created two novel visualizations of clustering acting upon class structure. In addition, we created a test suite of Java classes to use for the *Extract Class* refactoring. Given these classes as inputs, one can determine whether tools produce the desired recommendations for refactoring.

1.5 Organization of the thesis

The thesis is organized as follows. The next chapter, Chapter 2, discusses how to evaluate the quality of object-oriented classes and concentrates on the use of

metrics, especially cohesion metrics. It goes into some detail on some of the weaknesses of the cohesion metrics, and also discusses a structural technique we use to improve them for use with refactoring.

The next two chapters cover common material useful for understanding Chapters 5 - 7. Chapter 3 provides general background information on refactoring and clustering, while Chapter 4 describes the refactoring environment in which the clustering and refactoring investigations took place.

Chapters 5 - 7 contain the bulk of the analysis and new work. These chapters discuss the utility of various clustering techniques for refactoring object-oriented classes. Chapter 5 discusses distance-based clustering techniques, such as agglomerative clustering and k-means [Har75, HW79, Ber02, JMF99]; Chapter 6 discusses graph-based clustering techniques, such as betweenness clustering and max flow/min cut [FF09]; and Chapter 7 discusses how combining multiple clustering techniques can aid refactoring. All three of these chapters follow a similar outline. They start with a background discussion of the clustering techniques and then describe prior work on applying clustering techniques to refactoring object-oriented systems. Next is a discussion of our contributions of applying the particular clustering technique towards refactoring, and an evaluation of the various approaches. Chapter 8 contains a review of the contributions of the thesis and discusses potential future work.

Chapter 2

Evaluating the Quality of Classes

Our goal is to improve the structure of object-oriented classes to make them more maintainable. Given a sufficient description of the characteristics of good object-oriented classes, it should be possible to determine whether or not a class has those qualities.

Books often advise developers of object-oriented systems to create classes that represent single concepts [RBL+90] or have single responsibilities [Mar08], and state that the members of a class should all work well together [BME+07]. However, as Booch, et al. [BME+07] point out, it is difficult to define a class correctly the first time. Programmers can have different opinions about what constitutes a good class, because the concepts on which classes are based may be imprecise or overlapping, and there are subtleties that can affect the design, e.g., the desired granularity of the public interfaces to the class. There are additional constraints on design imposed by a desire to have maintainable classes, e.g., a class should be easy to analyze and change [ISO06, fS01].

These characteristics of a good object-oriented class are mostly based on the perceptions of the people using the class, and the degree to which a class possesses these characteristics will vary by individual. They are not easy characteristics to measure. These cognitive characteristics can be classified as *external attributes* of software, i.e., their measurement requires something in addition to the software itself. For example, measuring the analyzability of a class involves the software being analyzed, analysis tasks, and the humans performing the analysis.

Designing and executing experiments to measure cognitive abilities is difficult, due in part to the difficulty in determining the subjects' thought processes, the many variables involved, and the difficulty in creating appropriate controls for

the variables being tested. Simpler attempts to measure cognitive characteristics of software also have drawbacks. For example, while it is possible to query many programmers about each class to determine whether the class had the desired characteristics, such an approach is expensive and generally impractical, due to the many programmers that need to be involved to make the results statistically significant.

It is generally easier and cheaper to measure *internal attributes* of the software, i.e., measurements on the software itself. Some internal attributes that are frequently mentioned as desirable are small size, low coupling, low complexity, and high cohesion [CSC06, BBG08, DJ03, LH93, MPF08, ASKM07, CDK98]; however, it is worth noting that there is debate about whether these all correlate with maintainability [DJ03, EBGR01, BWZ02, BEGR00, SK03]. We use metrics based on internal attributes of the software to identify classes that may benefit from refactoring. Following refactoring, those same metrics are used to help evaluate the quality of the refactored classes relative to the original ones.

This chapter discusses how we attempt to measure the quality of object-oriented classes using internal attributes of the software. The chapter begins with a discussion of object-oriented software metrics, with an emphasis on class cohesion metrics. This is followed by a section on research that uses metrics to detect bad smells indicative of problem classes. Section 2.3 analyzes features of the Java language that can cause many cohesion metrics to give misleading results, while Section 2.4 discusses modifications we made to existing metrics to better handle those Java features, then discusses our experiments with the modified metrics to determine whether they helped to better identified flawed classes. The chapter concludes with an analysis of the use of metrics in the context of refactoring and a summary of our contributions.

2.1 Background – object-oriented software metrics

Software metrics can be used for many purposes, and particular metrics are better suited for some purposes than others. We are interested in using metrics to measure the quality of object-oriented classes – to help identify Java classes that are in need of refactoring and to help confirm that the classes' quality has improved as a result of refactoring.

This section discusses some object-oriented size and complexity metrics that

may correlate with maintainability. Many of these evolved from metrics that were used to help evaluate structured designs [YC79, SMC79] and were later adapted for the object-oriented paradigm [EKS94, CK91].

2.1.1 Class size

Intuitively, small classes should be easier to analyze and change than large classes, and several researchers have noted a correlation between class size and fault proneness [DJ03, EBGR01, LH93]. While measuring the size of software is a simple idea, there are several options of what can be measured – number of *lines of code (LOC), number of methods, number of fields*, etc. We most often use the total number of methods as a measure of an object-oriented class's size, because this gives an indication of the number of high-level functions a class supports.

2.1.2 Intraclass complexity – WMC

Simple classes should be easier to analyze and change than complex classes. The *Weighted Methods per Class* (*WMC*) metric combines both size and complexity. Chidamber and Kemerer [CK94] define the WMC as the sum of the complexities of a class's methods. They deliberately do not define what constitutes the complexity of a method.

Several researchers [BLL09, CLM06, CAG11, LM06] use WMC with cyclomatic complexity [McC76] as the method complexity metric. Cyclomatic complexity measures the amount of logical branching (number of decision points) in a method. We also use WMC in combination with cyclomatic complexity to detect large, complex classes.

2.1.3 Intraclass complexity – cohesion

In general, cohesion refers to how strongly related the members of a group are. In structured design, cohesion refers to how closely related the processing elements in a module are [YC79]. In the most cohesive modules, the processing elements all pertain to the module's single function. Slightly less desirable, but easier to recognize, are modules that are sequentially cohesive, where the output data from one processing element serves as input data for the next.

The ideas about cohesion from structured design were adapted for use in object-oriented design [EKS94, CK91]. For cohesion measurement purposes, the

classes, methods, and attributes in object-oriented design roughly correspond to the modules, processing elements, and data of structured design. Ideally, the methods and attributes in a class are closely related to each other, and this is reflected in their patterns of interaction. For object-oriented classes, the term *cohesion* refers to how well the attributes and methods of a class work together.

Because cohesion is considered integral to good object-oriented design, there have been many attempts to measure it, both from a structural and from a conceptual point of view. There are over 40 different cohesion metrics in the literature (see Appendix D). Rather than attempt a detailed review, this section highlights cohesion metrics and principles that researchers have found useful relative to refactoring. See Briand, et al. [BDW97], for a more detailed overview of the early work on measuring object-oriented cohesion.

Structural cohesion metrics

Structural cohesion metrics measure relationships between various code constructs. We are most interested in metrics that measure the relationships among attributes and methods within a single class [CK94, BT07, BDW97, CKB00, EGF^{+}04, Hen96, HM95, ZLLX04], although some metrics measure the relationships between methods and parameters [BEDL99, CSC06], and others consider relationships to members of other classes [ML09, ML07]. Intraclass structural cohesion metrics analyze the structure of a single class's code to calculate a cohesion score based on the degree to which a class's methods access the other class members (e.g., attributes and methods). This is the largest class of cohesion metrics, presumably because measurements of the existing static interactions between class members is a logical indication of how much the members belong together.

As several authors [CKB00, HM95, ZLLX04, AD10] have noted, we can interpret many structural cohesion metrics in terms of graph analysis, where nodes in the dependency graph are methods or attributes, and edges indicate a method calling another method or a method accessing an attribute. The code in Figure 2.1 is a toy `MetricDiscriminator` class that will be used to show the differences between how various structural cohesion metrics calculate cohesion. Figure 2.2 is the corresponding *intraclass dependency graph*, where circular nodes represent `MetricDiscriminator`'s methods, stars represent its attributes, and the directed edges indicate methods calling methods or accessing attributes. In Figure 2.2, there is an arrow from `m1` to `m0`, because the method `m1` calls the

```
public class MetricDiscriminator {
    private int a1, a2, a3, a4 = 4, a5;
    public int m0(int i1, int i2) { return i1 + i2; }
    public int m1() { return a1 = m0(a1, 1); }
    public int m2() { a2 = a3 = 3; return m0(a2, a3); }
    public int m3() { return a4 = a3; }
    public int m4() { return a4; }
    public int m5() { return a5; }
    public void m12() { m1(); m2(); }
}
```

Figure 2.1: MetricDiscriminator source code

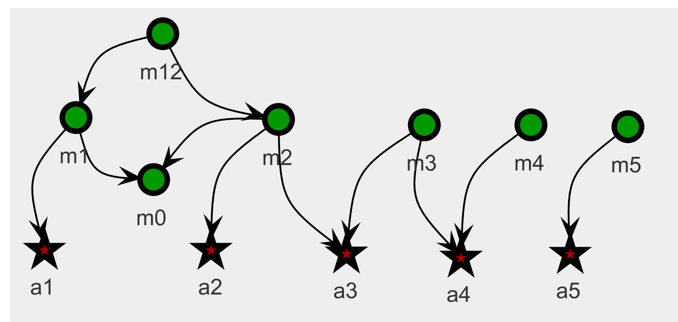


Figure 2.2: MetricDiscriminator intraclass relationships

method `m0`, and there is an arrow from `m1` to `a1`, because the method `m1` accesses the attribute `a1`.

LCOM

Chidamber and Kemmerer [CK94] thought that a well-designed class should have a set of attributes that are commonly used by the class's methods. Their *Lack of Cohesion in Methods (LCOM)* metric compares the number of dissimilar pairs of methods to the number of similar pairs of methods, where two methods are similar if they each access a common attribute directly. If there is a greater number of similar methods than dissimilar methods, then LCOM is zero. Otherwise, LCOM equals the number of dissimilar pairs of methods minus the number of similar pairs of methods. A maximally cohesive class has an LCOM score of 0, as do some

other less cohesive classes. There is no upper bound on the scores of noncohesive classes.

In Figure 2.1, methods `m2` and `m3` are considered similar because they both access attribute `a3`, whereas `m1` and `m2` would be dissimilar, because they access no common attributes. `MetricDiscriminator` has an LCOM score of 17, because (`m2, m3`) and (`m3, m4`) are similar, and the other 19 pairs of methods are dissimilar.

There are a number of problems with using LCOM to measure cohesion [HM95, BDW97, CKB00, ML06]. For example, many classes with varying degrees of cohesiveness can have an LCOM score of zero. Furthermore, LCOM is not a normalized metric. Because LCOM is measured based on the difference between the counts of dissimilar pairs of methods and similar pairs of methods, there is a correlation between the number of methods in classes and their LCOM scores. A relatively large LCOM score for a small class generally indicates a more disjointed class than that same score for a larger class. Despite its problems, LCOM has inspired a number of other cohesion metrics. In particular, many subsequent metrics also place primary importance on method-attribute interactions.

"LCOM*"

Henderson-Sellers' enhancements to LCOM [Hen96] have been referred to by different names (LCOM* [BDW98, FPn06, BT07, SB10], LCOM3 [WZW+05], LCOM5 [BDW97, AD10, CEJ06, BBG08, UFPG10], and LCOM-HS [LLL08]) by different authors. This thesis uses "LCOM*", because that is the name used by the Metrics2 plug-in [SB10] that is part of our refactoring environment.

Henderson-Sellers wanted a lack of cohesion metric that was normalized to a range of 0.0 to 1.0, where 0.0 indicates perfect cohesion (every method accesses every attribute) and 1.0 indicates total lack of cohesion. LCOM* is defined as:

$$LCOM* = \frac{\frac{1}{a}\sum_{i=1}^{a} n(A_i) - m}{1 - m} \tag{2.1}$$

where *a* is the total number of attributes, *m* is the total number of methods, and *n(A)* is the number of methods that access attribute A. The LCOM* score for `MetricDiscriminator` is 0.93.

“LCOM4”

Hitz and Montazeri’s enhancements to LCOM [HM95] have been referred to by different names (LCOM4 [BT07, CKB00, EGF+04], LCOM2 [WZW+05], and LCOM3 [BDW97]) by different authors. This thesis uses “LCOM4”, in keeping with the majority.

Hitz and Montazeri take a graph-theoretic approach to cohesion. *LCOM4* is defined as the number of connected components of an undirected intraclass dependency graph, so `MetricDiscriminator` has an LCOM4 value of 2. Among LCOM4’s enhancements to LCOM is that it considers both direct and indirect access of methods to attributes. For example, LCOM4 considers `m12` and `m3` to be similar, because they each access attribute `a3`. The `m12` method accesses `a3` indirectly through `m2`, so the original LCOM would not consider `m12` and `m3` to be similar.

There are several reasons for making indirect access from a method to an attribute within a class as important as direct access to an attribute. For example, consider a scenario where a class originally had a large `m12` that directly accessed `a3`. Later, a programmer refactors the large `m12` method, so that it calls the extracted `m2` method, which calls `a3`. The class has the same functionality as before the *Extract Method* refactoring, and `m12` is just as dependent on `a3` as it was originally.

LCOM4 is not normalized; it ranges from a maximal cohesion of one and has no upper bound on lack of cohesion. Because it is equivalent to the number of connected components of a graph, its definition of lack of cohesion appeals to those looking for classes that can be easily split.

TCC, LCC

The *Tight Class Cohesion (TCC)* and *Loose Class Cohesion (LCC)* metrics [BK95] measure the proportion of pairs of connected, visible instance methods to the maximum possible number of pairs of connected, visible instance methods, so TCC and LCC scores range from 0 (least cohesive) to 1 (most cohesive). TCC considers two methods to be connected when they both have paths to a common attribute following directed edges. Like LCOM4, TCC considers `m12` and `m3` to be connected, because they both access attribute `a3` either directly or indirectly.

If one converts a directed dependency graph to an undirected one, LCC considers two methods to be connected if there is any path between them in

the undirected dependency graph that passes through one or more attribute nodes. In our example, LCC considers m2 and m4 to be connected, because both of them are connected to m3 (via a3 and a4 respectively). The rationale behind this is that the intermediary methods can pass values indirectly to the other methods via the attributes. In the example, m2 sets the value of a3; m3 reads a3 and sets a4, and m4 reads a4, so m2 can affect the output of m4. Because this indirect connection exists, MetricDiscriminator's LCC value (0.48) is greater than its TCC value (0.24). The LCC value will always be greater than or equal to the TCC value.

DC_D, DC_I

All of the previously mentioned cohesion metrics calculate cohesion based on some notion of connectivity between methods and attributes. Badri and Badri [BB04] point out that methods can be connected via access to common methods that do not necessarily access any attribute in the class. *Degree of Cohesion Direct (DC_D)* and *Degree of Cohesion Indirect (DC_I)* are closely related to TCC and LCC respectively, but extend them, primarily by considering two methods to be connected if they access a common method either directly or indirectly, but also by considering static members. DC_D and DC_I scores range from 0 (least cohesive) to 1 (most cohesive).

DC_I extends DC_D similar to how LCC extends TCC. If one converts the directed dependency graph to an undirected one, DC_I considers public methods to be connected if there is any path between the methods in the undirected dependency graph.

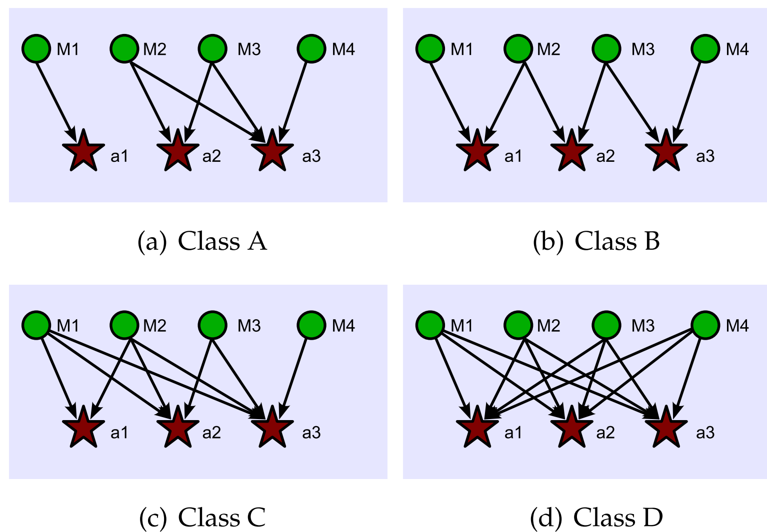
In the example of Figure 2.2, DC_D and DC_I consider the m1 and m2 methods to be connected, because they both use the m0 method. In contrast, TCC considers them to be unrelated, because they do not access a common attribute. (LCC considers m1 and m2 to be related via indirection through the method m12.) Consequently, the DC_D score (0.29) is slightly higher than the TCC score (0.24), while the DC_I and LCC scores are the same (0.48). DC_I considers the m1 and m3 methods to be connected, because both of those are connected to m2 (via m0 and a3 respectively), so the DC_I score is greater than the DC_D score. The DC_I value will always be greater than or equal to the DC_D value.

CBMC, ICBMC

The metrics mentioned thus far base their calculations on the number of interactions between methods and attributes, but do not consider the pattern

of those interactions. Chae and his colleagues' [CKB00] *Cohesion Based on Member Connectivity (CBMC)* cohesion metric considers the patterns of interaction based on a graph representation of the call pattern between methods and attributes.

They were motivated by what they perceived as deficiencies in some of the earlier approaches to measuring cohesion. Figure 2.3 illustrates the dependency graphs of four different classes, based on one of Chae's examples [CKB00]. Although all four classes have the same number of methods and attributes, Chae contends that intuitively, cohesion increases going from class `A` to class `D`. While `A` and `B` have the same number of attribute accesses, the graph of class `A` is disconnected. While the dependency graphs of classes `B`-`D` are all connected, the number of edges increases, so cohesion should also increase. However, when they calculated cohesion for these classes using nine existing metrics (including the ones mentioned above), none of the metrics gave the expected behavior.[1] Every metric showed at least two of the classes having the same cohesion value.



(a) Class A (b) Class B

(c) Class C (d) Class D

Figure 2.3: Cohesion graph variations

CBMC introduced a novel way of calculating cohesion, based on breaking apart a reference graph. Reference graphs are bipartite graphs consisting of nodes that represent methods or attributes. There is an edge between a method and an attribute if the method can access that attribute either directly or indirectly using other methods. CBMC breaks apart a reference graph by removing the nodes that

[1] Al Dallal [AD10] provides a larger set of dependency graphs that he uses to illustrate the lack of discriminatory power of 16 structural cohesion metrics.

would most quickly fragment the graph into maximally connected components (*MCCs*), where an MCC is a graph where all of the method nodes are connected to all of the attribute nodes. Figure 2.3(d) is an MCC without the removal of any nodes, so it is maximally cohesive. The mechanism for calculating CBMC is complicated and can involve evaluating multiple alternatives for fragmenting the graph, so it will not be reproduced here.

Zhou, et al. [ZXZY02] noticed that there were some cases where CBMC calculated a higher cohesion score for a graph with fewer edges than a similar graph that was more densely connected. They eliminated this problem with their Improved CBMC (ICBMC) cohesion metric, which fragments the reference graph by removing edges rather than nodes. CBMC and ICBMC scores both range from 0 (least cohesive) to 1 (most cohesive).

Chae, et al. [CKB00] also made explicit the idea of *special methods*, which are methods that should not be taken into account when calculating cohesion. Special methods are considered unrepresentative of the real purpose of the class and therefore serve as "noise" when trying to measure cohesiveness. Such methods include constructors, which access many attributes for initialization purposes, and accessors, which typically provide access to exactly one attribute but provide no behavioral logic.

Conceptual and semantic cohesion metrics

The ISO standard's [ISO06, fS01] characteristics of maintainability (analyzability, changeability, etc.) are predominately semantic criteria. "Conceptual" and "semantic" metrics attempt to measure the software based on interpretations of the programmers' intent based on word usage within the software [ED00, MP05, CEJ06, Etz06, MPF08, SDGP10, LPF+09, PM06, ED00]. Typically, these approaches analyze the words that are embedded in the source code and associated comments, looking for overlapping meaning in different parts of the software.

Etzkorn [Etz06] uses *semantic metrics* to refer to metrics such as *Logical Relatedness of Methods* (*LORM*) [ED00] and *Semantic Closeness from Disambiguity* (*SCFD*) [CEJ06], that measure software using knowledge-based natural language processing. These techniques are applied to identifiers and/or comments to see how close a class's members are to each other conceptually. Etzkorn uses *conceptual metrics* for metrics such as *Conceptual Cohesion of Classes* (*C3*) [MP05] and *Conceptual Lack of Cohesion Between Methods* (*CLCOM5*) [UFPG10], that rely on usage statistics

rather than some underlying knowledge base or dictionary. Because these terms are not always distinguished in the literature, and because they both analyze the content of identifiers and comments, we will generally use "semantic" and "conceptual" interchangeably.

The idea of using underlying knowledge bases or dictionaries to help determine the intent of the program is a good one; however, we do not use them in our research. Creating and using such knowledge bases is a challenging task and beyond the scope of this thesis. On the other hand, we do make use of conceptual metrics that are based on usage statistics.

The C3 metric [MP05] is a good example of a cohesion metric based on usage statistics. It measures the similarity in word usage between two methods. It is similar to document similarity metrics that measure the overlap in content between documents. For C3, the "documents" are the methods, and the words embedded in the identifiers and in the comments constitute the content of the documents.

C3 collects data on all of the methods in a class. For each method, it gathers the words that are present in the identifiers and comments, and puts them into a vector using latent semantic indexing (*LSI*) [DDF+90]. The authors note that other information retrieval based techniques could be used, e.g., vector space models [TP10, SWY75], but that they chose LSI due to positive experiences with other software tasks based on semantics. They define the *Conceptual Similarity between Methods (CSM)* function as the cosine similarity between their vectors. The C3 value for a class is the maximum of zero and the average CSM value for each pair of methods in the class. C3 ranges from a minimum of 0 to a maximum of 1.

Because a C3 implementation was not available to us, I have created a variant of C3, the *C3 (Vector space model variant)* (*C3V*) metric. The primary differences between C3 and C3V is that C3V uses a vector space model to create a vector rather than LSI, and that the distance function that C3 uses only computes similarity between methods, while C3V's function calculates distances between all class members.

While semantic cohesion metrics can be useful, they can also be unreliable. They depend on the programmer encoding meaningful information in the identifiers and documentation. For the extreme case of obfuscated code (or any code having non-meaningful identifiers), semantic cohesion metrics are nearly useless. Furthermore, a class's semantic cohesion measurement can change due to a simple renaming of one of its members. This is an undesirable consequence,

given that the class functions exactly as before.

Comparative studies

Many researchers have done comparative studies on cohesion metrics, often in the context of showing why their new new metric is an improvement, e.g. [BB04, CKB00, ZLLX04]. The following paragraphs give a brief summary of the results of some of the more independent studies.

Briand, Daly, and Wust [BDW97] analyzed many cohesion metrics and created a theoretical framework encompassing the key points behind the various metrics. They observed that metrics differ in how they handle various features of object-oriented programming languages (e.g., inheritance, constructors, accessors, etc.). They note that there are different reasons for measuring cohesion, and that the purpose of the measurement should guide the selection of an appropriate metric. For the purposes of deciding when to split a class, they recommend cohesion metrics that are based on indirect connections between methods and attributes, rather than on direct connections.

Etzkorn et al. [EGF+04] set up tests to correlate the opinions of software engineers regarding the cohesion of classes with the measurements produced by eight cohesion metrics. Each test was conducted on test sets consisting of fewer than 20 test classes. One group of evaluators was composed of seven software engineers with at least a BS degree in computer science or electrical engineering and five years of experience. The other was composed of graduate students studying software engineering. While there was variability between the opinions of the two group of evaluators, both groups' opinion-based cohesion scores correlated most closely with LCC. (However, Dagpinar and Jahnke [DJ03] found no correlation between LCC and maintainability when they examined the maintenance history of two software systems.) Etzkorn's study also found that there was often poor correlation between the cohesion values produced by some of the metrics.

Marcus and Poshyvanyk looked at the correlations between eleven different cohesion metrics [MP05]. They found significant correlations between some metrics and very little between others. The highly correlated metrics tended to be in the same "family" of metrics, e.g., the ones derived from LCOM; however, they did see some cross family correlations, including between the structure-based LCOM* [Hen96] and their semantics-based C3.

Barker and Tempero [BT07] conducted an empirical study of 16 cohesion metrics on 92 Java applications to see what values were typical for real-world systems. They also found some correlation between metrics in the same family and significant variation between metrics in different families. They concluded that "it is unclear which are of useful value".

All of these studies found significant variation between different families of structural cohesion metrics. Some of these variations are due to imprecision in how the metrics were defined. For example, some metrics do not state what happens when there are no attributes or no methods. In other cases, implementers of metrics may make arbitrarily different decisions. One metric will consider a class with a single method to be maximally cohesive while another will consider it maximally noncohesive. Metrics also frequently differ in how they treat special methods. Section 2.4 discusses some techniques we have introduced to address some of the problems relative to special methods.

2.1.4 Interclass complexity – coupling

Coupling refers to the dependencies of entities on external entities, for example, a method in one class depending on a method in another class. These dependencies include methods in one class calling methods in another class, inheritance relationships between classes, the use of a class as an attribute type, the use of a class as a parameter type or as a return type for a method, etc. [BDW99, CK91, EKS94, HM95].

Low coupling may indicate a good division of the functionality of the classes, i.e., classes are not directly reliant on the capabilities of many other classes. High coupling may indicate potential maintenance problems. With highly coupled classes, it can be more difficult to isolate the source of a behavior among the densely interconnected classes. Also, when a highly coupled class is changed, these changes are more likely to require follow-on changes to the coupled classes.

Metrics differ in how they assign weights to various kinds of coupling. For example, some metrics [EKS94, HM95] consider the direct access of one class's data by another to be a tighter coupling than when classes share data via the parameters of methods. In their summary of the variation between coupling metrics, Briand and his colleagues [BDW99] noted that the strength of coupling varies according to the types of connections and the numbers of connections between classes. Most coupling metrics give scores for an entity, e.g., they measure the degree to which a

class is connected to multiple other classes.

Coupling metrics can often be classified as either import coupling or export coupling metrics. *Import coupling* metrics measure the dependencies a particular class has on others. The *Message Passing Coupling (MPC)* metric [LH93] is an example of an import coupling metric, because MPC counts the number of times a class calls methods external to the class. *Export coupling* metrics measure how much a particular class is relied upon by others. *Coupling Between Object Classes (CBO)* [CK94] is a well-known example. For a given class, CBO is the number of classes which are in a *uses* relationship with the indicated class; i.e., the count of the classes that access the methods or attributes of the indicated class.

Coupling metrics typically indicate which individual classes are highly coupled to others. They do not typically identify excessive coupling between two particular classes, which can be useful information for the task of identifying classes in need of refactoring. Nevertheless, several refactoring researchers have either used coupling metrics or have incorporated some of the principles of coupling metrics [TM05, LM06, FTSC11], mostly for help in determining whether a method should be moved from one class to another. Section 2.2.1 discusses coupling relative to detecting feature envy.

2.2 Related work – detecting bad smells

Bad smells are emitted from code that "stinks" and needs changing [FBB+99]. Bad smells are indicators of potential maintainability problems that can often be fixed by refactoring.

We agree with Beck and Fowler [FBB+99] that "no set of metrics rivals informed human intuition" for detecting bad smells, at least for the current state of the art. In many small-scale software development scenarios, experienced developers can easily locate classes that are prone to error or difficult to maintain. However, on large systems, or when there is a lack of such expertise, it is useful to have a consistent, programmatic way of locating potential problem classes. The combination of metrics and visualization tools can be particularly effective for helping programmers identify classes in need of refactoring [SSL01, PGN08, WL08].

The existing research on locating classes with bad smells using metrics [CLM06, DBDV04, FTSC11, LM06, MB08b, SSL01, TC09, TM05, Mun05, WL08] generally attempts to translate qualitative bad smell descriptions [FBB+99] into quantitative

criteria by which faulty classes can be identified programmatically. This entails picking the proper combination of metrics and thresholds for those metrics.

This section discusses how the feature envy and large class smells can be detected using metrics. Subsequent chapters discuss techniques for identifying how to refactor the classes once these smells are detected.

2.2.1 Detecting feature envy

A class suffers from feature envy when it makes heavy use of another class's methods or attributes [FBB+99]. When this occurs, it is worth determining whether some of the methods and attributes should be moved between the classes to lessen interclass dependencies and the consequent coupling.

There are several feature envy detection strategies in the literature, e.g. [LM06, TC09, TM05, SSB06]. Of these, the approach taken by researchers at the University of Macedonia [TC09] is the most detailed. Because there is much detail, we will only give a brief overview here. The cornerstone of their approach is an *entity placement metric*, which measures how tightly associated each class member is with its own class, i.e., what proportion of the member's interactions are with members of its own class rather than members of other classes. The entity placement value for the system is the average value of the entity placement scores of its classes. Their technique allows them to identify which members are closely associated with other classes, and after relocating those members via refactoring, the system entity placement metric enables them to determine whether the system as a whole has improved. Their approach will be discussed in detail in Section 5.2.2.

2.2.2 Identifying large (god) classes

Large classes often contain too much functionality. When a class has too much functionality, some groups of functions might be better compartmentalized by placing them into additional classes.

There is no consensus on how to locate a class that is difficult to maintain due to its large size. There have been metric-based recommendations to limit class size since at least 1994, when Lorenz and Kidd [LK94] stated that the number of public instance methods in a class should be ≤ 20 for a non-UI class and ≤ 40 for a UI class, and the number of instance variables should be ≤ 3 for a non-UI class and ≤ 9 for a UI class. On the other hand, several large class detection strategies are

based on more than just the size of the class [LM06, TM05, CAG11, OCS10, WL08]. These authors use the term *god class* to indicate that it is a special kind of large class that is capable of performing many tasks. These god class detection strategies identify classes above a certain complexity and/or below a certain cohesion. (The specific criteria that we use to detect god classes are given in Section 2.4.3.)

However, it also seems clear that many programmers have a tolerance for large classes. For example, Gorschek, Tempero, and Angelis [GTA10] wanted to determine what theories of good object-oriented design were used in practice. They conducted a questionnaire-based on-line survey and analyzed 3785 responses from software practitioners. Over half of the respondents either were against a limit on the number of methods per class or did not care how many methods a class had. In the same study, only 45 respondents thought that there should be an absolute upper limit on the number of methods per class.

2.2.3 Determining thresholds of smell detection

For those people who believe in limits on class size, there is significant difference of opinion regarding what those limits should be. The 45 respondents from Gorschek's study [GTA10] that supported an absolute upper limit had a median suggested limit of 10 methods per class. For the 2450 respondents who were "somewhat supportive" of a limit, the median suggested limit was 15 (although the highest suggested limit was 1000). For those researchers who implement bad smell detectors, the thresholds for the different metrics within the god class queries can differ considerably from one approach to the next.

Some bad smell detection approaches attempt to derive meaningful thresholds based on statistical measurements. For example, Lanza and Marinescu's [LM06] thresholds are based on mean values and standard deviations, e.g., for many metrics, a high threshold is equal to the average value plus the standard deviation. Crespo, Lopez, and Marticorena [CLM06] make use of quartiles to determine metric value thresholds, e.g., a high value might be one in the upper quartile of a distribution. To our knowledge, nobody claims that their queries use threshold values based on empirical data that correlates metric values to maintenance activity. Because the threshold values are based primarily on intuition, it is not surprising that the values chosen vary widely. For example, the WMC threshold for god classes ranges from 13.5 [CLM06] to 47.0 [LM06], depending on the approach and project being investigated.

There is still considerable debate about what constitutes appropriate criteria for determining whether a large class is difficult to maintain and needs modification. However, unless the maintenance approach is fully-automated, it is not necessary to have precise criteria. Instead, maintainers can search for god classes in an iterative fashion. If the results of a search contain too many classes that do not need refactoring, the search criteria can be made more restrictive. If the results of a search contain a high percentage of classes that do need refactoring, the search criteria might be loosened in an attempt to find more classes to refactor. The decision about whether specific search criteria are effective can be based on human judgment or on other criteria, e.g. measurable improvement of the refactored classes.

2.3 Analysis of existing cohesion metrics – sources of failure

Some of our early work on refactoring god classes [CAGN09] used a god class query to identify potential candidates for the *Extract Class* refactoring. It seemed like our goal of locating poorly designed classes amenable to refactoring would be a simple matter. The plan was to create a bad smell query that used a cohesion metric to locate noncohesive classes, refactor those noncohesive classes, and then use the cohesion metric to validate that the cohesion had improved.

The plan did not work as well as we had hoped. Relative to our intuition, the cohesion metrics reported an inordinate number of false positives (classes that were reported as noncohesive, but were not) and false negatives (classes that were reported as cohesive, but were not). While the metrics helped narrow the scope of classes to look at, it was still necessary to inspect the classes manually to eliminate false positives.

We attribute much of the mismatch between our expectations and the actual cohesion results to flaws in how the metrics measure software. In their paper about software metric validation, Kitchenham et al. [KPF95] state that an important assumption regarding measuring software is "that two units contributing to a particular value are equivalent." This is where existing metrics often fail – they do not distinguish enough between the various roles of methods, attributes, and the relationships between them when measuring cohesion.

This section discusses our identification of Java features that can have

undesirable affects on cohesion measurements. It explains how the presence of these features can cause cohesion measurements to be higher or lower than expected, sometimes to a large degree. We have separated these features into two broad categories, illusory and hidden cohesion, depending on whether they typically generate higher or lower cohesion values than expected. Many of these features will be discussed in the context of some deliberately noncohesive "PersonCar" classes from our test suite.

2.3.1 Example noncohesive classes – PersonCars

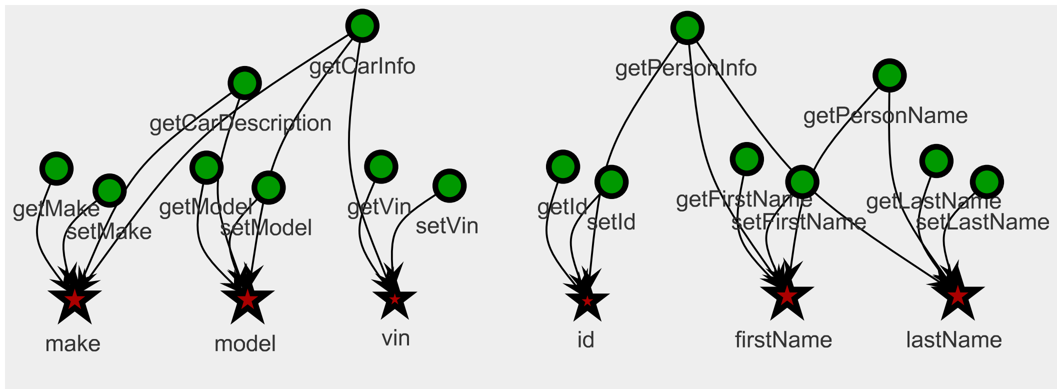
This section describes three closely related classes from our test suite that were designed to be conceptually noncohesive, and whose source code can be found in the appendices – `PersonCarDisjoint` (Appendix A.2), `PersonCarDirect` (Appendix A.3), and `PersonCarSpecial` (Appendix A.5). Figure 2.4 shows intraclass dependency graphs for the three PersonCar variants.

Each of these classes has functionality pertaining to persons and functionality pertaining to cars. The person functionality includes the attributes `id`, `firstName`, and `lastName`, their accessor methods, and the `getPersonInfo` and `getPersonName` methods. The car functionality includes the attributes `make`, `model`, and `vin`, their accessor methods, and the `getCarDescription` and `getCarInfo` methods. `PersonCarDirect` and `PersonCarSpecial` also have additional functionality unrelated to the primary purpose of the classes.

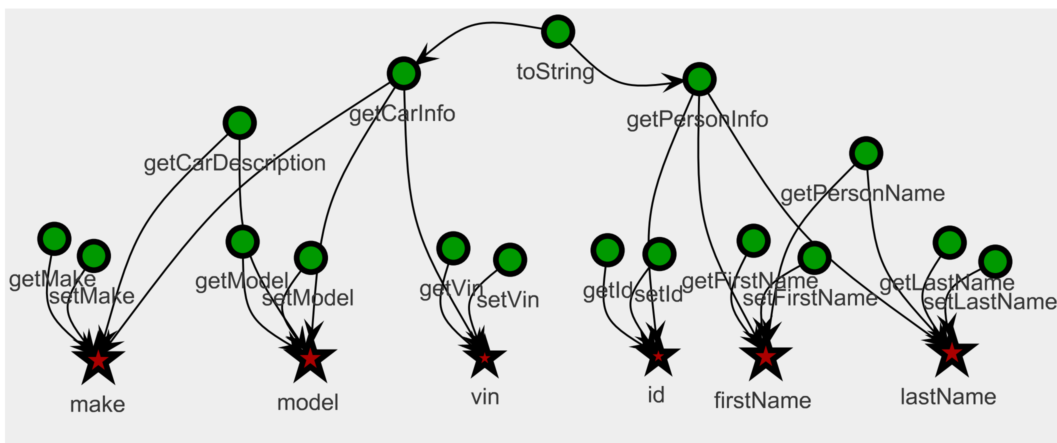
In each of the graphs in Figure 2.4, there is a group of members in the bottom left that pertains to cars, and is a group of members on the bottom right that pertains to persons. The three classes differ as follows:

1. The `PersonCarDisjoint` class of Figure 2.4(a) was designed to be noncohesive. It contains two distinct groups of members, and there are no interactions between the two groups.
2. The `PersonCarDirect` class of Figure 2.4(b) is like `PersonCarDisjoint`, but has an additional `toString` method that calls `getCarInfo` and `getPersonInfo`, thereby connecting the previously disconnected groups.
3. The `PersonCarSpecial` class of Figure 2.4(c) is like `PersonCarDirect`, but also has `equals` and `hashCode` methods that access all of the attributes and a `logger` attribute that is accessed by many of the methods.

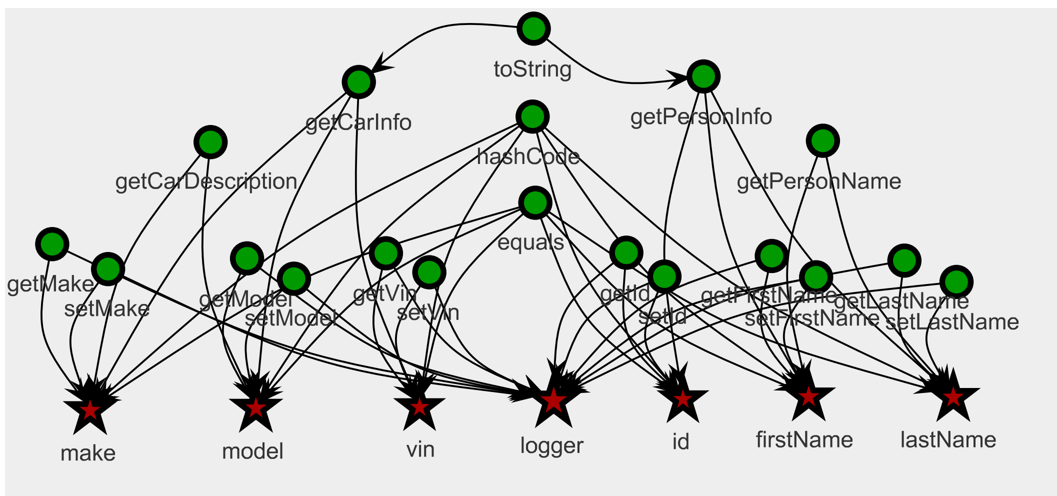
These classes will help illustrate the effects of illusory cohesion.



(a) PersonCarDisjoint



(b) PersonCarDirect



(c) PersonCarSpecial

Figure 2.4: PersonCar variations

2.3.2 Illusory cohesion

We use the term *illusory cohesion* to refer to code constructs that cause cohesion values to be higher than expected. Sources of illusory cohesion include:

- Code that initializes an object, generally setting the values of many attributes.
- Code that deals with the identity of an object based on its attributes.
- Other cross-cutting concerns – code that is not related to the primary reason of the object's existence, but that accesses many of the attributes of an object.

This section discusses many of the constructs causing illusory cohesion.

Constructors and initializers

Constructors and initializers frequently set the initial values for many attributes, while not providing any domain logic. This can cause misleadingly high scores, because the attributes are linked together through the initialization code. For this reason, a number of cohesion metrics [BB04, BK95, CKB00] do not consider constructors when calculating cohesion. A problem with this approach is that some programmers embed significant behavior in constructors, while those constructors may have little or no explicit attribute initialization.

It is easy to detect and omit constructors from consideration; it is not always easy to programmatically detect other initialization methods that set the initial values of many attributes and make the class seemingly cohesive. Particular projects may or may not have naming conventions that would make initialization easier to detect. In some cases, if initialization methods can not be detected readily, they may cause the cohesion metric to show a *lower* than expected score when constructors are removed from consideration. For example, factory methods' [GHJV94] primary activity is to call a constructor. When constructors are filtered out, factory methods will appear to be disconnected from the rest of the class. Notice the isolated `create*` elements in Weka's `XMLElement` class[2] in the upper right of Figure 2.5; the code for one of them is shown in Figure 2.6.

We believe that constructors should generally be ignored when calculating cohesion; however, there are circumstances when they should not be, e.g., when constructors perform some functions besides initializing attributes.

[2]Appendix B contains information on the open source software used in our work.

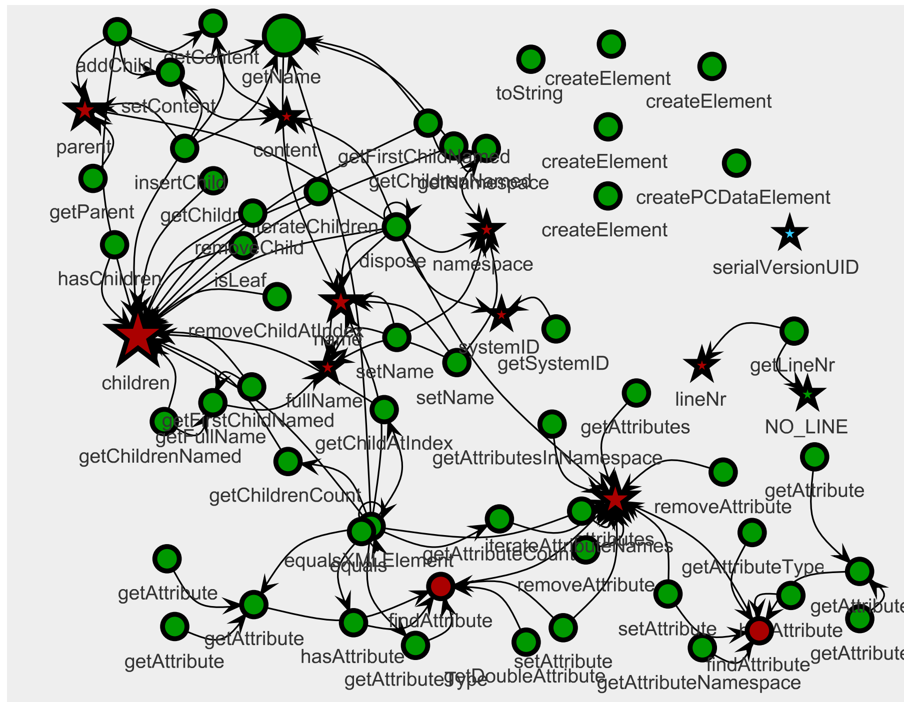


Figure 2.5: Constructor removal in XMLElement

```
public IXMLElement createElement(String fullName) {
  return new XMLElement(fullName);
}
```

Figure 2.6: Example factory method

Object's methods - toString, equals, ...

The methods provided by Java's `Object` class are not intended to provide domain logic; rather, they are used to provide general-purpose capabilities. For example, `toString` can be used to provide a human-readable version of an object, and `equals` is used to compare two objects to see if they are the same. For many metrics, these methods cause misleadingly high scores.

The only difference between the `PersonCarDisjoint` and `PersonCarDirect` classes is that `PersonCarDirect` has a `toString` method that calls `getCarInfo` and `getPersonInfo`, thereby connecting the previously disconnected graph. A small change such as this can have big effects on cohesion scores. For example, `PersonCarDisjoint` has an LCC score of 0.47.

`PersonCarDirect` has the maximal LCC score of 1.0, because `toString` enables (undirected) paths between all methods.

A close look at the top, middle of Figure 2.5 shows an interesting aberration in the `XMLElement` class – `toString` is a node disconnected from the rest of the graph, which would cause a misleadingly low score! In this case, `XMLElement-.toString` did not directly access the attributes. Instead, it made use of a method in another class, `XMLWriter.write(XMLElement)`, to generate a string.

Cross-cutting concerns (especially logging)

The presence of cross-cutting concerns can also make cohesion scores higher than expected, because many of the class members may make use of auxiliary functionality provided by the cross-cutting members. For example, classes often have methods that record information to a log file via a "logger" attribute, or they may have methods that print out debugging information as they execute.

The detection of cross-cutting concerns is difficult, and is an on-going research effort [ZGC08, MDM07]; however, simple approaches can suffice to identify some of the most common cross-cutting concerns, e.g., logging. A simple approach that works for some logging is to search for attributes of a known logging type; however, this is weak approximation of what needs to be done. In the case of logging, classes might utilize a less well-known logging package or a custom-built one. Other cross-cutting concerns can be harder to detect, because they may not have common implementations.

2.3.3 Hidden cohesion

We use the term *hidden cohesion* to refer to code constructs that are typically ignored in cohesion metrics but that indicate relationships between certain of a class's members. When hidden relationships are not taken into account, cohesion measurements appear lower than expected.

Static members

Some researchers choose to ignore static methods when measuring cohesion [BT07], because static methods can only directly access static members, not instance members. On the other hand, instance methods can access static members, so the use of static methods can indicate common functionality. For

cohesion metrics that consider connectivity between methods (e.g., TCC, LCC, DC_D, and DC_I), the omission of static methods from consideration can cause lower measurements than if they were included.

Sometimes there is very little difference between static methods and instance methods. Programmers may write methods in a procedural style, where all access to data is via the objects provided to the method as arguments. Because there are no direct attribute accesses, these methods may or may not be declared static. In these cases, it seems inconsistent to treat static and instance methods differently.

It is also unclear how to best handle static attributes, particularly static final "constants". Some constants are provided primarily for the use of external classes, and would lead to misleadingly low scores; however, others may indicate legitimate connections within the class. For example, in a GUI class, a constant might be used for a button label, and that same constant may be used within the class to help process the "button pressed" event. Use of constants in this way indicates a meaningful relationship between the method that constructs the button and the method that processes the button's events. In these circumstances, for cohesion metrics that consider connectivity between members, omission of constants from consideration would cause lower cohesion measurements than if they were included.

Inheritance

There are alternative perspectives in deciding whether a class is cohesive when it inherits from another class [BK95, BDW97, EKS94]. One perspective is to consider only the subclass's extensions of its superclass and determine whether the members defined in the subclass constitute a cohesive class independent of the members of its superclasses. Another perspective is to evaluate the combination of the subclass's members and its superclasses' members to see whether the combination constitutes a cohesive class.

Most papers describing metrics do not state how they deal with inherited members. In most cases, it appears that they concentrate on the first perspective. This is a simpler case to handle, because cohesion can be calculated by examining the code within a single class, without having to look at the class hierarchy. For cohesion metrics that consider only the non-inherited members, methods that access inherited attributes, but not their own, cause the cohesion score to be misleadingly low.

There are similar problems when measuring classes that are inherited from, i.e., superclasses. A superclass may define methods or attributes that are not used directly by itself, but which provide common functionality for its subclasses. As far as we are aware, no cohesion metrics consider this case, which can lead to seemingly noncohesive superclasses.

If one does consider inherited members, there are several options. Bieman and Kang [BK95] state that TCC and LCC have three options relative to measuring cohesion – (1) ignore the inherited members, (2) include the inherited members, (3) include the inherited attributes, but not the inherited methods; however, they do not go into detail about how the calculations should be performed.

There are several options for how inherited members might figure into a cohesion calculation. For example, inherited members could be given different weights than non-inherited members. Alternatively, the cohesion calculation could be performed by "flattening" the inheritance hierarchy, as though the inherited members were moved into the inheriting class. For now, the state of the practice seems to be to ignore inherited members.

Methods imposed by interfaces

Java interfaces give program designers the ability to specify combinations of methods and constants responsible for providing desired behaviors. Disparate groups of programmers can use interfaces as contracts that specify how their code interacts [ZHR+06]. Because interfaces serve as contracts, they are less conducive to change than groups of methods that are not parts of interfaces [Fow02].

We are not aware of any cohesion metrics that treat methods required by interfaces specially. For cohesion metrics that are attempting to measure the quality of a class based on the interconnections of the class members, this is not a problem. However, the constraints imposed by interfaces are important when the cohesion metric is to be used for other purposes, e.g., detecting classes that should be refactored. For determining whether classes are amenable to refactoring, existing cohesion metrics will provide a lower score than desired, because they do not recognize the requirement for keeping together the members imposed by interfaces.

Abstract methods

Abstract methods are methods that are declared but not yet implemented; implementations will be supplied by subclasses. Nevertheless, the abstract methods may be accessed by other methods in the abstract class. Because abstract methods do not reference any of the other class members, they cause misleadingly low scores for most metrics.

Data transfer classes

When we reviewed the results of a large class query that did not contain a WMC complexity term, we noticed a large number of classes that were little more than a collection of attributes with associated accessors and mutators (sometimes referred to as *data transfer classes* [PNA10] or *value classes* [Blo08]). These are meant to package data but not behavior. Classes like these give low cohesion scores when measured by most cohesion metrics that just examine interactions within the class, because there is no behavior in the class causing interaction between the various members.

This does not necessarily mean that these data classes should be split into smaller ones. Rather, classes like these need to have their cohesiveness judged on the basis of how their clients access them, e.g., whether different sets of client classes consistently access the same subsets of members. This proposed interclass analysis goes beyond the intraclass analysis of most cohesion metrics, so client-based cohesion techniques [ML07, ML09] are needed.

Indirection

Some code within a class does not call other code within the class directly, but may use that code through an intermediary (e.g., event listeners, callbacks). In such cases, cohesion metrics will produce a value that underestimates the cohesion of the class.

Consider the code in Figure 2.7. The `createEdgeTypeCombo` method creates a `JComboBox` GUI component. Without knowledge of the GUI event model, it is not obvious by examining this code that the GUI user can cause the component to generate an `ActionEvent` and that the event will have the `JComboBox` GUI component stored as its source.

The `actionPerformed` method listens to `ActionEvents`. If it detects an

```java
private JComboBox createEdgeTypeCombo()
{
  Vector<String> edgeTypes =
    new Vector<String>();
  JComboBox edgeTypeBox =
    new JComboBox(edgeTypes);
  // ...
  edgeTypeBox.setName("EDGE_TYPE_COMBO");
  return edgeTypeBox;
}

public void actionPerformed(ActionEvent event)
{
  // ...
  if (source instanceof JComboBox) {
    JComboBox box = (JComboBox)source;
    if ("EDGE_TYPE_COMBO".equals(box.getName())) {
      handleEdgeTypeRequest(box);
    }
  }
}
```

Figure 2.7: Example use of an event listener

`ActionEvent` from a `JComboBox` and the box's name is "EDGE_TYPE_COMBO", it executes the `handleEdgeTypeRequest` method. Although there is no calling relationship between the `createEdgeTypeCombo` and the `actionPerformed` methods visible within the class, there is a relationship via the unseen event forwarding and handling mechanism. This relationship is not handled by any cohesion metric that we know of.

In the general case, discovering these indirect relationships is a computationally expensive problem to solve, because there can be arbitrarily long chains of methods that lead to the indirect member access.

```
Object classifier = this.getUserObject();
// (code omitted)
Method getter = m_Properties[i].getReadMethod();
// (code omitted)
Object value = getter.invoke(classifier, args);
```

Figure 2.8: Example use of reflection

Reflection

Java's reflection capabilities enable a programmer to write code that accesses a class or members of a class without statically naming that class or those class members. The use of reflection is particularly common in GUI frameworks, e.g., JavaBeans. In these situations, there is a requirement for a method to be present, or there may be a hidden relationship between methods, but it is difficult to determine that relationship through simple source code examination. Often, these "required" methods are specified via a property file generated from a GUI builder. Situations like these are very difficult to detect.

As an example, consider the code in Figure 2.8, taken from Weka's `GenericObjectNode` class. This method obtains and invokes methods, but the specific methods being invoked can not be determined by examining the code of the class.

Many Weka classes use reflection. Consider the intraclass dependency graph of the `DataGenerator` class in Figure 2.9. It appears to have many disconnected methods with names ending in "TipText". A more thorough examination of Weka code in other classes shows that reflection is used to get the properties available for an object, and then to see if there is a `<propertyName>TipText` method. The `weka.gui.PropertySheetPanel` class is an example of a class that does this.

Serialization

The `serialVersionUID` attribute is present in many classes that implement the `Serializable` interface. It is often not accessed by any other code in the class, only by the virtual machine's serialization code. The same is true of the `readObject` and `writeObject` methods. These members will appear to be disconnected from the rest of the class. Practically speaking, these methods will have a small effect on the cohesion measurements of a large class, so they will not be discussed further.

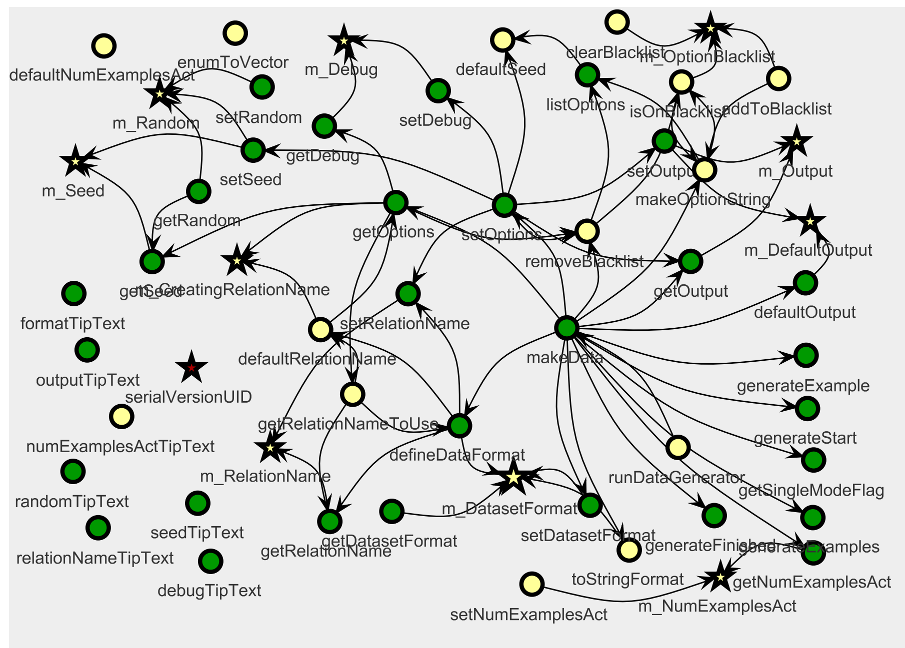


Figure 2.9: Reflection in DataGenerator (*TipText methods)

Nested classes

We have seen no metrics that explicitly address nested classes (e.g., inner classes, anonymous inner classes). This is not surprising, since some cohesion metrics were targeted at languages without nested classes, while other metrics were developed before nested classes entered Java. By default, the implementations we have seen treat nested classes the same as non-nested classes, therefore, any interactions between the containing class and the nested class do not contribute to cohesion. This approach seems consistent with the description of static nested class in the *Java Tutorial* [ZHR+06]: “A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.”

On the other hand, for non-static nested classes (a.k.a. *inner classes*), this seems wrong. These classes are created to provide some additional encapsulation of data for the use of the enclosing class, and one might expect extensive interaction between the members of the enclosing class and the members of the nested class. The *Java Tutorial* [ZHR+06] states “An instance of an inner class can exist only within an instance of its enclosing class and has access to its enclosing class’s members even if they are declared private.” Using this rationale, inner classes

are little more than convenient packaging of functionality for use of the enclosing class, and for cohesion purposes, might best be treated as part of the enclosing class.

2.4 A technique for improving structural cohesion metrics

We see two major axes of variation in structural cohesion metrics:

1. The data (methods, attributes and their inter-relationships) that are considered in the calculation.
2. The algorithms used to calculate the metric values based on this data.

Our examination of the sources of illusory and hidden cohesion convinced us that many of the unintuitive results were not due to the cohesion algorithms themselves, but rather due to the data on which they operated. We decided to implement a technique that would enable analysts to modify the input data based on their knowledge of project-specific coding conventions.

2.4.1 Approach – restructuring graphs

While there are opportunities to create improved algorithms, there are already many cohesion metrics based on sound underlying ideas. Rather than introduce another algorithm, we have attempted to make existing cohesion algorithms more effective by manipulating the data on which they operate. In effect, our approach transforms the input space by filtering out nonessential members and relationships, and introducing additional inter-member relationships (beyond method calls and attribute accesses).

Some existing cohesion metrics handle some sources of illusory cohesion by treating certain class members and inter-member relationships specially [BK95, BDW97, CKB00], but there is little consistency in how they do so. Consider the metrics discussed in Section 2.1.3. The original LCOM has no concept of special methods. TCC and LCC operate on "visible" methods but exclude constructors. DC_D and DC_I consider public methods, exclude constructors, and treat overloaded methods within a class as a single method. CBMC excludes constructors, access methods, and delegation methods. These are all instances of filtering the input data before calculating cohesion values, and each metric has its own filter.

If one views the representation of the input data as a graph representation problem, one sees that most existing metrics use a fairly simple transformation from code to graph, where methods and attributes are represented by nodes, and the accessing relationships are represented by the unweighted edges between them. The existing cohesion metrics differ in their determination of which members get included as nodes. In effect, special methods are handled by omitting their nodes from the intraclass dependency graph. More generally, we may choose not just to omit nodes, but also to add, delete, and/or merge both nodes and edges. Illusory cohesion is handled by removing nodes or edges. Hidden cohesion is handled by combining nodes or adding edges.

2.4.2 Cohesion metrics – modified implementations

Section 2.3 discussed how certain Java constructs can cause cohesion metrics to produce measurements which do not suit an intended use of the metrics. In addition, metrics may produce unsuitable measurements due to lack of simple domain knowledge. To address these deficiencies, I added flexibility to an existing metrics suite.

The *Metrics2* [SB10] Eclipse plug-in provides several cohesion metrics. Because metrics can be used for multiple purposes, and because projects can follow various coding conventions, I modified Metrics2 to allow users to indicate the handling of various language or design constructs by specifying preferences. Preferences provide analysts the ability to modify the output of the metrics based on their intended use and to adjust the behavior of the metrics based on project-specific knowledge, i.e., analysts can tune the metrics package to suit their needs.

The preferences control the construction of the input graphs used by the cohesion metrics but do not otherwise alter the metrics' algorithms. Users can specify that the cohesion metrics should process the class members as originally described in the metric definitions, or they can specify the members to be considered in the calculations by setting other preferences. There are two types of these preferences.

1. *Boolean user preferences* indicate whether certain language or design features (e.g., constructors) should be included in the cohesion calculations. These are useful when the feature is easy to detect programmatically. In most cases, these preferences indicate which types of nodes should be excluded from

the input graph, but some preferences indicate that certain nodes should be merged.

2. *Filter patterns* specify Java regular expressions to be used for excluding class members with matching names. These are useful for removing sources of illusory cohesion based on project-specific knowledge.

The following subsections describe the user preferences in more detail.

Illusory cohesion

There are several boolean preferences useful in eliminating illusory cohesion. The following preferences control whether or not particular constructs should be considered when calculating cohesion.

- *countConstructors* – If false, eliminates nodes representing constructors from the input graph.
- *countObjectsMethods* – If false, eliminates nodes representing methods declared by the `Object` class (`toString`, etc.).
- *countLoggers* – If false, eliminates nodes representing attributes whose type is `java.util.logging.Logger`.

In addition to the boolean preferences, the *ignoreMembersPattern* can be used to filter class members whose name matches the provided pattern. When these members are highly connected to others, removing the corresponding node from the graph can decrease the cohesion scores. For example, a pattern of "init" can eliminate from consideration nodes representing initialization code. Because initialization code often accesses many attributes, removing the matching "init" nodes removes a highly cohesive part of the input graph. Similarly, a pattern of "log.*Util" might be used to help eliminate user-defined logging members that are highly connected to other methods in the class.

Hidden cohesion

The following boolean preferences are useful in dealing with hidden cohesion by specifying whether or not particular constructs should be considered when calculating cohesion.

- *connectInterfaceMethods* – If true, methods imposed by interfaces should be considered as connected in the input graph (as though they called each other).

- *countStaticAttributes* – If true, adds nodes representing static attributes to the input graph and links these to the nodes of the methods that access them.
- *countStaticMethods* – If true, adds nodes representing static methods and links these to the nodes of the methods that access them.
- *countAbstractMethods* – If true, adds nodes representing abstract methods to the input graph.

The *ignoreMembersPattern* can also help compensate for some additional cases of hidden cohesion. It can increase cohesion scores by filtering out seemingly disconnected member nodes. For instance, there is no easy way to detect class members that are linked via reflection. However, if one knows that certain methods appear disconnected but are "secretly" connected (e.g., via JavaBean properties), the corresponding nodes can be removed from the input graph. Thus, in the case of `DataGenerator` (Figure 2.9), a user could indicate that members matching "TipText" should not be considered in the cohesion calculation.

2.4.3 Cohesion metrics – input restructuring experiments

We ran experiments to assess the influence of some of the structural transformations described above on cohesion measurements. One group of experiments measured cohesion on classes in our cohesion test suite [CAGN10]. Using these classes as input, we evaluated the effects of various language constructs on the cohesion measurements in a controlled manner.

Another group of experiments compared the original results of a god class query to those produced using transformed input data. This group of experiments was performed on two mature open source projects – Weka 3.6.2 [HFH+09] (Appendix B.2.6) and JHotDraw (Appendix B.2.5) 5.3.0 [GE07]. Measuring mature open source projects gives an idea of the effect of the transformations on the cohesion measurements of real code.

The *Metrics2* Eclipse plug-in provided measurements for four cohesion metrics – TCC, LCC, and slight variants[3] of DC_D, and DC_I. We chose TCC, because Lanza and Marinescu [LM06] have used it to help locate large, noncohesive classes, and we used LCC, because Etzkorn's study [EGF+04] indicated that LCC was the preferred cohesion metric of two groups of professional programmers. DC_D and

[3]Metrics2 modifies the original DC_D and DC_I in the following ways: (1) classes with fewer than two methods receive a value of 1.0 (max. cohesion); (2) overloaded methods within the same class are treated as separate methods.

DC_I are closely related to TCC and LCC, but also consider connecting methods in their calculations.

Test suite results

The results in this section show the effects of preferences on the cohesion scores of three similar classes. Table 2.1 shows the cohesion scores for the three `PersonCar` classes in Figure 2.4 using non-transformed inputs. For what is essentially the same class, the scores vary considerably, both between metrics and between classes. `PersonCarDisjoint` is the most obviously noncohesive class, and all the normalized metrics (all except LCOM) show it being under the 0.5 midpoint, although the metrics that measure indirect connections via class members, LCC and DC_I, are near that midpoint. On the other hand, `PersonCarSpecial` is shown as cohesive. LCC and DC_I show both it and `PersonCarDirect` as being maximally cohesive, despite its fundamentally dual nature.

Table 2.1: Original, non-transformed cohesion measurements

Class	TCC	LCC	DC_D	DC_I
PersonCarDisjoint	0.23	0.47	0.23	0.47
PersonCarDirect	0.32	1.0	0.32	1.0
PersonCarSpecial	0.81	1.0	0.81	1.0

When the same classes are measured again, using a transformation that removes loggers and methods inherited from `Object` from consideration, the results become consistent. The measurements for the three `PersonCar` variants are now the same and equal to those of the `PersonCarDisjoint` when measured with no transformations. For all three classes, TCC and DC_D are 0.23, and LCC and DC_I are 0.47, scores that seem to better represent fundamentally disjoint classes.

Open source results

God class queries based on cohesion metrics can be useful for identifying classes in need of refactoring [CLM06, LM06, TM05, CAG11, OCS10, WL08]. We wanted to determine how changing preferences affected the results produced by god queries.

Our god class detection SQL query identifies classes that meet the following conditions:

1. number of instance methods > 20
2. tight class cohesion (TCC) [BK95] < 0.34
3. weighted method count (WMC) [CK94] ≥ 47.0
4. depth in the inheritance hierarchy = 1.0

The first three conditions specify a class's size, cohesion, and complexity. The fourth simplifies the analysis by eliminating the effects of inherited members. The NOM threshold is based on a recommendation by Lorenz and Kidd [LK94], while the WMC and TCC thresholds are based on recommendations by Lanza and Marinescu [LM06]. In addition to the above conditions, the query specifies the graph transformation preferences that were in effect for the TCC calculation.

We collected measurements on JHotDraw and Weka using various preferences. The results discussed in this section are based on two of these result sets. The *original* preference set specifies the use of the original definitions of the metrics, without transformations.

The choice of a *modified* preference set was based on two things: (1) the intended use of the cohesion metric, i.e., to identify classes that could be split, and (2) on manual examination of the results produced by the god class query using the original preference set. We were using the query to locate classes in need of splitting, so the modified preference set specifies that methods defined in the same interface should be connected (as discussed in Section 2.4.2). There is a preference to include static members, because static members provide meaningful connectivity information. The final difference from the original preference set is to exclude class members with "TipText" as part of their identifiers. Many of the Weka classes had seemingly isolated methods with "TipText" as part of their names. As discussed in Section 2.3.3, these methods are actually used via reflection.

Weka We ran god class queries using the two preference sets. Overall, 18 different Weka classes met the god class criteria for at least one set of preferences; 17 of the classes satisfied the god class query using the original inputs, but only 12 satisfied the god class query using the transformed inputs. 11 classes satisfied both queries. Manual inspection of the results showed that the reduction in the result set generated using the transformed input graph was predominately due to the effects of filtering out nodes representing methods that matched "TipText".

The only class in the result set for the transformed inputs that was not in the result set for the original was `XMLDocument`, which has a large number of static

final attributes. Static attributes were not included in the original calculations, so that TCC score (0.48) was above the 0.34 threshold, while the transformed inputs resulted in a 0.27 score.

Table 2.2 contains statistics regarding the amount of change (Δ) seen in the Weka classes' cohesion measurements between the two preference settings. While the average change was about 0.15 for the four metrics shown, the largest change seen was an increase of 0.61 in the DC_I measurement for Weka's `weka.datagenerators.DataGenerator` class (see Figure 2.9), again due mostly to the filtering out of `*TipText` methods.

Table 2.2: Weka cohesion measurement changes

Statistic	TCC	LCC	DC_D	DC_I
Average Δ	0.13	0.14	0.15	0.16
Max Δ	0.38	0.41	0.38	0.61
Min Δ	-0.21	-0.24	-0.22	-0.17

The above results show that modifying the input graph processed by cohesion metrics can alter the cohesion scores considerably. Modifying the cohesion preferences had a beneficial effect for identifying Weka god classes amenable to refactoring, mostly by eliminating from consideration the false positives, produced by seemingly isolated `TipText` methods.

JHotDraw We also ran god class queries on JHotDraw classes using the same preference sets as for Weka. Overall, 5 different JHotDraw classes met the god class criteria for the original specification, but none met the criteria using the transformed inputs. This was mostly due to the JHotDraw classes' heavy use of interfaces. Condensing the nodes representing the inherited methods resulted in a more highly connected graph and higher cohesion scores.

Table 2.3 contains statistics regarding the amount of change seen in the JHotDraw classes' cohesion measurements. The average change was about 0.46 for the four metrics shown, and the largest change was 0.73 for DC_D.

The effects of a particular preference can be large. The `AbstractFigure` class has a TCC value of 0.21 with the original preferences; however, it has 46 methods imposed on it by the `Figure` interface. Many of these methods simply return null or a constant in `AbstractFigure`. When these imposed members are condensed,

Table 2.3: JHotDraw cohesion measurement changes

Statistic	TCC	LCC	DC_D	DC_I
Average Δ	0.43	0.38	0.53	0.49
Max Δ	0.60	0.55	0.73	0.70
Min Δ	0.08	0.08	0.26	0.26

the TCC score becomes 0.81. A similar situation occurs in two of the other classes – `AbstractHandle` and `AbstractTool`. Modifying the cohesion preferences had a beneficial effect for identifying JHotDraw classes amenable to refactoring, mostly by eliminating from consideration classes that seemed noncohesive based on the structure of methods imposed by interfaces.

2.4.4 Cohesion experiments – conclusions

Our test suite results indicate that restructuring the inputs to cohesion metrics can improve the metrics' usefulness for detecting classes in need of refactoring and also for making the results of different metrics more consistent with each other. The technique can have a dramatic effect in cohesion metrics that are based on connectivity, because the numbers produced by some metrics are extremely sensitive to the addition or removal of a small number of edges.

Measuring cohesion is not the main emphasis of this thesis; however, the techniques provided should facilitate analysis of the sensitivity of cohesion metrics to different code constructs. We have shown how the presence of a `toString` method can cause a cohesion score for a class to jump from just under the mid-point of the metric's range to the maximum possible score.

The cohesion metrics are likely to give the most useful results when the preferences are used to adjust for local coding practices. Analysts who are familiar with a code base can adjust preferences based on their knowledge, thereby compensating for some sources of illusory and hidden cohesion. This claim has not yet been verified. In the future, we would like to do a more complete investigation of the effects of restructuring the inputs on the cohesion scores to better quantify how much each restructuring affects the various metrics.

2.5 Metrics and refactoring

Research in software metrics has helped our research in two main ways. This section discusses the use of software metrics as a means of measuring the quality of classes, e.g., locating potentially problematic classes, and also discusses how the qualities that are considered important by metrics can be used to help choose appropriate clustering algorithms.

2.5.1 Metrics as quality measurements

Several metrics based on the internal characteristics of software may correlate with maintainability [LH93, BBD01]; however, there is still considerable debate [DJ03, EBGR01, BWZ02, BEGR00, SK03, LH93], and there are remarkably few empirical studies. Given this lack of conclusive data, we rely on the conventional wisdom – small, non-complex, cohesive classes are preferable to large, complex, noncohesive ones [CSC06, BBG08, DJ03, LH93, MPF08, ASKM07, CDK98].

Prior work provides some guidance for choosing metrics for size and some aspects of complexity. Like other refactoring researchers [LM06, TM05, CLM06], we typically measure size based on the number of methods and/or attributes. We also follow the lead of prior researchers in measuring the complexity of a class. The WMC metric measures one aspect of class complexity, by adding up measures of the methods' complexity, and the methods' complexity is measured using the cyclomatic complexity metric [LM06, TM05, CLM06].

Cohesion metrics measure another aspect of class complexity. Choosing an appropriate cohesion metric is more difficult, because there are over 40 cohesion metrics in the literature, and there is no consensus on which of these metrics is most suitable for a particular purpose.

Based on the lack of correlation between many of these cohesion metrics [MP05, BT07], it seems clear that, at best, they measure different aspects of cohesion. In some cases, this is by design – the metrics are intended to measure different aspects of cohesion, e.g., structural characteristics vs. semantic characteristics. In other cases, this may be due to deficiencies in the metrics, such as those discussed in Section 2.3.

Rather than rely on any single cohesion metric, we use multiple cohesion metrics (LCOM, LCOM*, TCC, LCC, DC_D, DC_I, C3) in an attempt to get "truth by consensus". Analogous to some work in expert systems [MD85], we treat each

of several metrics as a source of expertise and then combine the opinions of the experts for a final result.

2.5.2 Metrics as hint providers for clustering

The software characteristics that are measured by software quality metrics also provide insight on how to refactor classes using clustering techniques. For example, structural cohesion researchers value interconnectedness among the members of a class. Classes with highly interconnected methods and attributes tend to receive high cohesion scores. The metrics discussed in Section 2.1.3 emphasized common access of methods to attributes. Some of the earlier metrics, e.g., LCOM, calculate their cohesion scores based on methods that directly access attributes. Chapter 5 discusses how agglomerative clustering can be used to cluster class members based on local structural information, e.g., methods directly accessing attributes.

Later cohesion metrics, e.g., LCC and DC_I, took a more global view of structural connectedness. These metrics take into account methods indirectly accessing attributes through chains of methods within the class. CBMC and ICBMC took the idea further by representing the access structure as graphs and determining cohesion by measuring the effort needed to disconnect the graphs. These ideas are similar to how graph-based clustering techniques are used to form clusters in Chapter 6.

Likewise, the criteria used in semantic cohesion metrics can be used as a basis for deciding how to cluster the methods and attributes of classes. Semantic cohesion researchers value overlap in meaning between the words found in the identifiers and comments in the different class members of a class. Section 5.3.2 discusses how agglomerative clustering can be used with semantic information to form clusters of members with similar semantic content, and how those clusters form the basis of revised classes.

Some of the characteristics of classes that can cause problems for metrics (Section 2.3) can also cause problems for clustering algorithms. Future chapters discuss how techniques like those discussed in Section 2.4.1 can be used to improve the results of clustering.

2.6 Summary of contributions

Cohesion measurements are used for many purposes. We used existing cohesion metrics to help identify classes from a variety of test classes and open source systems that may need refactoring. Some of those measurements did not correspond to our intuitive notion of cohesion. Further analysis of the classes that produced these measurements identified deficiencies in the metrics' handling of certain Java language features and constructs.

This chapter described our modification of certain graph-based cohesion metrics to address these deficiencies by restructuring the input graphs on which the metrics operate, based on user preferences. Most preferences indicate whether nodes corresponding to certain class members should be present in the intraclass dependency graph. Others affect the edges that are included in the graph. The "best" preferences to use may vary by software project, depending on the programming conventions being used. Knowledge of these conventions can enable the analyst to use preferences that adjust the input graph appropriate to the project.

While there has been previous work on reducing illusory cohesion by filtering specific "special methods" from the input, we believe that our modifications to Metrics2 create the first implementation of cohesion metrics with a general model for filtering special methods, and that our technique is also the first to address hidden cohesion. These enhancements to the metrics enable more accurate identification of classes amenable to refactoring.

The generality of our approach for specifying the features to measure should also be beneficial to analysts using cohesion metrics for purposes besides refactoring. However, cohesion measurement is not the emphasis of this thesis, and this claim has not yet been tested.

Doubtless, there are other language features and programming idioms not mentioned in Section 2.3 that affect cohesion. Some of these will be amenable to this transformation technique; others will not.

Chapter 3

Background – Refactoring and Clustering

This chapter presents background material on refactoring and clustering, including relevant definitions and high level concepts. It provides a description of some specific class-based refactorings and issues related to them, which will be helpful for understanding the refactoring results discussed in later chapters. This chapter also discusses clustering, how it relates to refactoring, and how the results of clustering can be evaluated. This material provides the basis for understanding Chapters 5-7, which discuss the application of particular clustering techniques to refactoring classes.

3.1 Refactoring

Refactoring [Opd92, FBB+99] refers to the restructuring [Arn89] of software, typically to make it more maintainable. The earliest use of the term "refactoring" as a software engineering discipline, seems to have been in work by William Opdyke and Ralph Johnson [Opd92, OJ90] in the early 1990s, who were interested in supporting the iterative design of object-oriented frameworks. Refactoring gained popularity with the publication of Fowler et al.'s book on refactoring [FBB+99] and the advocacy of the *extreme programming* [Bec00] community, who saw refactoring as an integral part of the rapid development of quality software. Since that time, "refactoring" has been used by many people in a variety of contexts, including ontologies [BS06], UML models [Ste11], and databases [CDPV07], among others [MT04]. With this increase in popularity, the meaning of "refactoring"

has become somewhat vague. This section provides definitions for the important terms used in this thesis and discusses some of the important issues pertaining to refactoring class structures.

3.1.1 Terminology

In this thesis, *restructuring* is the modification of software to make it easier to understand and change, or less susceptible to error when future changes are made. Refactoring is similar, but more restrictive. Rather than add to the confusion, this thesis will use Fowler, et al.'s [FBB+99] definitions:

Refactoring (noun) : a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactoring (verb) : to restructure software by applying a series of refactorings without changing its observable behavior.

In terms of maintaining behavior, the refactoring discussed in this thesis will only address black box functional behavior, i.e., a given set of input values should produce the same output values before and after refactoring. Our definition of refactoring is not concerned with maintaining other aspects of behavior that may be critical requirements for particular kinds of software, e.g., maintaining low execution times for real-time software.

This thesis makes an additional distinction between strict and loose refactoring, which is based on assumptions about potential client code. *Strict refactoring* means that all client code that worked before refactoring must also work after refactoring. In contrast, *loose refactoring* refers to code changes where known clients are not adversely affected by code changes, but unknown clients could be. In general, loose refactoring is used in situations where there are no unknown clients (or where potentially breaking client code is an acceptable risk).

Strict refactoring is appropriate for situations where released code supplies functionality for unknown clients. For example, object-oriented frameworks provide functionality to client programs via public classes, which imposes significant constraints on potential code changes. In particular, public and protected methods can not be removed or have their signatures altered unless all clients of that code can also be modified in the same refactoring operation; otherwise, the client code may not work as before.

Loose refactoring seems to be what is more commonly referred to in the refactoring literature. Indeed, many published refactorings [FBB+99, Ker05] are guaranteed to break unknown client code that depends on the maintenance of public interfaces, e.g., *Remove Parameter* or *Rename Method*. Loose refactoring depends on an assumption that all clients can be modified as part of the refactoring, as is the case for unreleased code or standalone programs. Section 3.1.3 includes an example of *Extract Class* that illustrates the differences between strict and loose refactoring.

Some researchers [MT04, PC09] use the terms *restructuring* and (loose) *refactoring* interchangeably, consistent with the opinion expressed by Mens and Tourwe [MT04] that "Refactoring is basically the object-oriented variant of restructuring". However, unless otherwise specified, our work on refactoring open source software employs strict refactoring, based on a lack of knowledge regarding potential clients of that code and the impact of potentially breaking client code.

3.1.2 Refactoring using automated tools

There are often alternative ways to refactor to fix a particular bad smell [PC09, FBB+99]. Generally, it is up to the programmer to decide which refactoring to apply, although some smell detectors make recommendations [FTSC11, SB10].

In some cases, while multiple refactorings might improve the system quality, none of them may be advisable. The costs of doing a refactoring can exceed the benefit. Even a seemingly simple renaming of a public method requires multiple checks. Programmers must determine whether the new name would cause a conflict with another in the same scope; they need to identify all clients of the method, so that the client method calls can be modified to reflect the new name, and they need to check subclasses of the modified class, so that overriding methods can also be modified, etc. There are also the costs of updating associated software artifacts, e.g., tests and documentation. There are risks involved with any software change.

Automated refactoring tools [Jem08, FTSC11, Wid06] lessen the risks of unintended consequences and can decrease the cost of maintenance by mechanically restructuring source code, given the appropriate inputs. The "automated" term is misleading, because the programmer still has the responsibility of deciding the inputs, i.e., the programmer needs to specify what should be changed. For

example, IntelliJ's [Jem08] implementation of the *Extract Class* refactoring requires programmers to specify which methods and attributes should be moved from the original class to the extracted class. The automated refactoring tools have the responsibility of seeing whether the refactoring can be made legally, e.g., whether the inputs provided by the programmer can produce a legal refactoring given the rules of the programming language. This typically requires extensive checking of preconditions before the restructuring takes place [KS08, Wid06, AEF08, Jem08]. After checking preconditions, the tool performs the code transformations that effect the refactoring, including changes to all client classes of the class being refactored.

Correctly restructuring classes semi-automatically is difficult [MS98, SEM08, CCS10]. Existing tools do not guarantee correctness. For example, Abadi, et al. [AEF08] note a number of difficulties with Eclipse's implementation of the *Extract Method* refactoring. *Extract Class* is considerably harder. Nevertheless, these tools are great aids to programmers modifying code, because they perform most of the tedious consistency checking that used to be done by programmers. Meanwhile, there is ongoing research [BM06, CCS10] on making refactoring provably correct.

3.1.3 Refactoring class structures

Many refactorings modify the structure of object-oriented classes – *Move Method*, *Extract Class*, etc. [FBB+99]. The common theme of the refactorings discussed in this section is that closely associated attributes and methods get put together in the same class.

Move Method and Move Field

The *Move Method* refactoring [FBB+99] moves a method from one class to another. Often, methods are moved when the method is using significant functionality in another class, so the method is moved to that class. Similarly, *Move Field* [FBB+99] moves an attribute from one class to another. This is often done when an attribute is used more heavily by a class other than the one in which it is defined. Alternatively, methods and fields may be moved because a programmer feels they are more conceptually related to one class than another. Both *Move Method* and *Move Field* typically decrease coupling between the two classes involved in the refactoring.

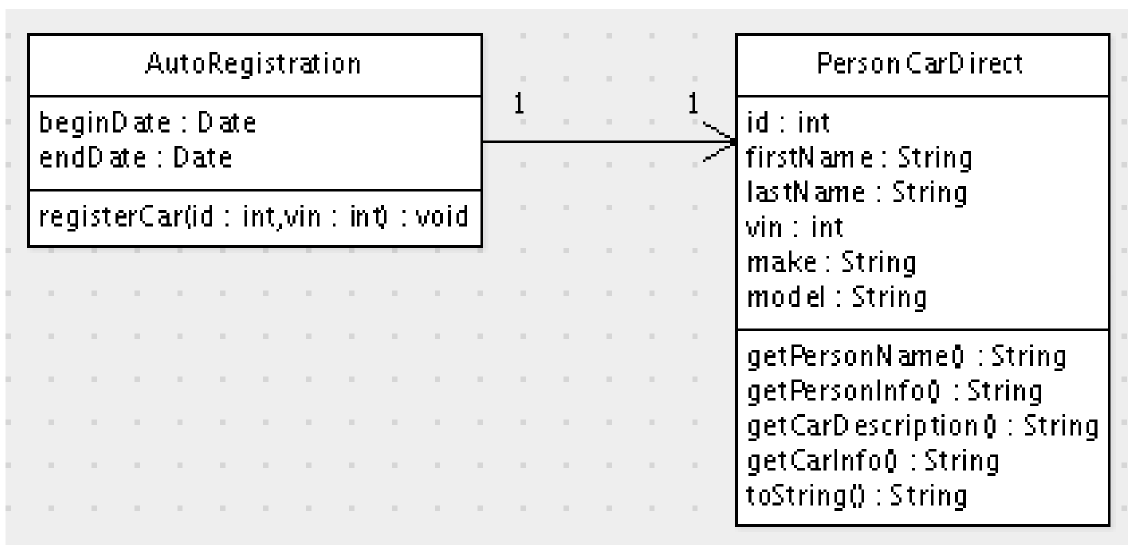


Figure 3.1: Client with original PersonCarDirect

As an example, consider a situation where an `AutoRegistration` class uses a `PersonCarDirect` class to store information about persons and cars, as shown in Figure 3.1. This figure shows the important methods and attributes of the classes, but omits the accessors (`get-` and `set-` methods) to avoid clutter. Figure 3.2 contains some of the code from `AutoRegistration`, including two of its calls to methods of `PersonCarDirect`, namely `setId` and `setVin`. Suppose there is also an `Automobile` class. Because a vehicle identification number is an attribute of an automobile, a programmer might decide to move `vin` and its accessors from `PersonCarDirect` to `Automobile`, because he thinks that is a better conceptual fit.

When a class member is moved from one class to another, all accesses to that member need to be updated to reflect its new class. If all client classes are known, e.g., `AutoRegistration` is in the same project as `PersonCarDirect`, then a loose refactoring is feasible, and there is a simple update to the client class (`AutoRegistration`) to call the moved `setVin` method, as shown in Figure 3.3.

If all client classes are not known, e.g., `AutoRegistration` is in a different project than `PersonCarDirect`, then a strict refactoring is necessary. `Auto-Registration` can not be updated; it is the "unknown" client, and it calls `PersonCarDirect`'s `setVin` method. Consequently, `PersonCarDirect` must continue to have a `setVin` method. With strict refactoring, the moving of a class member can be handled via delegation, as shown in the code listing of Figure 3.4. `PersonCarDirect`'s `setVin` method now calls `setVin` on an

```
public class AutoRegistration {
  public void registerCar(int ssn, int vin) {
    // ...
    PersonCarDirect personCar = new PersonCarDirect();
    personCar.setId(ssn);
    personCar.setVin(vin);
    // ...
  }
}
```

Figure 3.2: Original AutoRegistration source code

```
public class AutoRegistration {
  public void registerCar(int ssn, int vin) {
    // ...
    PersonCarDirect personCar = new PersonCarDirect();
    personCar.setId(ssn);
    Automobile auto = new Automobile();
    auto.setVin(vin);
    // ...
  }
}
```

Figure 3.3: AutoRegistration.registerCar after a loose Move Method

associated `Automobile` object. This avoids breaking `AutoRegistration`, but adds coupling between `AutoRegistration` and `Automobile`.

The *Move Field* refactoring is much the same as *Move Method*, with one exception. It is common practice for programmers to make their attributes either private or protected. Consequently, when an attribute is moved from one class to another, it may be necessary either to relax the access rights on the attribute, or to make sure there are available accessors for the attribute in its new class. Due to such considerations, refactorings may affect software metrics in ways that at first seem surprising. For example, a *Move Field* refactoring may cause an increase in the number of methods in the software system, due to the necessity of creating

```
public class PersonCar {
  private Automobile m_auto;
  //...
  public void setVin(int vin) {
    m_auto.setVin(vin);
  }
}
```

Figure 3.4: PersonCar.setVin after a strict Move Method

additional accessors and/or delegation methods.

Extract Class

The *Extract Class* refactoring splits a large class that has too many responsibilities into two smaller classes with more focused responsibilities, so an *Extract Class* refactoring should improve the average cohesion of the system. We refer to the class that gets refactored as the *original class*, the post-refactoring class that is most like the original class as the *modified class*, and the other post-refactoring class as the *extracted class*.

The `PersonCarDirect` class, illustrated in Figure 3.1 and described in Section 2.3.1, combines the traits of a person with that of a car. Because it represents two concepts, it can be split into two classes using the *Extract Class* refactoring.

If there is a situation where all clients of `PersonCarDirect` are known, e.g., all code is part of an unreleased prototype, then we can perform a loose refactoring. In the loose refactoring, `PersonCarDirect` is split into cohesive `Person` and `Car` classes, and the offensive `PersonCarDirect` can be removed. All known clients, e.g., the `AutoRegistration` code in Figure 3.5, are modified to access `Person` and `Car` directly. The resultant classes might look like those shown in Figure 3.6.

If there is a situation where some clients of `PersonCarDirect` are not known, e.g., the `PersonCarDirect` code is part of a released framework, then it is reckless to perform a loose refactoring, because the unknown clients of `Person-CarDirect` can not be modified to use the `Person` and `Car` classes that replaced it, and they will no longer compile.

In the strict refactoring, `PersonCarDirect` is split into a modified `Per-`

```java
public class AutoRegistration {
  public void registerCar(int ssn, int vin) {
    // ...
    Person person = new Person();
    person.setId(ssn);
    Car car = new Car();
    car.setVin(vin);
    // ...
  }
}
```

Figure 3.5: AutoRegistration source code after loose refactoring

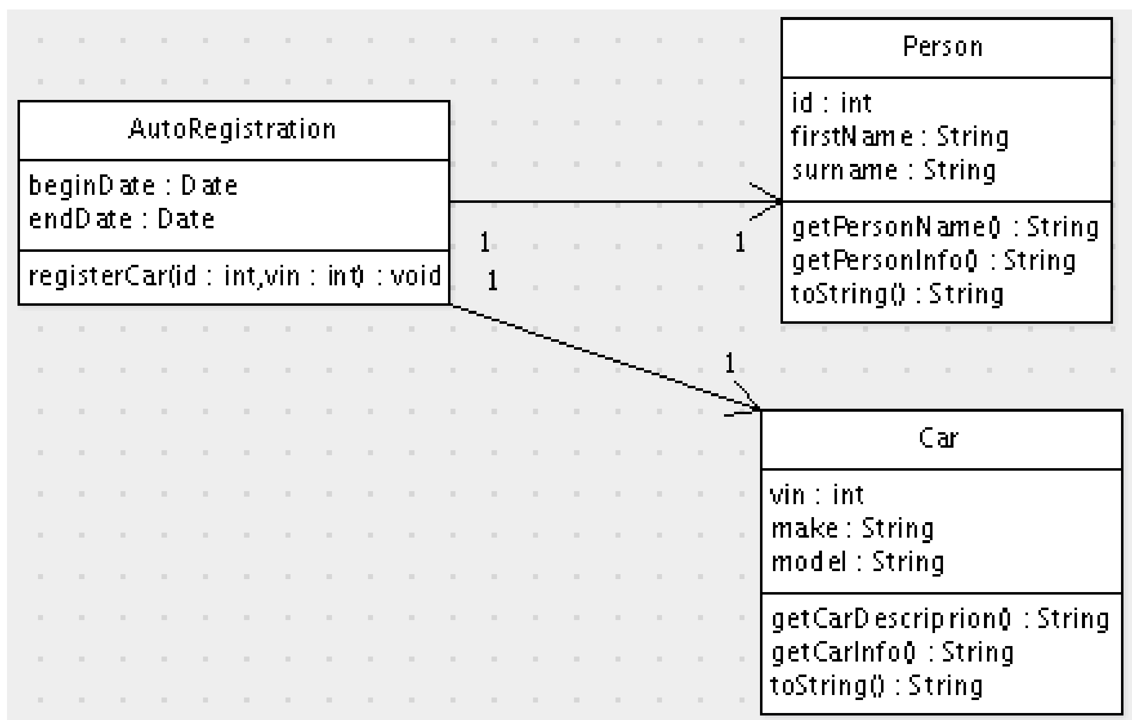


Figure 3.6: Client with PersonCarDirect after loose refactoring

sonCarDirect class and an associated Car class. Figure 3.7 shows the UML after a strict refactoring of the PersonCarDirect class. A modified version of the PersonCarDirect still exists, because of the requirement to maintain the original interfaces. The logic of the original car methods has moved to the extracted Car class, while the PersonCarDirect class has proxy methods that forward calls to the corresponding Car methods. This refactoring approach avoids potentially breaking an external client's code. Any pre-existing code that used the original PersonCarDirect still works after the refactoring. Known clients can be refactored to use the extracted Car class; however, unknown clients continue to use the modified PersonCarDirect as they did before the refactoring.

The use of strict refactoring means that the original class's dual nature persists, which has negative effects on system quality as measured by certain metric values. The overall number of fields in the system generally increases, because the modified classes may include new fields that point to the extracted classes. The overall number of methods generally increases, because of additional accessors, and because the modified classes maintain the original public interfaces of the classes by using proxy methods that delegate to the corresponding methods in the extracted classes. To remove this duality, and improve the system quality, requires some administrative coordination with clients, e.g., through the use of a *deprecation* process [Fow02, ZHR+06]. Fortunately, there is research underway [DJ06, cRGA08] to determine how to provide automated assistance to clients to adjust APIs that were changed via refactoring.

3.2 Clustering

Clustering algorithms put entities into groups (or clusters), where the members of a cluster are somehow related to each other. Highly related or similar entities should be put into the same cluster. The criteria that determine whether entities are highly related or similar is determined by the analysts performing the clustering.

There are many different clustering techniques described in the literature [JMF99, Ber02, Sch07, New10]. Berkhin [Ber02], for example, lists over 20 categories and subcategories of clustering algorithms. Each algorithm has its own strengths, and because they have distinct ways of operating, different algorithms may produce different results with the same data set. (In fact, some algorithms are stochastic [JMF99, HW79], so those algorithms may produce different results on

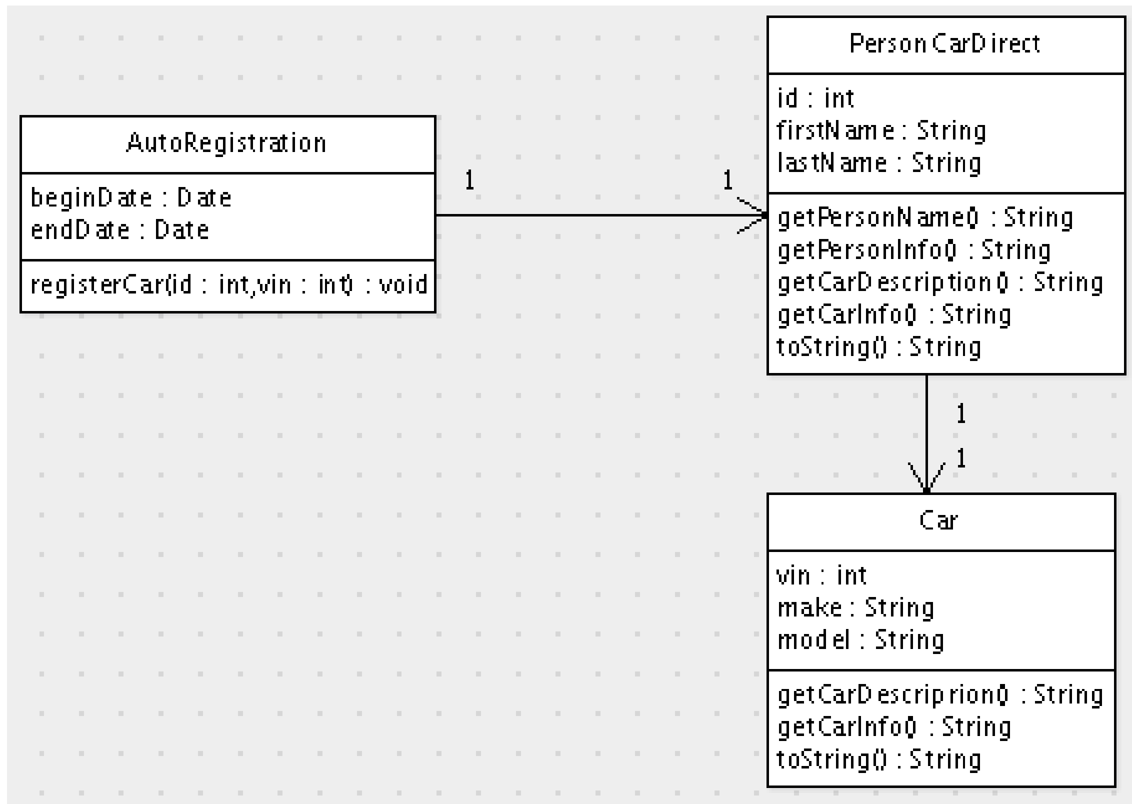


Figure 3.7: Client with PersonCarDirect after strict refactoring

```
public class PersonCar {
  // ...
  public void setVin(int vin) {
    m_car.setVin(vin);
  }
}
```

Figure 3.8: PersonCar.setVin after strict refactoring

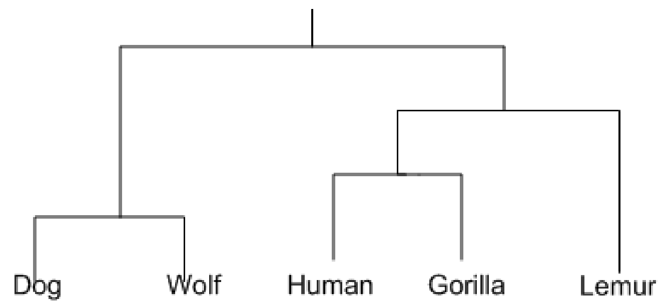


Figure 3.9: Clustering of mammals

consecutive runs using the same input data.) This section discusses some common themes of clustering. In chapters 5-7, we discuss specific clustering techniques, and how they apply to refactoring classes.

It is important to choose an appropriate clustering algorithm. It is also important to provide the clustering algorithms the appropriate entity attributes and relationships that enable the algorithm to form useful clusters. For example, morphological similarity might be used to determine whether two animals belong in the same cluster. Physical proximity might be used to determine clusters in a geographical application. For clustering web pages, it might be the pattern of hyperlinks connecting web pages that is important. One of the objectives of our research is to identify which clustering techniques and which relationships are useful for determining how classes should be refactored.

Whether entities should be put together in a cluster is often determined by analyzing the results of a *distance function*, which computes the distance between two entities based on the attributes and relationships of the entities. A good distance function should indicate that two similar entities are nearer to each other than two dissimilar entities. One of the challenges for the designers of clusterers is creating an effective distance function when the entities which will form the clusters are of different types, with different characteristics. For example, Java methods and attributes have some characteristics in common, while others differ.

Some clusterers are hierarchical; they identify clusters that are composed of more highly related subclusters. *Agglomerative* clustering produces hierarchical clusters by iteratively merging smaller clusters into larger ones, whereas *divisive* clustering produces hierarchical clusters by breaking up larger clusters into smaller ones.

Figure 3.9 shows an example of a hierarchical clustering of five animals, created using a distance function based on genetic similarity. The tree structure represents

the hierarchical clusters. The more similar animals are clustered together towards the bottom of the diagram, e.g., the dog and wolf are most similar, and the human and gorilla are the next most similar. The lemur is somewhat similar to the human and gorilla, so it is put in a cluster with the subcluster containing human and gorilla. The topmost, most general cluster combines the subcluster with the dog and wolf with the subcluster with the human, gorilla, and lemur. For some domains, hierarchical clusters give insight into the fundamental hierarchical structure of the domain, e.g., in the animal example, the levels of clustering may indicate important phylogenetic categories. In other domains, the nested clusters may not be significant, yet prove to be convenient for an analyst. For example, one can break up the tree into arbitrarily many clusters by slicing it at a given level (equivalent to cutting the tree structure in Figure 3.9 by drawing a horizontal line across it). Section 5.1.3 provides a detailed discussion of some agglomerative clustering algorithms that have been applied to refactoring classes.

Not all clusterers are based on explicit similarity or distance functions. For some domains, entities are considered part of the same group based on connectivity information (e.g., social networks, web pages). For these domains, graphs provide a convenient abstraction, based on their explicit representation of connections. Nodes that are highly connected may be clustered together. For example, imagine a graph representing a citation network, where the nodes represented researchers, and the edges represented one researcher citing another. The nodes representing researchers in any particular field, e.g., biology or physics, would be relatively highly interconnected compared to the number of connections between researchers in different fields. Certain graph-based divisive clustering algorithms use information about relative connectivity to split graphs and create clusters. Section 6.1 provides a detailed discussion of the graph-based divisive clustering algorithms that have been applied to refactoring classes.

3.2.1 Refactoring based on clustering

Because clustering techniques are useful for identifying entities that belong together, they have been applied to identifying software subsystems (e.g., the files that should be in the same package) and to modifying the module structure of software systems [AFL99, KE00, Lak97, MM06, Wig97]. Recently, a number of researchers [BDLO11, FTSC11, CC08, PLM+09, SC08] have begun using clustering techniques to determine how to refactor classes.

For the task of refactoring classes, clustering algorithms group together entities representing attributes and methods, and the resulting clusters indicate which methods and attributes belong in the revised classes. For example, a programmer might decide how to perform an *Extract Class* refactoring by forming clusters of a large class's members, and using these clusters as the basis for the refactored classes. Identifying clusters of methods and attributes that belong together in classes is a similar problem to the subsystem identification problem, but harder. To refactor classes requires the clustering techniques to be able to group disparate entities (e.g., methods and attributes), while maintaining certain relationships between the members, e.g., methods in an interface being kept together.

It is not clear which properties of object-oriented classes and their members are most suitable for using clustering as a basis for refactoring. Some properties are common to multiple kinds of entities, e.g., visibilities (`public`, `protected`, etc.). Other properties are unique to a particular kind of entity, e.g., only methods have return types. Important relationships in the object-oriented domain include the *calling* relationship (i.e. which methods call which other methods) and the class inheritance relationship. Some properties of an entity are more important than others for determining which entities belong together. For example, clustering together methods with the same visibilities would likely result in a poor class design.

The software characteristics measured by software metrics, especially those considered by cohesion and coupling metrics, give insight on the characteristics of classes that may provide a useful basis for clustering. As discussed in Chapter 2, many cohesion metrics consider the structural relationships within a class, e.g., which methods are called by other methods and which attributes are accessed by methods.

Structural information can be represented in a variety of ways. Some researchers [BDLO11, FTSC11, SC08] represent structural information as property sets and use distance functions to calculate similarity between the members of classes. Other researchers [PLM+09, CAGN09] use graph-based representations and clustering techniques. Chapters 5 and 6 discuss the strengths and weaknesses of these approaches.

Other cohesion and coupling metrics consider semantic or conceptual information embedded in identifiers and comments. This information can be exploited by agglomerative clustering algorithms, as discussed in Section 5.3.2.

3.2.2 Evaluating the results of refactoring based on clustering

We are interested in determining the relative utility of various clustering algorithms for determining how to refactor classes. Software metrics can help indicate whether refactoring has improved the quality of software, but they do not indicate whether the refactoring was the "optimal" refactoring.

It is possible for a programmer to perform a refactoring that improves certain metric scores, but replaces one software maintenance problem with another. For example, a programmer can eliminate a god class smell by breaking up a large, noncohesive class into small, cohesive classes. However, the small, cohesive classes might contain little functionality (the *lazy class* smell [FBB+99]). For the `PersonCarDirect` example introduced in Section 2.3.1, extracting a cohesive class consisting of `id`, `setId`, and `getId` is probably not desirable, because such a class does little – it only provides access to a simple data value. (We are not aware of any studies that rate the relative undesirability of various bad smells, e.g., god classes vs. lazy classes.) Some tools [CAG11, BDLMO10a], including ours (see Chapter 4), have built-in checks to help prevent the introduction of lazy classes when refactoring large classes; however, others do not [Fok10, FTCS09, SC08].

Determining a standard

A well-designed test suite helps analysts understand the applicability of algorithms to an intended domain. Given a standard set of inputs with expected outputs, an analyst can test algorithms to determine their fitness for a given purpose. For evaluating clustering algorithms, researchers frequently apply their algorithms to a known input data set and compare their results to some predetermined preferred clusters [Sht10, WT04, ST09, KE00, TH99]. The preferred output clusters are known as the "gold standard".

In 2000, Koschke and Eisenbarth [KE00] discussed the need for having a reference corpus of software systems to help determine the efficacy of clustering techniques on software. As far as we know, there are no such corpora for Java systems. To help address this need, we built an initial test suite [CAGN10] for use in analyzing the results of *Extract Class* refactorings. This test suite provides a simple basis for evaluating clustering outputs and facilitates comparison of different algorithms. Evaluating algorithms relative to a known test suite indicates how well the algorithms work for those data sets, but may not indicate how well the algorithm works on different data sets. Our test suite is intended to provide a

noncontroversial baseline, and will need to expand over time.

It takes effort to create a good test suite from scratch. Several researchers have taken a different approach, where they choose a pre-existing, well-designed system (often JHotDraw [GE07]) to serve as the output gold standard. From this gold standard output, they create a set of inputs from which the tested algorithms should reproduce the gold standard output. There are two main variations to this approach used by researchers on refactoring based on clustering. The first variation [SC07, CS06] disassembles the gold standard system's classes into attributes and methods. The researchers then determine whether the clustering algorithms can create clusters equivalent to the original classes. The second variation [BDLMO10a, DLOV08] creates a poorly designed, "mutant" system by programmatically rearranging the contents of the gold standard system. The researchers then determine whether the algorithms they are testing can take the mutant system as input and produce the clusters that correspond to the original classes.

While the creation of test inputs from the gold standard output can be a useful technique, it can also lead to tainted results. The developer of the test inputs must take care that the formation of those inputs does not bias the tests relative to the algorithm being tested. For example, when the gold standard is broken up into its constituent attributes and methods, if the information stored with the attributes and methods includes information about the original class, as it does for the techniques [SC07, CS06] discussed in Section 5.2.2, it is not surprising that many of the original classes get recreated. The creation of a "mutant" system can suffer from a different problem. Depending on how it is created, a "mutant" system may present unrealistically simple cases to refactor (akin to refactoring many `PersonCarDisjoint`s).

Comparison to a standard

Comparing results to a gold standard can be nontrivial. Even when the gold standard is the "best" design, it may not be the only possible "good" design. In this situation, it may not be necessary for the clusters produced by an algorithm to exactly match the gold standard. There are many ways that entities can be organized into clusters, so there may be acceptable alternative clusterings for a given set of inputs [ST11]. This is particularly true of clusters produced from classes composed of hundreds of members. Among the class members, it is

more important for some members to be in the same class than others – some methods provide central functionality, while others provide peripheral or auxiliary functionality. Some clusterings may not be ideal, but may still be acceptable as the basis for refactored classes. Consequently, there is a need for an algorithm for evaluating the similarity between a proposed clustering and the gold standard.

There are a variety of algorithms available for measuring the similarity between two clusterings. Some of these cluster comparison algorithms are fairly generic and use little software-specific knowledge ([TH99, WT04, ST09]), while others are geared towards a particular software development task [MM01]. Those cluster comparison algorithms that are geared towards software development face many of the same issues that cropped up with measuring cohesion. In the context of clustering to identify subsystems, Mitchell and Mancoridis [MM01] pointed out that some entities require special treatment. They recommend excluding "special" modules, including modules that are highly interconnected to others, and modules that with a high in-degree and a zero out-degree. This is a similar idea to the exclusion of special methods while calculating cohesion measurements, as discussed in Section 2.1.3.

The research in this thesis does not make use of these cluster comparison functions, because there is no suitable gold standard for which they would be applicable. The test suite that serves as the gold standard for some of our experiments has non-ambiguous preferred clusterings that provides a clear-cut pass/fail evaluation of clustering algorithms. Future work may require more subtle tests, and the choice of cluster comparison algorithms will be looked at more closely.

3.3 Summary

This chapter described the Move Method, Move Field, and Extract Class refactorings and issues related to them. It then discussed how clustering techniques can help programmers determine how to refactor, and how the results of clustering can be evaluated. Subsequent chapters will discuss the application of particular clustering techniques to refactoring classes.

Chapter 4

The Refactoring Environment

The Eclipse IDE [SDF+03] is an extensible Java development environment that includes code analysis, search, and refactoring APIs [Wid06]. However, Eclipse lacks some capabilities that can make refactoring object-oriented classes easier.

This chapter discusses my extensions to the Eclipse coding and refactoring environment to help investigate the use of clustering techniques for refactoring object-oriented classes. The ExtC (*Ext*ract Class) plug-in is useful for both programmers and researchers. For programmers, it helps identify problematic classes and provides detail on how to refactor those classes. Researchers have additional needs. For researchers, ExtC provides the means for investigating various clustering algorithms and provides insights into how those algorithms produce their results.

Figure 4.1 shows a high level view of the ExtC environment, including the roles of some of the more important plug-ins. The metrics plug-in, metrics database, and database plug-in provide the means of obtaining, saving, and accessing metric data for Java software. The tables of the database can be populated by multiple sources. The metrics plug-in role is primarily filled by Metrics2 [SB10], which communicates with a SQL metrics database using a database plug-in provided by Apache's Derby project [Sch08]. While Metrics2 provides a means of gathering structural measurements, it does not provide a way of measuring semantic cohesion. ExtC provides a function that measures the semantic cohesion (C3V) of the classes and stores it to the database. This metric data is used by the ExtC plug-in to help identify classes that violate metric guidelines and to help evaluate the results of refactoring. ExtC does not communicate directly with the metrics plug-in. It gets the metric data from the database, also using the Derby

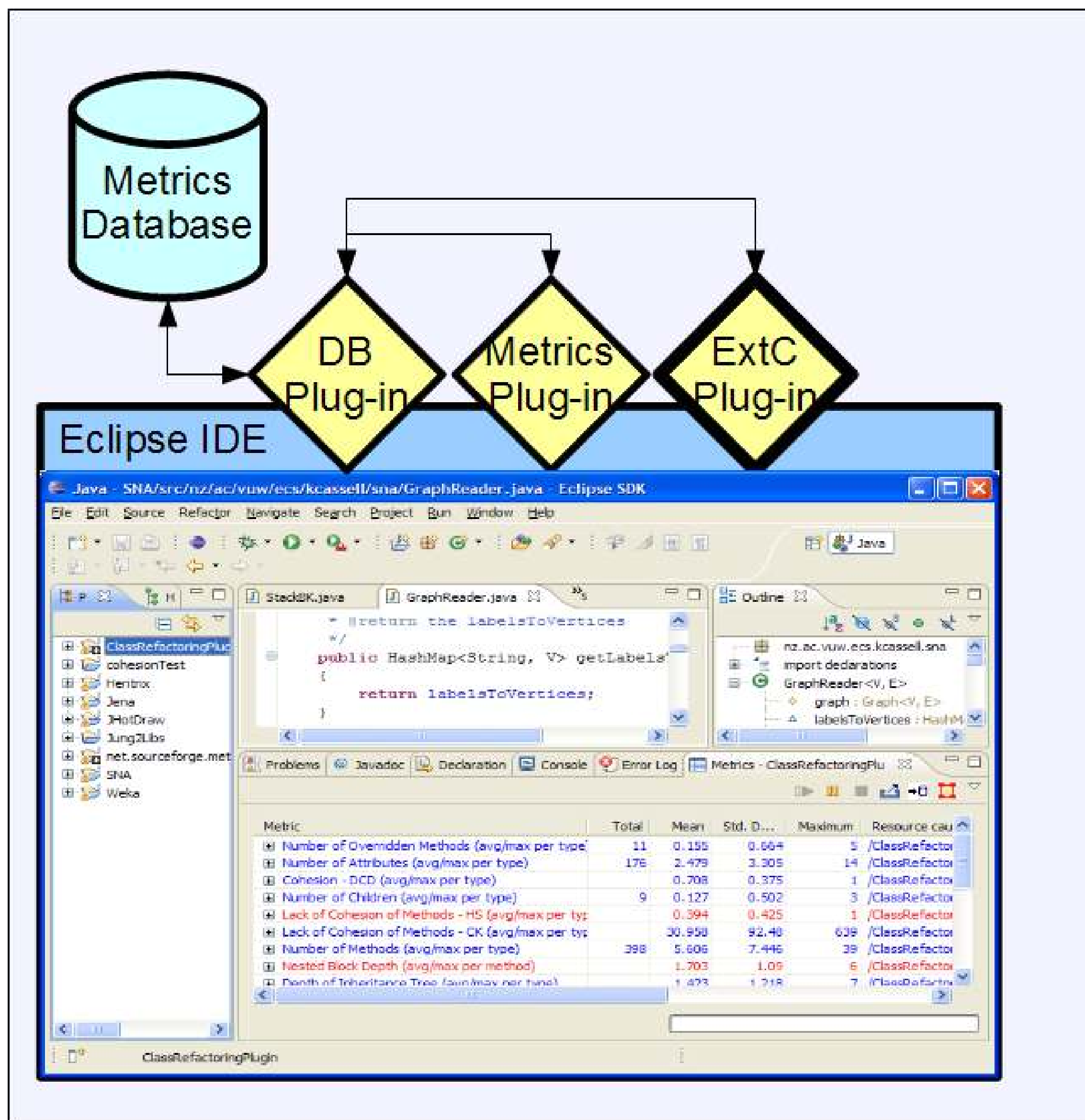


Figure 4.1: The ExtC refactoring environment

plug-in.

The ExtC plug-in also provides a graphical user interface (GUI) that helps programmers visualize Java classes; however, the main purpose of ExtC is to propose restructurings of Java classes based on clustering results. We are aware of no other refactoring environments that provide all of these capabilities. Appendix B.1 provides detail about the open source software used in our research, including version numbers, web sites, and available documentation.

4.1 Identifying problematic classes

ExtC provides users the capability of composing SQL queries to locate potentially problematic classes based on their metric values. Figure 4.2 shows a user interface containing a user-composed query and a table of matching classes and their metric values. From such tables, ExtC users can select classes for further processing, e.g., to inspect the class's code, to view its intraclass dependency graph, or to cluster its methods and attributes.

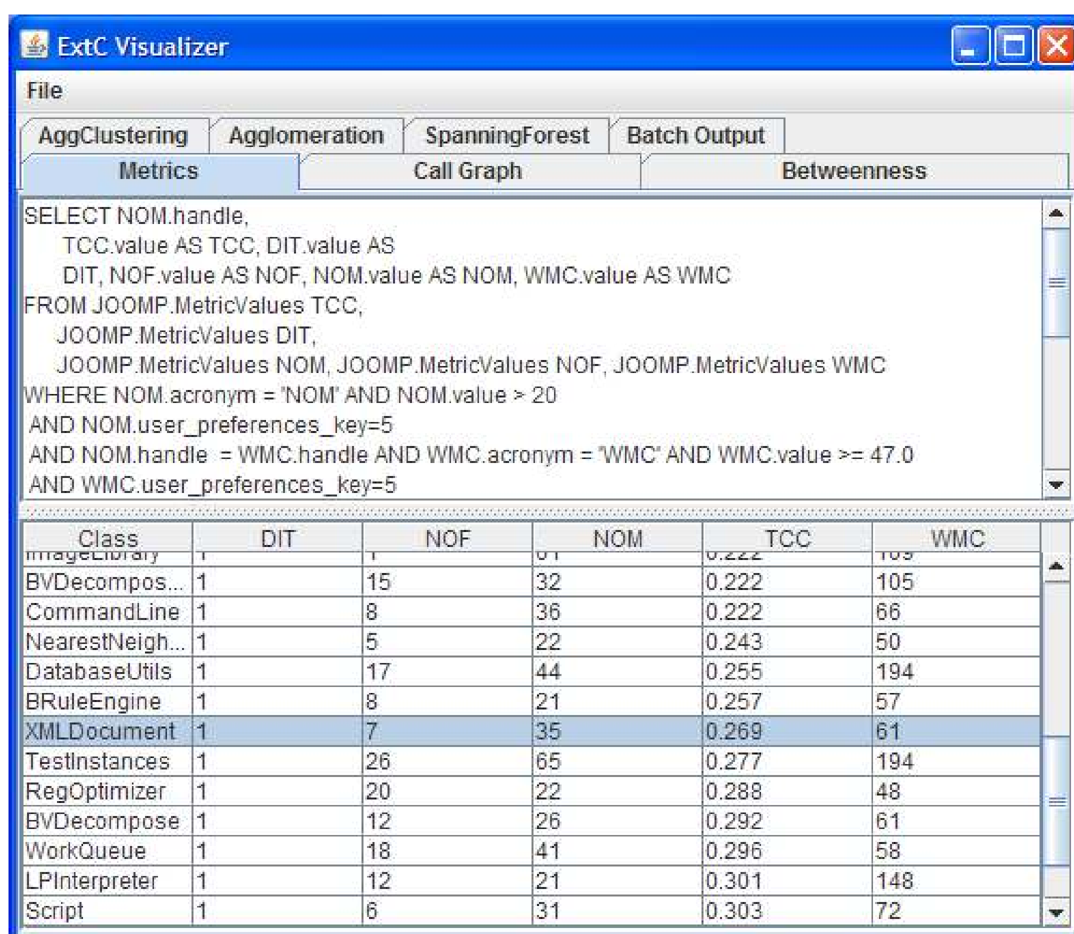


Figure 4.2: ExtC metrics view

As an example, several chapters in this thesis discuss experiments on large classes that came from open source Java projects. These classes were identified using a god class query that searched for classes meeting the following conditions:[1]

1. number of instance methods (NOM) > 20

[1]Appendix B.2 provides detail on the four open source projects (FreeCol, Heritrix, Jena, and

2. tight class cohesion (TCC) [BK95] < 0.34
3. weighted method count (WMC) [CK94] ≥ 47.0
4. depth in the inheritance hierarchy = 1.0

The rationale for these metric criteria is discussed in section 2.4.3. Analysts are not limited to using pre-built queries such as this; they can compose SQL queries to identify classes that are appropriate to their task.

4.2 Visualizing classes

Because it is tedious to determine the interrelationships present within large classes solely by reading code, ExtC provides visualizations of class structure that highlight key characteristics of class members and the relationships between them. The ExtC graph view (Figure 4.3) shows the intraclass dependency graphs of classes, where circular nodes represent methods, stars represent attributes, and the directed edges indicate methods calling methods or accessing attributes. Node colors indicate the accessibility of a class member (green, yellow, and red for public, protected, and private access, respectively), and the nodes' sizes can be made dependent on characteristics of the underlying class member through a menu choice.

The graph view provides options for modifying the graph display and for altering the graph structure. ExtC users can rearrange graphs by choosing a graph layout algorithm, or by repositioning nodes using the mouse. Some of the available graph layout algorithms are force-directed layouts that can be useful for visually spotting clusters [SSL01]. Some options affect the structure of the graph. The graph view provides check boxes that permit the analyst to optionally include nodes that represent members originally defined on `Object`, constructors, static members, inner class members, and loggers. It is also possible to "condense" nodes, for example, to group all methods inherited from `Object` into a single node. The provided graph transformations are those discussed in Section 2.4.2.

Weka) and the identified god classes. Out of approximately 3000 top level public classes, 30 classes (1%) matched the query.

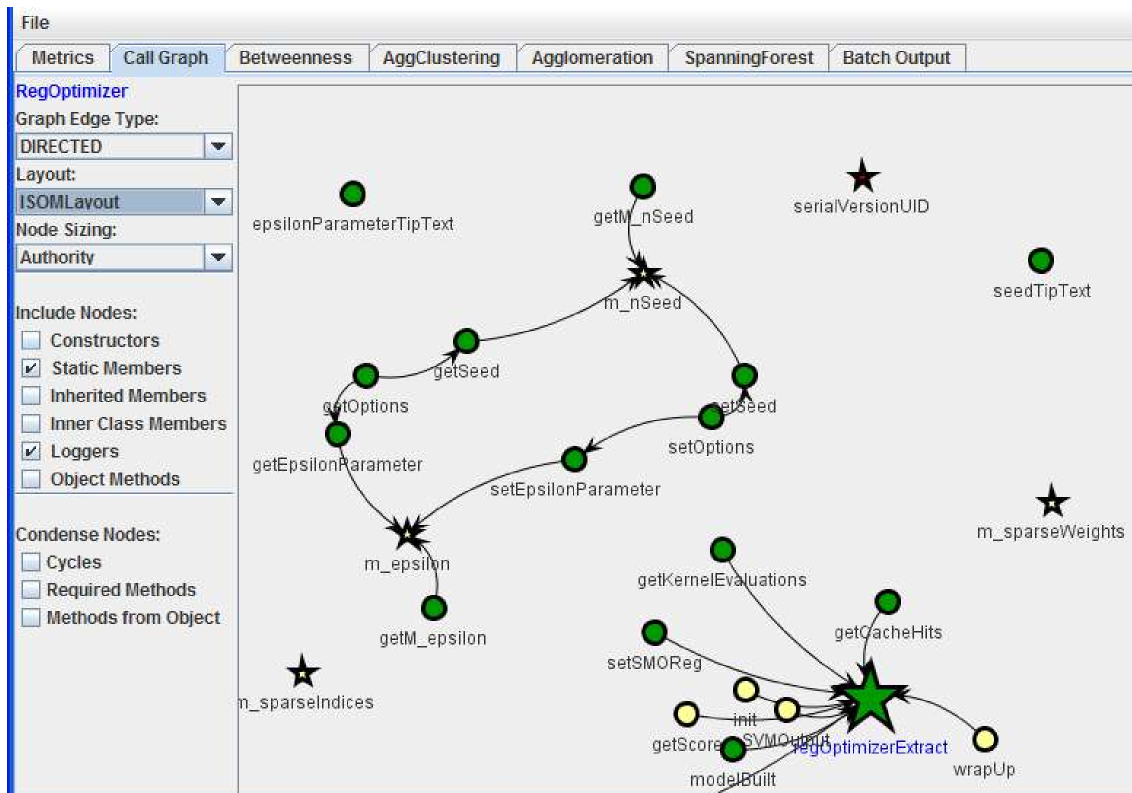


Figure 4.3: ExtC graph view

4.3 Proposing refactorings

Given a clustering algorithm and some input data, it is fairly easy to run the clustering algorithm on the input data to produce clusters. Many of the experiments described in later chapters do so, using ExtC's batch processing capabilities. The batch mode is activated by pushing any of several buttons, which typically causes one or more clustering operations to be run on selected classes and output on the resultant clusters to be dumped to a file. The drawback of this mode of operation is that it provides little insight into why the members were put into those clusters.

To help remedy this problem, ExtC provides three views that help researchers understand why class members are clustered together. The agglomeration view (Figure 4.4) provides a tree representation that shows the nested clusters produced by agglomerative clustering. The view provides menus for the specification of parameters to the clusterer, e.g., a user can select from alternative distance calculators to see how the choice of distance function affects the generated clusters.

Section 5.1.3 discusses hierarchical agglomerative clustering in detail.

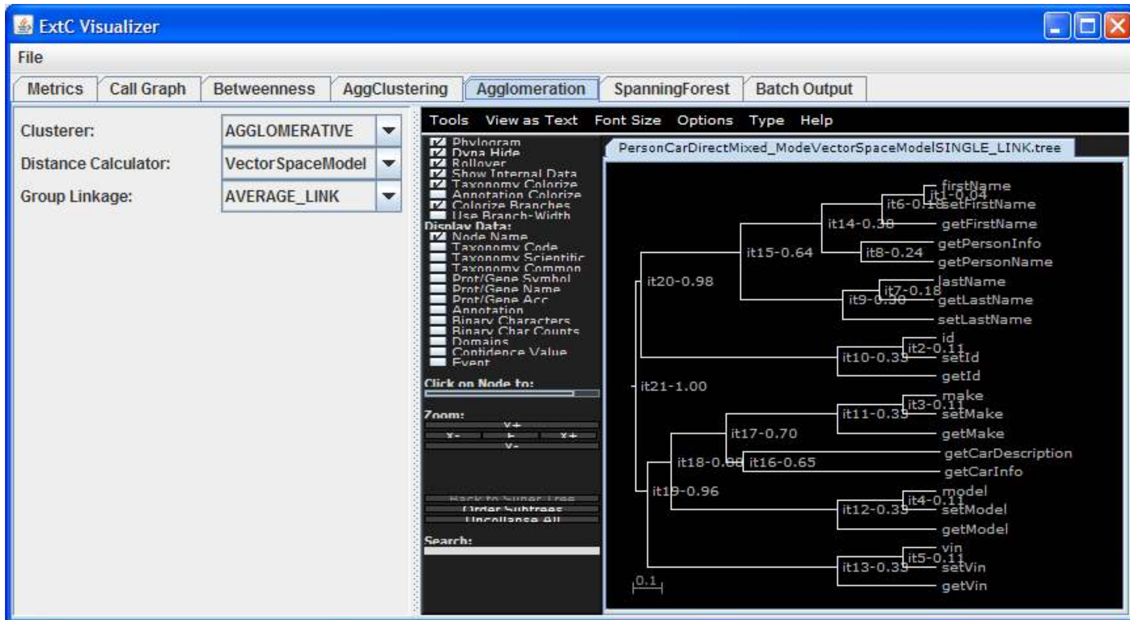


Figure 4.4: ExtC agglomeration view

There are also two interactive views that show clustering in action. The agglomerative clustering view provides the ability to step through agglomerative clustering, while the results are shown in the context of an intraclass dependency graph. Section 5.5 discusses this view in detail. The betweenness clustering view provides users the ability to see the behavior of the betweenness clustering algorithm as it splits an intraclass dependency graph. Section 6.3.1 discusses this view. Both of these interactive views help researchers determine whether the iterations in which members are joined (or separated) is consistent with their intuition.

The batch mode and the interactive views can all be used to generate clusters of class members. These clusters serve as inputs to the automated refactoring tool that will modify the code.

4.4 Performing refactorings

There are many options available when programmers perform certain refactorings, e.g., *Extract Class*. For example, when a programmer moves a private attribute from the original class to an extracted classes, but that attribute still need to be

accessible, he could make the attribute public, or he could create one or more accessors.

Automated refactoring reduces the variability of how classes are refactored, and makes evaluation and comparison of results more consistent. Our original intent was to have the output of the clustering step be usable as a programmatic input to Eclipse's *Extract Class* refactoring. Eclipse 3.6.2 offers an *Extract Class* refactoring; however, it is extremely limited. Eclipse's refactoring is restricted to transferring user-specified attributes to the extracted class; it does not offer the capability of transferring methods also.[2]

As far as we are aware, there are no publicly available programs that are highly effective for splitting large, complex classes. The IntelliJ IDEA version 8.1 development environment [Jem08] provides an *Extract Class* refactoring that transfers both user-specified attributes and methods. IDEA does the code manipulation necessary to form the new class and to modify other classes that might be affected by the refactoring; however, due to some defects, the classes it extracts will sometimes not compile. (The JDeodorant Eclipse plug-in[3] can extract classes, but it only does so given the inputs provided by JDeodorant's clustering [FTSC11]).

Our experiments described in Chapters 6 and 7 use the *Extract Class* capability provided by IntelliJ, whose refactoring approach is similar to the strict refactoring approach described in Section 3.1.3 and illustrated in Figure 3.7, but differs in its handling of static members. IntelliJ modifies client code to use any moved static members. IntelliJ's approach to performing Extract Class means that client code will remain as cohesive as it was prior to the refactoring, although coupling may increase if the client code used any static members that were moved to the extracted class.

There is no convenient programmatic way of sending the clustering results generated in the Eclipse environment to serve as the inputs to IntelliJ's *Extract Class*, so the inputs are entered manually, using the IntelliJ GUI. While the classes IntelliJ generates do not always compile, they can usually be fixed with a small amount of manual coding. The most common bugs pertain to methods in one class lacking access to another class's attributes. This can be fixed by increasing the attributes' visibility or by providing accessors. The main manual actions required to complete the refactoring are adding back in the features that were filtered out

[2]Eclipse bug 312347 – `https://bugs.eclipse.org/bugs/show_bug.cgi?id=312347`
[3]http://jdeodorant.com/

in earlier steps. For example, the initialization via constructors is filtered out to simplify the analysis, so the initialization within the refactored classes needs to be performed manually.

4.5 Contribution Summary

There are many software development environments available for use in developing Java code, and some of these provide capabilities for refactoring. However, we are unaware of any IDE that provides the combination of capabilities that ExtC adds to the Eclipse environment – identifying problematic classes, visualizing classes, visualizing the clustering of the classes' members, and proposing the reorganization of class members to facilitate the refactoring of problematic classes. ExtC is open source software.

Chapter 5

Refactoring Using Distance-Based Clustering Techniques

Many clustering techniques determine which entities belong in the same cluster based on the notion of distance between entities. This chapter discusses the use of distance-based clustering for determining how to refactor object-oriented classes. It begins with background material about representing domain entities and measuring distance, and then discusses two of the major distance-based families of clustering algorithms that have been applied to software refactoring – agglomerative clustering and partitional clustering. Section 5.2 discusses how other researchers have applied distance-based clustering to refactoring, including their choices of how to represent the object-oriented software that serves as input, and their algorithms for processing those inputs. Section 5.3 contains our evaluation of various distance-based clustering techniques using object-oriented classes from our test suite as inputs, while Section 5.4 examines the clustering techniques using open source classes as inputs. Following a discussion of our visualization of agglomerative clustering in Section 5.5, the chapter concludes with an evaluation of the distance-based techniques as applied to refactoring and a summary of our contributions.

5.1 Background – distance-based clustering

In terms of clustering, distance is an abstract concept. Distance is made concrete by analysts, who devise functions that calculate distances between entities, based on characteristics of those entities. The effectiveness of the clustering relative to

the analysts' needs is partially dependent upon the characteristics chosen by the analysts, and how the differences in those characteristics are mapped to numeric values. This section discusses issues related to entity representation, together with some popular representations and distance functions that use them, followed by a discussion of some of the more popular distance-based clustering techniques.

5.1.1 Entity representation

One of the key determinants to the success of clustering is the choice of information upon which the clustering algorithm operates. This section uses an example where the information that provides the basis for clustering the class members consists of the words that comprise their identifiers, e.g., the properties for a method named `getDatabaseValues` are the words `get`, `database`, and `values`. An advantage of this generic scheme is its usability for many entities that are dissimilar in many ways. This does not necessarily imply that it is useful for a particular purpose. (Counting the number of "e"s in the identifiers is also a generic scheme, but one that is not likely to be useful.) Section 5.2 discusses representations that have been used with clustering techniques and applied towards refactoring object-oriented classes.

After one determines the important entity information, it should be stored in a form that is amenable to computation. Two of the more popular representations for use in distance-based clustering are property sets and feature vectors.

Property sets

Property sets provide a flexible and concise way of storing unstructured information by grouping together the properties (information bits) in a set. In the example, the property set consists of the words that are part of the member identifier, e.g., the property set for the `getDatabaseValues` method is a set that contains the words `get`, `database`, and `values`.

Feature vectors

Feature vectors are more structured than property sets. In a feature vector representation, each entity (e.g., class member) has an associated vector, and each element of the vector consists of a value for a particular feature of the domain being

represented. All entities have this same vector representation, so the information stored at a given index is the same for all entities.

For the member identifier example, the vector might have an index for each distinct word that occurred in some identifier in the class. The value for that index might be the number of times that word occurred in the identifier. For `getDatabaseValues`, there might be "1" entries in the vector positions for `get`, `database`, and `values` and "0" entries for the words that were present in other identifiers, but not in `getDatabaseValues`.

5.1.2 Similarity and distance functions

Distance-based clustering techniques determine whether entities should be in the same cluster based on the results of a *similarity function* or *distance function*. Because a similarity function can be considered a kind of distance function (two similar things are less distant conceptually), this thesis will generally use the term "distance function". Some distance functions measure distances in a continuous multidimensional space. Entities exist at various coordinates in the space, and distances are typically measured using any of several "generic" distance measures, e.g., Euclidean distance or Manhattan distance [Ber02]. Other distance functions are independent of any underlying spatial representation, and may be highly specialized to calculate distances between two entities based on their characteristics. The remainder of this section describes distance and similarity functions that have been used for clustering based on the characteristics of software entities as contained in property sets or feature vectors.

Jaccard similarity and distance

The *Jaccard similarity* measure [SGM00] is often used to compute the similarity of pairs of property sets. For a given two entities, the Jaccard similarity is the size of the intersection of their properties divided by the size of the union of their properties (or zero in the unusual case of both entities having no properties). Consequently, the Jaccard similarity ranges from 0 for the most dissimilar entities to 1 for the most similar entities. The *Jaccard distance* is one minus the Jaccard similarity.

In our example, the Jaccard similarity is the number of words two identifiers have in common divided by the total number of distinct words in the two identifiers. Therefore, the Jaccard similarity for `getDatabaseValues` and

`saveDatabaseValues` is 0.5, because there are two common words, `database` and `values`, out of a total of four different words. The Jaccard distance is also 0.5, because $1.0 - 0.5 = 0.5$.

Euclidean distance

The Euclidean distance is the "straight-line" distance between two points in space. For two vectors, $\vec{v_a}$ and $\vec{v_b}$:

$$dist(\vec{v_a}, \vec{v_b}) = \sqrt{\sum_i (a_i - b_i)^2} \tag{5.1}$$

where a_i is the number at index `i` for vector `a`. Because our example is not space-based, Euclidean distance is not applicable.

Cosine similarity and distance

The cosine similarity is useful for comparing two feature vectors when all of the features have numeric values. For two vectors, $\vec{v_a}$ and $\vec{v_b}$:

$$cos(\vec{v_a}, \vec{v_b}) = \frac{\vec{v_a} \cdot \vec{v_b}}{\|\vec{v_a}\|\|\vec{v_b}\|} \tag{5.2}$$

where $\|\vec{v}\|$ represents the Euclidean norm of a vector. The cosine similarity ranges from 0 to 1, with values increasing as more terms are shared. The *cosine distance* is one minus the cosine similarity. For our example, the cosine similarity is 0.67 and the cosine distance = 0.33.

5.1.3 Agglomerative clustering

Agglomerative clustering is an iterative process of combining clusters into larger clusters based on a distance function. Often, the initial clusters consist of single entities. The clustering starts by combining those clusters that are closest together and proceeds to combine more distant clusters. This section gives a brief overview of generic agglomerative clustering issues. There are many sources available that provide a more detailed picture, e.g. [Ber02, JMF99, WFH11].

For agglomerative clustering to be effective, several questions need to be answered, including:

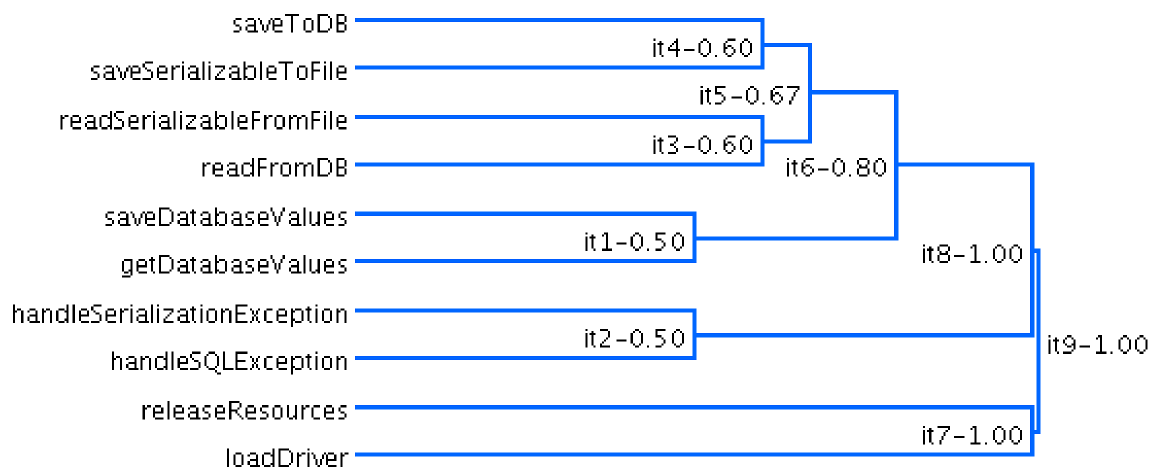
- Representation – What information about the entities is important for determining whether they should be in the same clusters, and how should this information be stored?

- Entity comparison – What determines when individual entities should be combined?
- Group comparison – What determines when groups of entities should be combined?
- Completion – Which collection of clusters is the most desirable one, i.e., when should the iterative clustering stop? (A single cluster containing everything is seldom the desired outcome.)

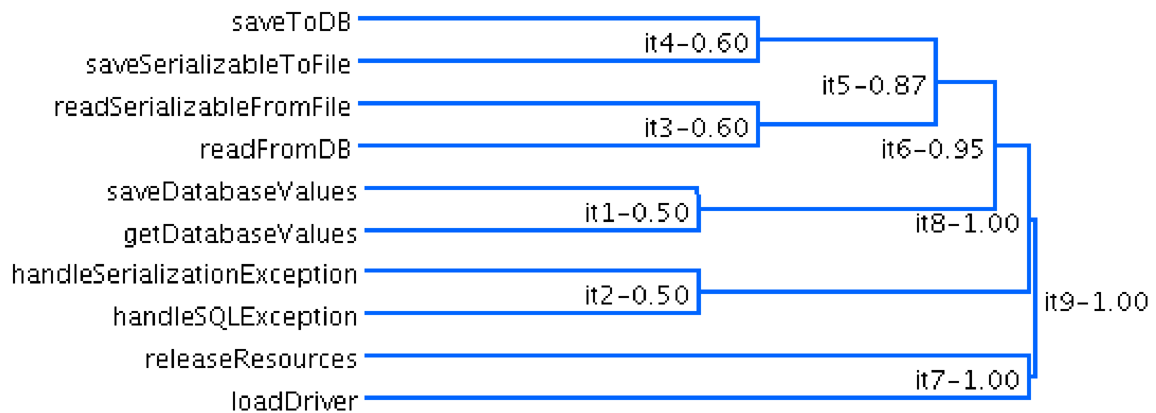
This section discusses these questions in the context of clustering the members of a test class, `AnonymousPersistence`, whose source code can be found in Appendix A.1.

Representing nested clusters as dendrograms

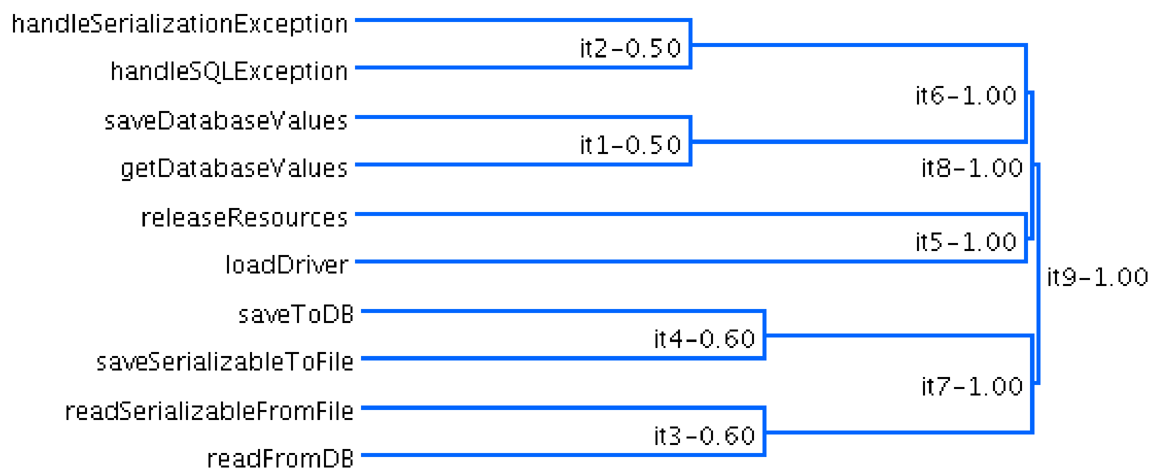
Agglomerative clustering generally uses distance functions to determine when clusters should be combined. The result of agglomerative clustering is often displayed as a *dendrogram*, a tree structure that indicates when clusters get combined into bigger clusters. Figure 5.1(a) shows a dendrogram that represents agglomerative clustering using a Jaccard distance based on the words in the member identifiers. The x-axis represents distance, generally ranging from 0.0 to 1.0. The y-axis merely provides space to distribute the initial clusters, which will often consist of single individuals. Horizontal lines extend to the right of each cluster. Vertical lines indicate when clusters get merged into a larger cluster. In this document, these larger clusters are generally labeled with the iteration in which the merger occurred and the distance between the merged clusters. Moving from the left, the first vertical line encountered connects the horizontal lines for `saveDatabaseValues` and `getDatabaseValues` clusters, indicating that these are the two clusters that are nearest to each other. The label next to the vertical line, "it1-0.50", indicates that these clusters got merged at the first iteration, and the distance between them was 0.5. `handleSerial-izableException` and `handleSQLException` are joined at the same level, because they also have a Jaccard distance of 0.5. Two more clusters form at a distance of 0.6 – `saveToDB` combines with `saveSerializableToFile` and `readSerializableFromFile` combines with `readFromDB`.



(a) Single link clustering



(b) Average link clustering



(c) Complete link clustering

Figure 5.1: Agglomerative clustering

Group comparison

Thus far, the example of Figure 5.1(a) has only shown the merging of clusters consisting of single entities. How to merge clusters consisting of multiple entities is less clear; there are multiple options. This section discusses three of the most popular ways of measuring the distance between multiple-entity clusters – single link, complete link, and average link clustering [JMF99].

Single link clustering merges clusters based on the nearest individuals (a.k.a. *nearest neighbors*) of the different clusters as computed by the distance function. For entities that can be represented spatially, it tends to produce non-compact, "stringy" clusters, where some entities may be much farther apart from entities in their own cluster than they are from some entities in different clusters.

The first merger of multiple entities clusters in Figure 5.1(a) is at `it5-0.67`. The `it4-0.60` and `it3-0.60` clusters are merged, because `saveSerial-izableToFile` and `readSerializableFromFile` share two out of six words for a Jaccard distance of 0.67. One more cluster is formed at a distance of 0.8, before the final clusters are formed at the maximal distance of 1.0.

Complete link clustering merges clusters based on the smallest distance between the most distant members of each cluster. For entities that can be represented spatially, it tends to produce compact clusters. Figure 5.1(c) shows the same members being clustered as in Figure 5.1(a), but using complete link clustering rather than single link. The initial four merged clusters are the same. However, after their formation, the distance between the most distant members of each pair of clusters is 1.0, so nothing else gets merged until the maximal distance value of 1.0.

Average link clustering combines clusters based on the average distance between the members of the two different clusters. It tends to produce clusters intermediate between single link and complete link clustering. Figure 5.1(b) shows the same members being clustered as in Figure 5.1(a), but using average link clustering rather than single link. In this example, the clusters produced by average link clustering are the same as those produced by single link clustering, but with the multi-member clusters being merged at a greater distance. The top four entities are merged at a distance of 0.67 for single link clustering, but at a distance of 0.87 for average link clustering. Although the pattern of formation of clusters is the same for single link and average link clustering in this example, this does not generally happen.

Result collection

The examples of Figure 5.1 do not illustrate when to stop clustering, i.e., they do not stop the clustering process when the "best" clusters have been formed. When to stop clustering is generally a domain-specific decision that is based on the number of clusters desired or on a quality criterion for the clusters. Without criteria regarding what constitutes a good cluster relative to a particular domain, there is no way of knowing whether any of the clusters formed are good relative to that domain.

In most potential class refactoring investigations, computational costs are not an issue, so it is convenient to cluster everything into a single nested cluster and examine the cluster formation history afterward in a search for the clusters that best match the quality criteria mentioned in Chapter 2. Because clusters are composed of subclusters, an analyst can examine intermediate results embedded in the dendrograms. One possibility is to examine all clusters that existed at a given iteration of the clustering. By drawing a vertical line through a dendrogram at a specific distance value, an analyst can identify the clusters that existed at that point in the clustering. For example, Figure 5.1(a) contains four clusters at a threshold of 0.9 – one cluster of six entities, one cluster of two entities and two clusters of a single entity. Figure 5.1(c), on the other hand, has six clusters at 0.9. However, an analyst does not need to be constrained to accept a given set of clusters produced at a particular distance, any subcluster may provide insight into a possible organization of entities.

5.1.4 Partitional clustering

Given a set of entities to cluster and an input `k`, partitional clustering algorithms split the set of input entities into `k` clusters. Unlike hierarchical algorithms, partitional clustering produces "flat" clusters, rather than nested clusters. Two well-known families of partitional clustering algorithms are k-means and k-medoids.

K-means

K-means [Har75, HW79, Ber02, JMF99] is an iterative process of choosing points in space to serve as nuclei for clusters and then clustering about those nuclei. K-means starts by choosing `k` points in space that will serve as the initial nuclei.

Depending on the variant of k-means being used, the initial nuclei can be chosen either randomly or using domain knowledge. The algorithm forms clusters by associating each entity with the nearest nucleus based on the distance (e.g., Euclidean or cosine distance) between them. K-means then computes the mean location (a.k.a. *centroid*) for each of the clusters. These centroids serve as the nuclei for the next iteration of clustering. This process repeats until it reaches either a stable state or a threshold of iterations.

K-means is fast, simple, and generally effective. It is an efficient algorithm, so it can be used on data sets of millions of nodes [FLE00]. Because the clusters are formed around the centroids, k-means produces compact clusters that locally minimize the total squared distance of the cluster's entities to the cluster center.

K-means has limitations. One of the reasons it is fast is because it is a heuristic solution. The quality of the solution depends in large part on the selection of the initial nuclei chosen for the clusters. An optimal solution is not guaranteed, especially when outlying entities are chosen as initial nuclei. K-means also tends to produce compact clusters of roughly the same diameter; it is not good at detecting clusters of different diameters or long, stringy clusters. This can be a problem for many domains.

Figure 5.2[1] illustrates this problem in two-dimensional space. The entities in Figure 5.2(a) are divided into the preferred set of three circular clusters in a Mickey Mouse-like arrangement[2] – a large "head" cluster (labeled "C") and two smaller "ear" clusters (labeled "A" and "B"). For a `k` of 3, k-means can create clusters like those in Figure 5.2(b). Because k-means tends to make clusters of similar diameters, the "ear" clusters produced by k-means include entities that were part of the "head" in the preferred clustering.

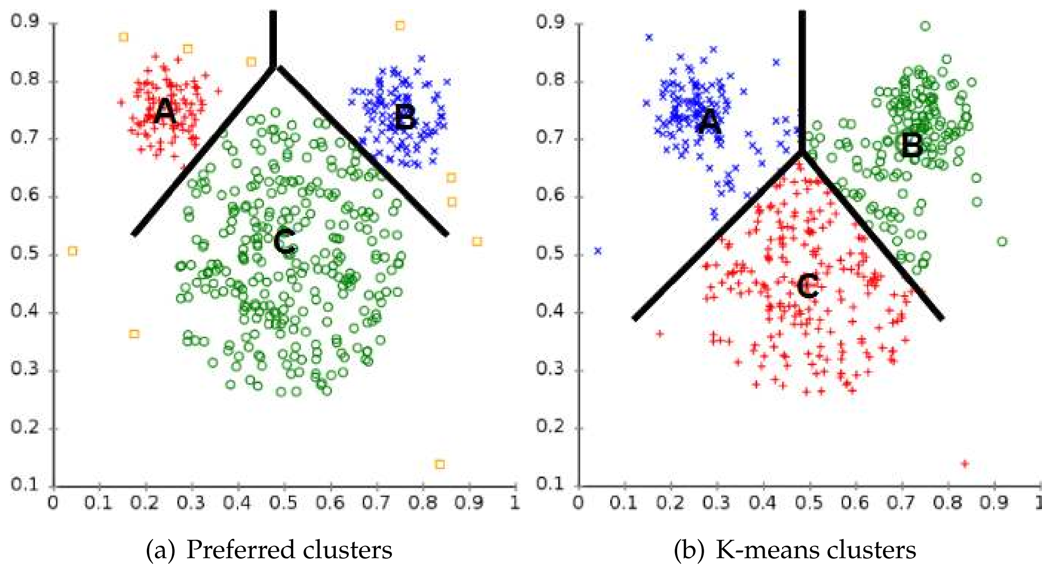
The quality of the clusters produced by k-means is also dependent on the input value for `k`. For a `k` unequal to 3, k-means will split at least one of the preferred clusters of Figure 5.2(a).

K-medoids

The *k-medoid* family [Ber02] of partitional clustering algorithms is similar to k-means. Like k-means, k-medoids is an iterative process of choosing points in space to serve as nuclei for clusters and then clustering about those nuclei. It differs

[1] This figure is based on a public domain image [Chi10].

[2] This example is based on `http://en.wikipedia.org/wiki/K-means_clustering`.



(a) Preferred clusters (b) K-means clusters

Figure 5.2: Spatial clusters

from k-means in how it chooses those nuclei. K-means chooses the mean location (centroid) of a cluster to serve as a nucleus for the next iteration of clustering; k-medoids chooses the most representative member of the cluster (a.k.a. *medoid*) to serve as a nucleus. The k-medoid algorithms share most of the strengths and weaknesses of the k-means algorithms, particularly regarding the choice of the number of partitions to create, and the tendency to create clusters of similar diameters.

Unlike k-means, k-medoids does not require its entities to be represented in continuous space, because there is no need to compute means. Consequently, the entities to be clustered can contain non-continuous properties (e.g., categorical properties), as long as the distance function can accommodate them, so that a medoid can be chosen.

5.2 Related work – applying distance-based clustering to software

This section discusses other researchers' applications of distance-based clustering techniques to refactoring software, especially as applied to modularizing subsystems and to the *Move Method* and *Extract Class* refactorings. In this section, we review their choices of representation, distance functions, and clustering

algorithms for these refactoring tasks, and evaluate the strengths and weaknesses of those choices. In much of this research, entities were represented using property sets. The property sets are summarized in Section 5.2.4, after all of the individual property sets are discussed.

5.2.1 Modularization

Some of the earliest work on applying distance-based clustering to software involved reverse engineering tasks, particularly the modularization of software [AFL99, KE00, Lak97, MM06, Wig97]. Modularizing software encompasses a broad spectrum of topics, including grouping files or classes into packages, grouping subpackages into packages, grouping functions and variables into classes, etc. This section will concentrate on the higher level modularization tasks, the grouping of files or subpackages into packages. The lower level grouping of methods and attributes into classes will be discussed in Sections 5.2.2 and 5.2.3.

It is generally easier to apply clustering techniques to higher level modularization than to class refactoring. In forming high level modules or packages, it is not generally necessary to maintain interfaces or behavior, and the components to be repackaged are generally of the same or very similar types. However, because many of the issues pertinent to high level modularization are also pertinent to refactoring class structure, high level modularization serves as a useful starting point for discussions on the *Move Method* and *Extract Class* refactorings.

Entity Representation

One way of classifying the characteristics of software entities is to divide them into *formal* and *informal* features, where formal features consist of information that directly affects the behavior of the software (e.g., type information, calling relationships, etc.) and informal information does not (e.g., identifier names and comments [AFL99]). Some researchers use the terms "conceptual" or "semantic" when discussing certain informal features, perhaps emphasizing a programmer's intent to produce meaningful identifier names. (Lack of meaningful identifiers and comments severely limits the effectiveness of many "semantic" techniques.)

For example, consider `AnonymousPersistence`'s `saveToDB` method, shown in Figure 5.3. Formally, it references several of `AnonymousPersistence`'s local methods directly (e.g., `loadDriver` and `saveDatabaseValues`), and it also references methods in other classes (e.g., the `DriverManager` class's

```
public void saveToDB(Object obj) {
  String url = "org.apache.derby.jdbc.ClientDriver";
  loadDriver(url);
  Connection connection = null;
  PreparedStatement statement = null;
  try {
    connection = DriverManager.getConnection(url);
    // TODO populate tables using values obtained from
       reflection
    String sqlString =
      "INSERT CLASS_TABLE VALUES (?, ?)";
    statement = connection.prepareStatement(sqlString);
    saveDatabaseValues(obj, statement);
  } catch (SQLException sqle) {
    handleSQLException(sqle);
  } finally {
    releaseResources(connection, statement, null);
  }
}
```

Figure 5.3: saveToDB method

`getConnection` method). There is also informal information encoded in variable names (e.g., `statement`), string constants (e.g., `INSERT`), and comments (e.g., `tables`).

While much of the modularization research concentrates on clustering using formal features [MM06, Wig97], some research suggests that informal features can give better results [AFL99]. Regardless of whether formal or informal features were used, most of the research used either property sets or feature vectors to store the features.

Similarity and distance functions

Anquetil and Lethbridge [AFL99] studied a variety of similarity and distance functions in the context of re-modularization. They recommend the Jaccard similarity function based on the agglomeration results it produced and also due to

its simplicity.

Group clustering

Lakhotia [Lak97] examined many programs that organized software components into subsystems. Of those that used agglomerative clustering techniques, the majority used single link clustering.

Anquetil and Lethbridge [AFL99] recommended complete link clustering. They claim that it emphasizes having modules with high cohesion, whereas single link clustering emphasizes the creation of modules with low coupling. Theirs is a theoretical argument based on the idea that complete link clustering forms compact clusters when the entities can be represented spatially, while single link clustering produces "stringy" clusters for spatially represented entities. They equate compact clusters with highly cohesive clusters. This argument has flaws. First, the software under consideration may not have a good spatial representation for the desired clustering task. Second, even for those cases where the software entities can be represented spatially, dense clusters do not necessarily indicate cohesive software from the standpoint of object-oriented cohesion. Such a correspondence would be highly dependent on a correlation between the similarity function and the object-oriented cohesion metric.

5.2.2 Moving attributes and methods

When a class member is used more heavily by a client class than by the defining class, it may make sense for the definition to be moved to the client class. A distance function can be used to determine which members should be moved to which classes. All of the work discussed in this section involves the idea of measuring the distance or similarity of object-oriented classes and their members. The clustering step varies from being manually performed, as in Crocodile, to being automated, as in CASYR.

Crocodile

Frank Simon and others at the Technical University of Cottbus [SSL01] created a visualization of the distances between class members to help programmers determine whether members need to be moved from one class to another. Their Crocodile tool created property sets for attributes and methods as follows:

- Attributes – the identifiers of all methods that access an attribute, and the attribute itself.
- Methods – the identifiers of all invoked methods, all accessed attributes, and the method itself.

They used a Jaccard metric to calculate the distances between the class members' property sets.

Based on the distances between the various classes' members, the authors produce a visualization of those members using a spring-embedder algorithm [SSL00]. By coloring the members based on the class in which they originated, it is possible for a user to see when a member is more closely associated with a class other than the class in which it was defined. Effectively, it is manual clustering utilizing a visualization. The authors point out that that this visualization can be useful for the *Extract Class* refactoring also.

The researchers did not actually perform any automated clustering of class members using their proposed distance function; they concentrated on visualizing clusters. However, it is easy to use their distance function within a distance-based clustering framework, and JDeodorant has used it to help determine how to extract classes (see Section 5.2.3).

JDeodorant

While they do not use automated clustering techniques for deciding whether to move members from one class to another, researchers at the University of Macedonia make use of the Jaccard distance measure in their JDeodorant plug-in [TC09]. JDeodorant creates property sets for attributes and methods based on the classes, methods, and attributes they are associated with. Based on Jaccard distances between class members and classes, they determine whether feature envy exists. When JDeodorant detects a class member that is closer to another class than the class in which it is defined, JDeodorant suggests Move Method refactorings that can correct the problem.

With some exceptions, they define the property sets for the various entities as follows:

- Attributes – the identifiers of all methods from any class that directly access the attribute, and methods from other classes that access the attribute through accessor functions (a.k.a. getters and setters).

- Methods – the identifiers of all attributes accessed by the method either directly or through accessors, directly accessed methods in the same class, and methods from other classes that are accessed via references.
- Classes – the identifiers of all attributes and methods defined in the class.

The exceptions incorporate knowledge about the software domain via special rules. The rules involve special handling of accessors, static members, delegates, recursive methods, and access to library classes, analogous to the special methods discussed in Section 2.1.3.

CASYR

Serban and Czibula [SC08] were among the first researchers to apply agglomerative clustering techniques to the problem of restructuring classes. They have experimented with several different ways of clustering the entities, including several agglomerative algorithms. These algorithms vary according to how they determine the number of clusters to produce and their criteria for merging clusters.

Their *Clustering Approach for Refactorings Determination (CARD)* system enables experimentation with ways of recombining the attributes and methods of a system into classes. CARD's *Clustering Algorithm for Software Systems Restructuring (CASYR)* algorithm is intended to be useful for *Move Method*, *Move Field*, and *Extract Class* refactorings [SC08]. CASYR creates property sets for all classes, methods and attributes in a system. The property sets for the various entities are defined as follows:

- Attributes – the identifiers of the attribute itself, the application class where the attribute is defined, and all methods that access the attribute.
- Methods – the identifiers of the method itself, the application class where the method is defined, and all attributes accessed by the method.
- Classes – the identifiers of the application class itself, and all attributes and the methods defined in the class.

Because the property sets for the different entities are similar, they can calculate the Jaccard distance between entities of different types and use these distances to determine which entities to put together. For a given step, a class might be combined with another class, a method, or an attribute.

It is clear that their approach to creating property sets has some problems. Their algorithm produces poor results under certain circumstances, because their

property sets for methods do not include either called or calling methods. As an example, for a class that contains no attributes, each method has a property set consisting of itself and the class. Because there is one shared property (the class) and two non-shared properties (the two methods), each pair of methods is equidistant, with a Jaccard distance of 0.67. Agglomerative clustering can not form useful clusters under these conditions, because there are no methods that are closer together than any others.

They say that average link clustering gave better results than single link or complete link clustering, but do not go into any detail about how the results differed. In another part of the paper, they make a theoretical claim that complete link clustering is “generally more useful” than single link clustering, but they do not offer any further backing for this claim either.

kRED

Czibula and Serban [CS06] experimented with partitional clustering using their CARD system. Their *k-means for REfactorings Determination (kRED)* algorithm is a variant of the k-means clustering algorithm that can be used to recommend restructurings for the classes in a software system.

The kRED algorithm creates a vector for each entity (class, attribute, or method) in the software system. The vector for an entity has one entry for each class in the system, containing the distance of the entity to the class. Thus, their spatial “coordinate system” is based on distances from the original classes. To compute the distances, they use the same Jaccard distance measure and property sets as for their CASYR system.

One of the challenges of using k-means effectively is choosing the desired number of partitions/clusters to be created. Czibula and Serban set this number to equal the number of classes in the system, which is a reasonable choice if the designers of the original system have a class structure that is mostly well-designed, but less suitable for situations where classes need to be split or merged. They put the initial centroids at the locations of the original classes, so kRED’s clustering is biased towards maintaining the existing classes in the system. Because of this, *Move Method* or *Move Field* refactorings are the most likely recommendations. However, in a variation from the typical k-means, kRED will reduce the number of clusters whenever an empty cluster is produced, so kRED may also recommend *Inline Class* refactorings [FBB+99], where all of the members of one class will be

inserted into another class.

The kRED approach has some drawbacks. First, the effectiveness of the algorithm depends on the initial nuclei chosen for the clusters, so an optimal solution is not guaranteed. Second, the user needs to specify the number of desired clusters in advance. This is difficult when one of the desired results is a determination of how many classes there should be. Using the existing number of classes is a reasonable heuristic, and their modification to k-means permits the number of classes in the system to decrease. However, the kRED algorithm provides no way of increasing the number of classes in the system, as might be desirable when god classes exist.

The major part of Czibula and Serban's evaluation was on JHotDraw 5.1. Because JHotDraw is considered a well-designed system, they clustered the entities in JHotDraw to see how close they came to reproducing the original class system. They state that only six methods were placed in different classes than they were originally. They then analyzed those six methods and considered the refactoring recommendations to be reasonable.

While these results seem impressive, their kRED technique is biased towards maintaining the status quo. Their coordinate system is based on distances to the original classes; the choice of the number of clusters to create is based on the number of original classes, and the initial centroids are the positions of the original classes. Furthermore, the property sets used to calculate the Jaccard distances for a class member include a property for the class in which the class member is defined, so there is a built-in bias for a class member to cluster with its original class and the class members of that class.

5.2.3 Extracting classes

When classes contain too much functionality, the functionality can be redistributed into additional classes using the *Extract Class* refactoring [FBB+99] (see Section 3.1.3. Several researchers have used agglomerative clustering to determine how class members should be apportioned to the revised classes. Most of these researchers [SC08, FTCS09, BDLO11] use property sets when determining the clusters of members that belong together. The property sets of a class member generally consist of the identifiers of other class members that are related to it via a calling or accessing relationship.

JDeodorant

The JDeodorant researchers [FTCS09, Fok10, FTSC11] have published papers about extracting classes based on the results of single link agglomerative clustering using a Jaccard similarity function. Depending on the paper, the Jaccard similarity function operates on either of two different property sets, both of which differ from those used in CASYR. In all cases, the entity sets are built solely from the class to be split. In 2009 [FTCS09, Fok10], they used the same property sets for class members as were used by Crocodile (Section 5.2.2). In 2011 [FTSC11], the property sets consist of the "local neighborhood" of the class member:

- Attributes – the identifiers of all of the members of the class that use or are used by the attribute.
- Methods – the identifiers of all of the members of the class that use or are used by the method.

They do not explain why they changed their property sets, and in some cases, the newer property set gives worse results (see Section 5.3.1).

JDeodorant does not determine when to stop clustering or decide which are the best clusters. Rather, its shows the clusters as they exist at 0.1 increments of the distance function and lets the users decide which they like best.

Fokaefs, et al. [FTCS09] used the JDeodorant Eclipse plug-in to recommend classes to extract from a student project and a research project. Then, the proposals were discussed with the programs' designers. For both projects, the designers thought that it was worthwhile to apply 43% and 64% of the suggested refactoring changes to increase maintainability. Unfortunately, the programs they examined are not publicly available, so we could not do a comparative study.

University of Salerno

Researchers based predominately at the University of Salerno use both structural and semantic information to cluster members and extract classes. They describe two separate techniques, a "two-step" technique [BDLMO10a, BDLMO10b] and a max flow/min cut technique [BDLO11], that use a common similarity function. In contrast to the previously mentioned research in this chapter, their similarity function is not a Jaccard similarity function. This section begins with a description of the common similarity function, and then discusses how the two-step technique

makes use of it. Their max flow/min cut technique is discussed in Section 6.2.2 in the chapter on graph-based clustering techniques.

Salerno similarity function

Both University of Salerno techniques [BDLMO10a, BDLO11] use a similarity function that combines structural and semantic information to calculate pairwise similarities between all of a class's methods. Their similarity function has three weighted terms. Two of these pertain to structural characteristics of the code, while the third uses semantic information.

The first structural component of the distance function calculates the *Structural Similarity between Methods (SSM)*. SSM measures the similarity of methods based on the common attributes they reference. It is equivalent to a Jaccard similarity measure for methods, where the property set for a method consists of the attributes it references. The paper does not mention whether or not the attributes need to be accessed directly.

The second structural component of the distance function calculates the *Call-based Dependence between Methods (CDM)* [BDLO11], a.k.a *Call-based Interaction Between Methods (CIM)* [BDLMO10a, BDLMO10b]. The CDM considers the exclusivity of access between two methods. Two methods are most similar if one of them only accesses the other. The CDM of two methods, m_i and m_j, is the maximum of the two directional CDM values for the method pair, where the directional CDM value for a method m_i to a method m_j is the number of method calls from m_i to m_j divided by the total number of method calls to m_j.

The third component of the similarity function is the *Conceptual Similarity between Methods (CSM)*, previously discussed in Section 2.1.3. CSM measures the amount of similarity in word usage between two methods.

They empirically determined the best weights to use for the three components of the similarity function based on refactoring results from their two-step technique [BDLMO10b] (described below). In their experiments, they randomly combined pairs of classes from well-designed open-source systems to create artificially noncohesive classes. They then split these noncohesive, merged classes multiple times using their two-step technique, using different weightings on the three components of the similarity function each time. After refactoring, they compared the refactored classes to the original (pre-merger) classes to determine which weightings caused the refactored classes to most resemble the original

classes. The best weightings for the three components of the similarity function varied somewhat, but had broad consistency. For all three projects, they got the best refactoring results when the semantic component (CSM) had the highest weighting (0.6 - 0.7), followed by the structural components SSM (0.2 - 0.3) and CDM (0.1). It is not surprising that the semantic component warranted the highest weighting, because CSM considers all of the semantic information available in a method, whereas both structural elements consider only a portion of the available structural information.

Salerno "two-step" technique

Bavota, et al.'s "two-step" technique [BDLMO10b, BDLMO10a] consists of an initial step that creates multiple clusters, followed by a second step that reconnects small clusters to larger clusters. Using the similarity measures described in the preceding section, they create a fully connected graph, whose nodes are the class's methods and whose edges are weighted with the similarity scores between the connected nodes. Then, all edges with a weight below a threshold of 0.1 are removed, which disconnects the graph. This is equivalent to agglomerative single link clustering, where the "cut" to determine clusters is set at a particular distance. Their second step combines clusters containing fewer than three members with the larger ones using the same similarity function discussed above, using the average link method. Because the two-step technique combines aspects of agglomerative clustering and graph-based clustering (see Chapter 6), it will be discussed in more detail in Section 7.1.

They evaluated their approach by randomly combining pairs of classes from well-designed systems and then seeing whether their technique would extract the original classes when applied to the hybrid classes. They got good results; however, that is not surprising given that the hybrid classes were based on pairs of classes randomly chosen from the system. In general, each of the pre-merger classes would likely have little connectivity with the other, so the resultant hybrid classes were likely to be highly noncohesive, facilitating class extraction.

5.2.4 Property set summary

Table 5.1 lists the items that go into each of the property sets discussed in previous sections, plus one new one, `Nhood`, which is a composite of most of the other property sets in the table. The property sets are:

- `Sim01` – described in Section 5.2.2 and used by JDeodorant in 2009 [FTCS09, Fok10].
- `Ser08` – described in Section 5.2.2 and used in CARD/CASYR [SC08].
- `JD11` – described in Section 5.2.3 and used by JDeodorant in 2011 [FTSC11].
- `SSM` – described in Section 5.2.3 – one component of the University of Salerno's distance function [BDLMO10a, BDLMO10b].
- `Nhood` – a union of the property sets `Sim01`, `JD11`, `SSM`. It does not include all of `Ser08`'s properties, because including the class in the property set serves as noise when clustering is being performed on a single class.

Because some researchers assign different properties to attributes and methods, each property set configuration listed in Table 5.1 has a row for the attribute's properties and a row for the method's properties. A "+" in a column indicates that the item in the column header is included in the property set corresponding to the row. For example, the `Sim01` property set for attributes does not include any called methods. It does include all calling methods, and it also includes the attribute itself, but does not include the class in which it is defined. The table shows considerable variability between what properties are considered important by the different refactoring researchers. The only consensus is that the property set of a method should include called attributes.

Table 5.1: Property sets used for extracting classes

Prop Set	**Member**	**Called**		**Calling**	**Itself**	**Class**
		attr	**meth**	**meth**		
Sim01	attribute			+	+	
	method	+	+		+	
Ser08	attribute			+	+	+
	method	+			+	+
JD11	attribute			+		
	method	+	+	+		
SSM	attribute					
	method	+				
Nhood	attribute			+	+	
	method	+	+	+	+	

All of the entities put into the property sets are connected methods and/or attributes, with the exception of the enclosing `Class` property, which is used only by CARD/CASYR. Presumably, CARD/CASYR make use of the `Class` property to make a class member slightly closer to its original class, when methods and attributes are being moved between classes or when the class hierarchy is being reorganized. For the case of attempting to extract a class based only its members, adding the class to the property set ensures that all members will have something in common, but will not serve to distinguish the class members in any way.

It is interesting to compare these property sets with what is considered important by the object-oriented cohesion metric researchers. As discussed in Section 2.1.3, there are many structural cohesion metrics that center around methods accessing common attributes. The earliest cohesion metrics, like LCOM, generally considered methods directly calling attributes, although later cohesion metrics, like TCC, also considered methods indirectly accessing attributes. Later still, cohesion metrics like DC_D also took into account methods calling other methods, regardless of whether those methods eventually accessed a common attribute.

Many structural cohesion metrics have been viewed from a graph-theoretic basis [CKB00, HM95, ZLLX04, AD10], asserting that the edges between adjacent nodes (class members) are important. Some of these have emphasized directed dependency graphs, e.g., LCOM, TCC, and DC_D. The property sets that are not symmetric correspond to these. For example, the method property sets for `Sim01` include the called methods but not the calling methods. Other cohesion metrics emphasized undirected dependency graphs, e.g., LCOM4, LCC, and DC_I. The symmetric property sets correspond to these. For example, the method property sets for `JD11` and `Nhood` include both the called and calling methods.

5.3 Test suite experiments

We wanted to assess the effectiveness of the prior research, but the work is difficult to repeat. The JDeodorant plug-in is publicly available, but the other software is not, and the descriptions of the algorithms are insufficiently specific to enable duplication in many cases. In addition to the inaccessibility of the refactoring software, there is a lack of access to the software that serves as input. Most of the experiments done using JDeodorant, CARD, and CASYR refactor software that

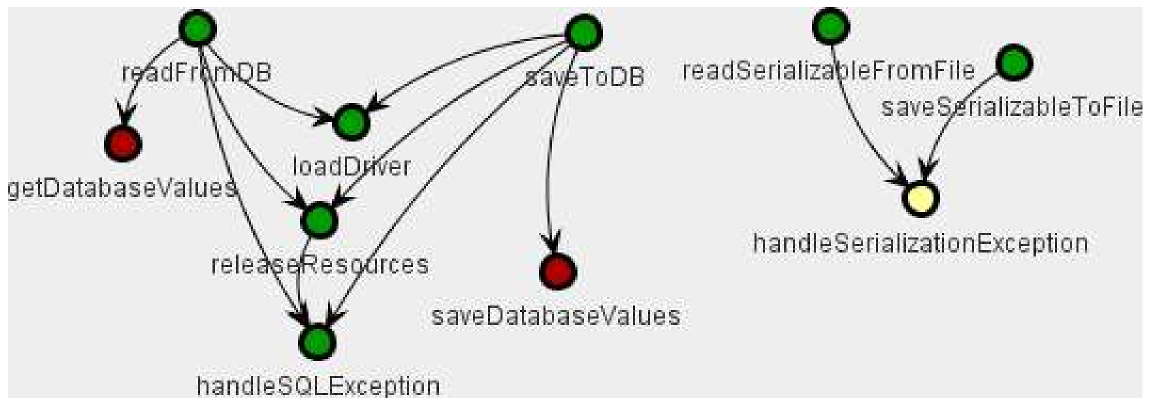


Figure 5.4: AnonymousPersistence intraclass dependency graph

is proprietary or otherwise unavailable. The University of Salerno researchers refactor open-source software, but the basis of many of their experiments involves randomly merging classes using software that is unavailable to us.

In an attempt to help address the problem of lack of reproducibility in the refactoring of object-oriented classes, we created a publicly available test suite for use in testing cohesion metrics and *Extract Class* refactorings (see Appendix A). For the *Extract Class* portion of the suite, a technical report [CAGN10] describes the expected results of the refactoring.

In this section, we discuss experiments we ran to evaluate the effectiveness of various agglomerative clustering approaches for determining how to refactor two simple classes from our test suite, `AnonymousPersistence` and `PersonCarDisjoint`. The code for `AnonymousPersistence` can be found in Appendix A.1 and its dependency graph in Figure 5.4. The code for `PersonCarDisjoint` can be found in Appendix A.2 and its dependency graph in Figure 2.4(a). The clustering algorithms we test are our implementations, based on the descriptions from the research literature, as described in Section 5.2.

The figures show that both classes have two structurally distinct parts. They differ in that `AnonymousPersistence` is composed solely of methods, while `PersonCarDisjoint` has both methods and attributes. `AnonymousPersistence` performs two main tasks. It has three methods that handle saving and restoring serializable objects to files and another seven methods that handle saving and restoring objects to a database. Clustering algorithms should be able to produce two clusters for `AnonymousPersistence`, one consisting of three serialization members and one consisting of seven database members. `PersonCarDisjoint` has a nice property for analysis purposes. It is structurally

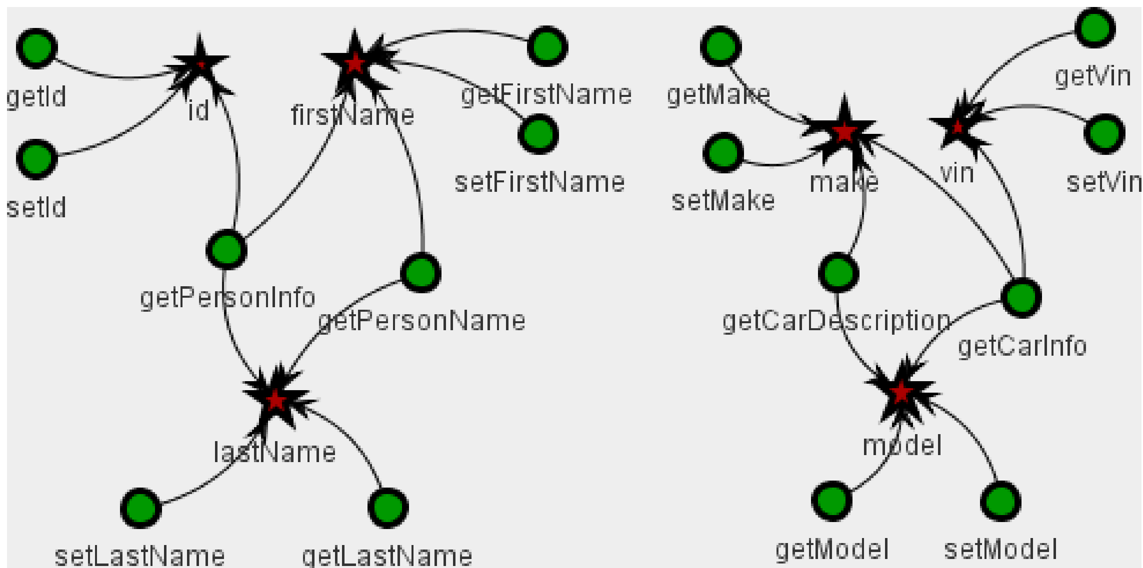


Figure 5.5: PersonCarDisjoint intraclass dependency graph

symmetric; the eight methods and three attributes in the person parts of the class have the same pattern of member access as the eight methods and three attributes in the car parts of the class. Clustering algorithms should be able to produce two clusters for `PersonCarDisjoint`, one consisting of the eleven person members, and one consisting of the eleven car members.

5.3.1 Refactoring based on structure

All of the work on refactoring classes previously discussed in this chapter makes at least some use of local structural information within the property sets of a class's members, that is, a member's properties include a subset of those members to which it is directly linked via *accesses* or *calls* relationships. All of the approaches use a Jaccard similarity function to compare property sets. They differ in what they put in the property sets and in how they combine clusters.

This section examines how combinations of various structure-based property sets and cluster linkage schemes affect the results of clustering. For each of the property sets, we perform three agglomerative clustering runs – one time each for single link, complete link, and average link clustering using a Jaccard distance function, and the clusters produced are compared to the expected results for the test classes. Because the two test classes each have two structurally distinct parts, a structure-based cluster algorithm should identify the class members in those

two parts as the penultimate clusters.

Test class clustering results

We ran the agglomerative clustering algorithms on the `PersonCarDisjoint` and `AnonymousPersistence` test classes. For each of the property sets of Table 5.1, we ran single, complete, and average link agglomerative clustering on the members of the two classes using a Jaccard distance measure to calculate the distance between entities in the clusters.

Table 5.2: Test classes – clustering results

Prop Set	Link	PerCar	AnonPer	Ref
Sim01	single	+	+	[FTCS09, Fok10]
	average	+	+	
	complete	-	-	
Ser08	single	+	-	[SC08]
	average	+	-	
	complete	+	-	
JD11	single	-	-	[FTSC11]
	average	-	-	
	complete	-	-	
SSM	single	-	-	[BDLMO10a, BDLMO10b]
	average	-	-	
	complete	-	-	
Nhood	single	+	+	
	average	+	+	
	complete	-	-	

Table 5.2 summarizes the results. If the clustering produced the preferred clusters at some distance less than 1.0, then a "+" appears in the column corresponding to the class ("PerCar" for `PersonCarDisjoint` and "AnonPer" for `AnonymousPersistence`). For example, the first data row of the table indicates that agglomerative clustering with the `Sim01` property set and single link clustering produced the preferred results for both `PersonCarDisjoint` and `AnonymousPersistence`, whereas the third data row indicates that

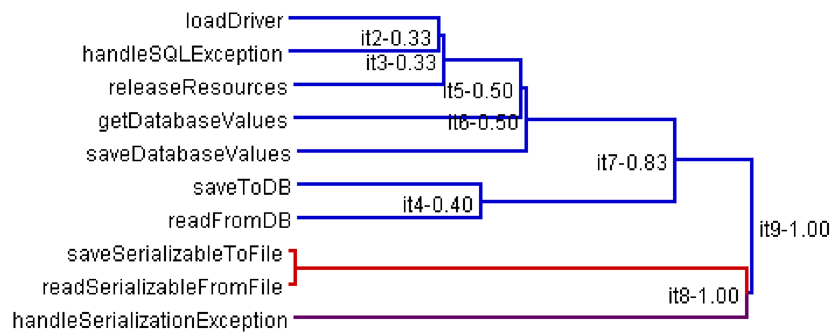


Figure 5.6: Clusters produced – JDeodorant's 2011 distance function (single link)

agglomerative clustering with the `Sim01` property set and complete link clustering did not produce the preferred result for either class.

Results – property sets

`Sim01` and `Nhood` are the property sets from Table 5.1 that produced the best results. They both produced the desired clusters for `PersonCarDisjoint` and `AnonymousPersistence` when either single link or average link clustering was used.

There appear to be two main properties that distinguish these from the other property sets relative to the quality of the results. First, both `Sim01` and `Nhood` include the member in its own property set, which promotes similarity between adjacent class members. Consider the `AnonymousPersistence` test class, whose `handleSerializableException` method is called by two methods. The property sets produced by configuration `JD11` for `handleSerializableEx-ception` will only contain the two methods that call it. Both of those methods have property sets that only contain `handleSerializableException`, but not themselves. Consequently, `handleSerializableException` will not cluster with any other members until a distance of 1.0. Figure 5.6 contains a dendrogram showing this situation, where the cluster containing `handleSerial-izableException` is purple, and the cluster with the two methods that call it, `saveSerializableToFile` and `readSerializableFromFile`, is red.

The second main property that distinguishes the `Sim01` and `Nhood` configurations from the others is that their property sets for methods include all called methods. This is the only difference between `Sim01`, which works for the test classes, and `Ser08`, which does not.

Based on these results, we conclude that agglomerative clustering works best for determining how to split classes when the property sets for class members include the member itself and all methods and attributes that are connected to it via calling or accessing relationships. This is consistent with how many structural cohesion metrics (e.g., TCC, DC_D) use these relationships when calculating connectedness as part of their cohesion calculations.

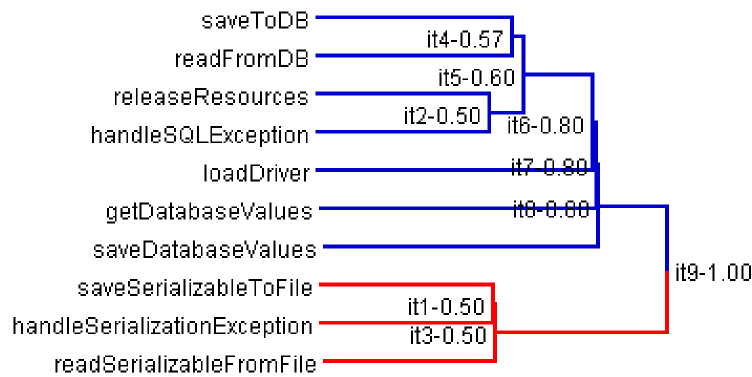
Results – cluster linkage

Table 5.2 shows that single link and average link clustering gave better results than complete link clustering for the property sets we examined. The dendrograms produced for each of these linkage methods for the `AnonymousPersistence` test class help to explain why.

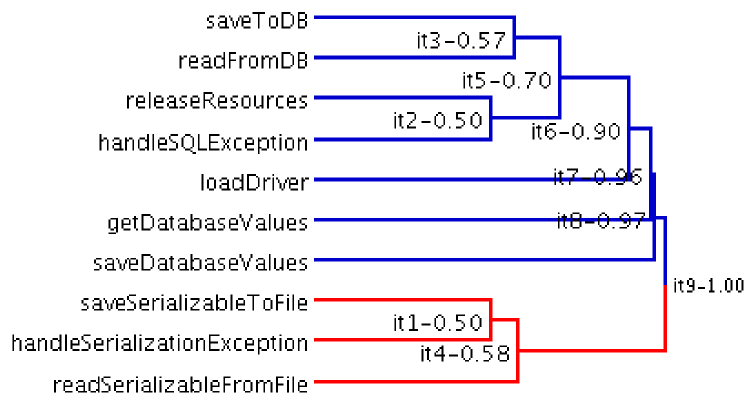
Figure 5.7 shows the dendrograms produced by agglomerative clustering using the `Sim01` property sets for the three different cluster linking schemes. In each dendrogram, the final two clusters are shown in blue and red. Figure 5.7(a) shows the dendrogram produced for `AnonymousPersistence` using single link agglomerative clustering, the configuration used by JDeodorant in 2009 [FTCS09]. This dendrogram shows that the preferred two clusters are produced for distance cutoffs greater than 0.8 and less than 1.0.

Figure 5.7(b) shows the dendrogram when average link clustering is used. It is quite similar to the dendrogram produced via single link clustering, with the exception of the distance values at which many of the clusters are merged. In particular, the preferred two major clusters can not be distinguished until a distance threshold of 0.97, not far from the maximum possible distance of 1.0.

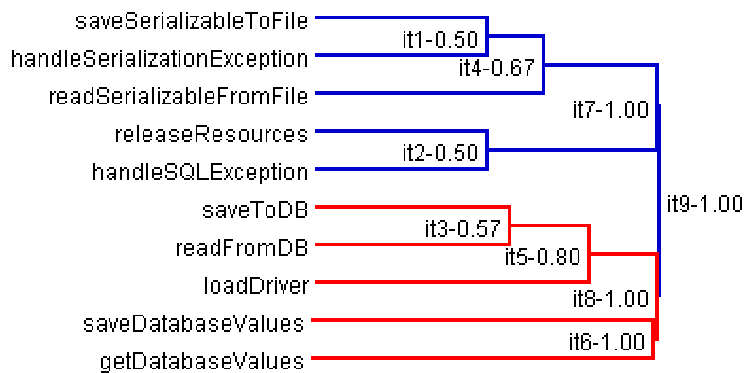
Agglomerative clustering does not produce the preferred clusters when complete link clustering is used with `Sim01`, as shown in Figure 5.7(c). This is due to the interaction of the Jaccard distance function acting on property sets based on local neighborhood information and the functioning of complete link clustering, which merges clusters based on the distances between the most distant members in the two clusters. Both `saveDatabaseValues` and `getDatabaseValues` are only called by a single, distinct method, so each of them has a property set that is composed of the method itself and the single calling method. As a consequence, each of these methods will have a Jaccard distance of 1.0 when compared with any class member that is not connected to its single calling method. This problem with complete link clustering and Jaccard distances based on local connectivity



(a) Single link



(b) Average link



(c) Complete link

Figure 5.7: Agglomerative Clustering – SIM01

information is not particular to this test class. More generally, for any chain of four or more members, any class members that are separated by more than two other members will have no common nearest neighbors, so those members will never

be in the same cluster for any distance less than 1.0.

Although the combination of local neighborhood information and complete link clustering may not produce the preferred clusters for many classes, it does not mean that complete link clustering is useless. A perceptive programmer might still be able to use some of the clusters formed early in the process to form ideas about how best to refactor the class. Nevertheless, single link and average link clustering appear to be better choices.

Restructuring clustering inputs

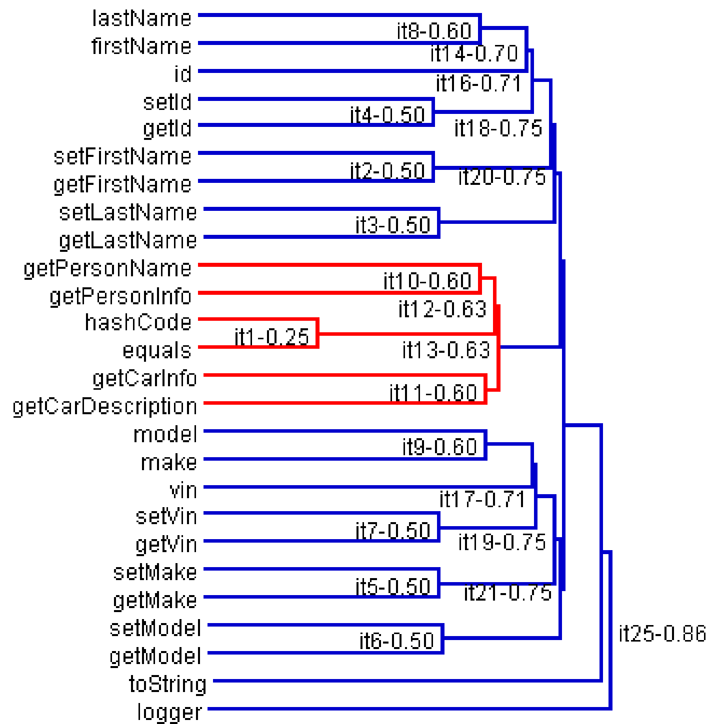
The graph restructuring technique we introduced in the context of cohesion measurement in Section 2.4.1 is also useful in the context of agglomerative clustering. Consider the `PersonCarSpecial` class in Appendix A.5, which has the same basic code as `PersonCarDisjoint`, but with many methods originally defined in the `Object` class being overridden, such as `toString`, `equals`, etc., and the addition of a `logger` attribute. These methods that were inherited from `Object` tend to structurally connect the other methods.

The dendrogram in Figure 5.8(a) shows single link agglomerative clustering with the `Sim01` property set when `Object`'s methods are not filtered out. The `equals` and `hashCode` methods both access all of the attributes, so they are the nearest class members and get clustered first. The `getPersonName` and `getPersonInfo` methods get clustered, because they both access multiple person attributes. Similarly, the `getCarInfo` and `getCarDescription` methods get clustered because they both access multiple car attributes. Because all of these clusters access many of the same attributes, they are combined at a distance of 0.63, resulting in a subcluster that has a mixture of person methods, car methods, and general purpose methods from `Object`. This cluster is near the middle of the dendrogram and is shown in red.

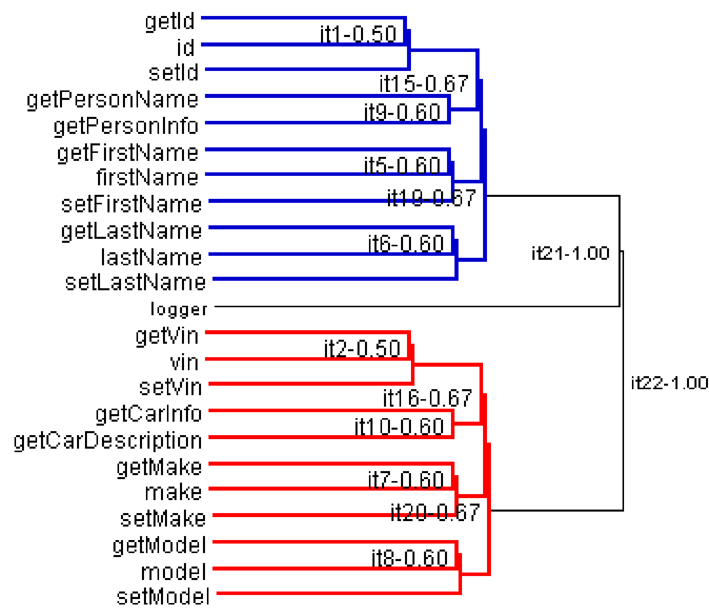
The dendrogram in Figure 5.8(b) shows single link agglomerative clustering with the `Sim01` property set when `Object`'s methods are filtered out. This has the preferred clusters for the person functionality and the car functionality, colored blue and red, respectively.

5.3.2 Refactoring based on semantics

A smaller number of researchers [BDLO11, BDLMO10b, AFL99] have refactored based on informal (conceptual or semantic) features, mostly in the context of



(a) with Object methods



(b) without Object methods

Figure 5.8: Single link agglomerative clustering of PersonCarSpecial (structure)

modularization. It is more difficult to analyze these approaches, due to the variability in the features considered and lack of detail in the papers.

The University of Salerno's approach [BDLMO10a, BDLO11] is the only one we know of that applies informal features to refactoring object-oriented classes. As discussed in Section 5.2.3, they parse identifiers and comments, and use the extracted words to compute the CSM term of their similarity function, which is emphasized more than the two structural terms in the function. Their papers do not provide sufficient detail to enable us to reconstruct their similarity function; however, the semantic distance function we use in the following analysis appears to be sufficiently similar to enable an adequate evaluation.

The idea behind our semantic distance function is based on document clustering techniques [FWE03]. A class's methods and attributes are treated as documents, and the documents' contents are the words present in the identifiers, constants, and non-Javadoc comments. The distance function compares the words present in two given documents. Code adapted from TopicXP [SDGP10] extracts this information. For example, the words extracted from the `saveToDB` method in Figure 5.3 include `savetodb`, `object`, `obj`, `url`, `drivermanag`, `driver`, `manag`, `todo`, `popul`, `tabl`, etc. The words stored are the stemmed forms, so `manag` would be produced from either of `manager` or `manages`. The words stored include the stemmed form of the full identifier, e.g., `drivermanag`, and the stemmed form of its components, e.g., `driver` and `manag`. Certain common words are filtered out, including Java reserved words like `void` and common classes, like `String`.

We use UCLA's S-Space package [JS10] to create a *vector space model* [SWY75] of these documents. In a vector space model, documents are represented as vectors, where each element in the vector corresponds to the number of occurrences of a unique word in the corpus (the collection of documents). We can then compute the distance between two documents (class members) based on the cosine similarity of their vectors.

Our distance function differs from Salerno's CSM in several ways. CSM constructs its vectors using latent semantic indexing (LSI), while we use vector state models. (The underlying semantic cohesion research on which this is based [MP05] states that either can be used.) CSM only computes similarity between methods, while our function also calculates distances involving attributes. It is also unknown whether the words stored in the feature vectors used in the semantic comparisons are the same for the two distance functions. There are likely

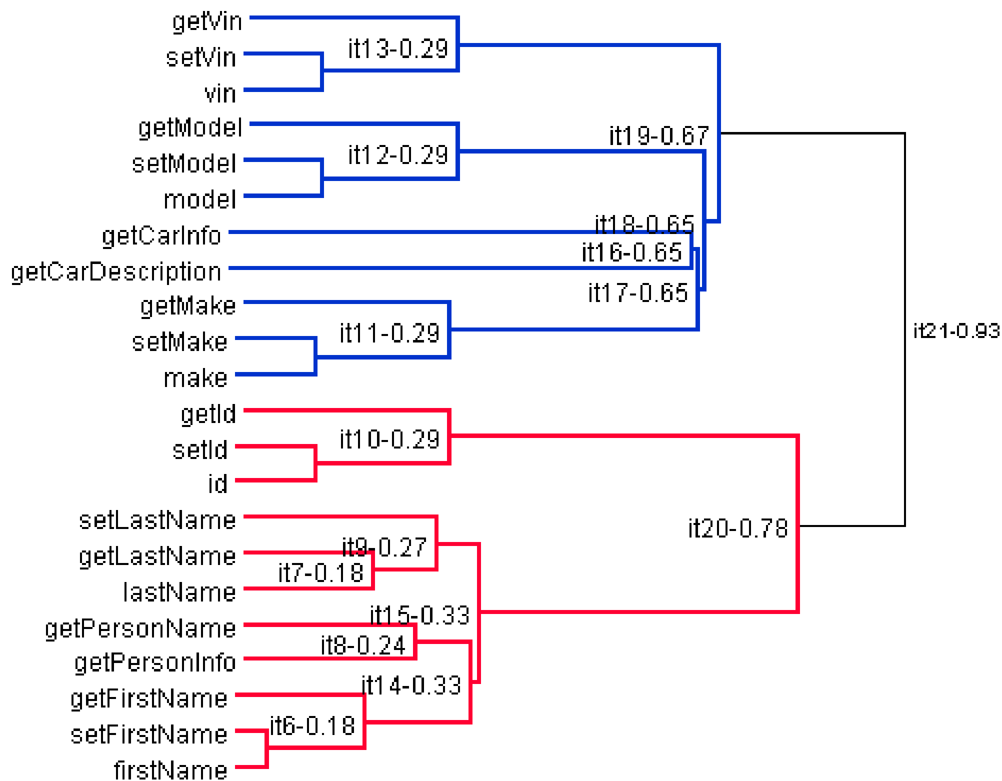


Figure 5.9: Agglomerative clustering of PersonCarDisjoint (semantics)

to be differences, e.g., in how identifiers are parsed, which words are ignored, etc. Despite these differences, we believe the results generated by our distance function on the two simple test classes are likely to be similar to theirs; however, we are less confident that the results would be consistent for larger, more complex classes.

We used our distance function to cluster the members of the `PersonCar-Disjoint` and `AnonymousPersistence` classes. The results of single link clustering for these classes are shown in Figures 5.9 and 5.10.

The clustering for `PersonCarDisjoint` produces the preferred clusters, which are colored blue and red. It is interesting to note that, although the class is structurally symmetric, the dendrogram produced by single link agglomerative clustering is not. This has multiple causes. For example, some accessors, e.g., `getFirstName`, contain more words than others, e.g., `getVin`, and this causes slight differences when computing distances. Similarly, `set-` methods contain more words to be compared (because of their method arguments) than `get-` methods, so the `set-` and `get-` methods are slightly different distances from the attributes they access.

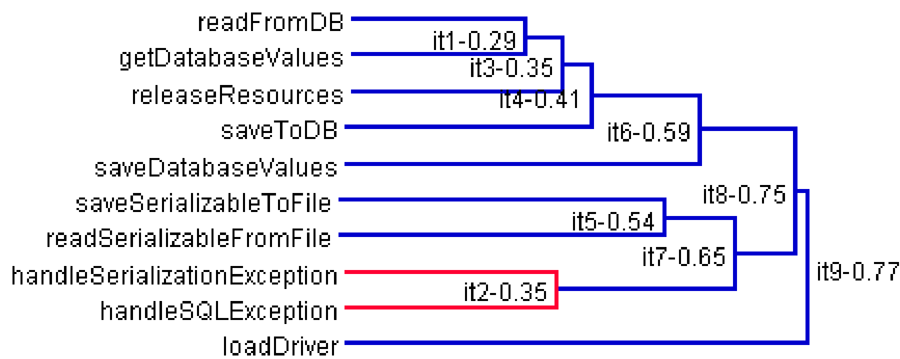


Figure 5.10: Agglomerative clustering of AnonymousPersistence (semantics)

The clustering for `AnonymousPersistence` does not produce the preferred set of clusters. In particular, it is interesting to note that `handleSerialization-Exception` and `handleSQLException` form a cluster (shown in red) early on, because they contain many of the same words. Functionally, they also perform similar tasks, and from an aspect-oriented point of view, some might argue that these methods should be clustered together.

As with agglomerative clustering based on structure, the average link clustering produced the same clusters as the single link clustering, while the complete link clustering failed to produce the preferred clusters for both test classes.

5.4 Open source studies

The experimental results in Section5.3.1 show that several of the agglomerative clustering techniques described in the literature can not determine how to divide simple classes that were designed to be easy to split. This section discusses further investigations of the effectiveness of the two techniques that were successful. We used agglomerative clustering with a Jaccard distance function and the `Sim01` and `Nhood` property sets to cluster the members of thirty open source classes that were selected using the query described in Section 4.1. For each of the thirty classes, we created clusters of their members using single, complete, and average link agglomerative clustering.

Table 5.3 contains the median number of clusters that existed for each of the six combinations of property set and linkage at four different distances (0.5, 0.75, 0.9, and 0.999), where the distance function returns values ranging from 0.0 to 1.0.

Appendix C.2.1 contains detailed data for the number of clusters for each of the 30 classes. At the midpoint of the distance scale, 0.5, the median number of clusters exceeded 41 for every combination of property set and linkage. Even at a distance of 0.999, the median number of clusters exceeded 8 for every combination. Clearly, for these open source classes, it is seldom useful to determine how to split classes by choosing the last two clusters to be merged.

Table 5.3: Open source classes - median number of clusters

Linkage	**Property Set**	**0.50**	**0.75**	**0.90**	**0.999**
Single	Nhood	41.5	14.5	8.5	8.5
	Sim01	43.5	19.5	10.0	8.5
Average	Nhood	44.5	28.0	20.5	8.5
	Sim01	45.0	31.0	22.5	9.0
Complete	Nhood	45.5	30.5	23.5	23.5
	Sim01	47.0	31.5	27.5	26.5

Another possibility for determining whether the clusters are suitable for determining whether to extract classes is to set a cutoff value at which the largest clusters can be used as the basis for new classes. Because the average class has approximately seven methods [LM06], we collected data to determine at which points in agglomerative clustering there were at least two clusters having at least seven members. Table 5.4 summarizes this data. Appendix C.2.2 contains more detailed data about the cluster sizes at the various cutoffs.

Table 5.4: Number of open source classes with 2 clusters of over 6 members

Linkage	**Property Set**	**0.50**	**0.75**	**0.90**	**0.999**
Single	Nhood	0	20	7	7
	Sim01	0	17	7	7
Average	Nhood	0	2	15	7
	Sim01	0	2	13	7
Complete	Nhood	0	2	4	4
	Sim01	0	1	5	5

Table 5.4 shows that, of the six combinations of property set and group linkage,

the highest number of clusters of suitable size for forming revised classes is produced by single link clustering and the `Nhood` property set at a 0.75 cutoff. Here, 20 classes (67%) produced at least two clusters containing at least seven class members.

While two thirds of the classes produced clusters of an appropriate size for creating revised classes, there is a question of whether these clusters would form a sound basis for revised classes. There is an important additional issue as well – how the many, smaller clusters should be distributed to the revised classes.

Figure 5.11: CommandLine intraclass dependency graph

This section examines Jena's (Appendix B.2.4) `CommandLine` class as a case study. `CommandLine` is close to the median number of clusters at the various cutoff values for the combinations of linkage and property sets. Figure 5.11

displays the intraclass dependency graph for `CommandLine`. Structurally, `CommandLine` has a group of methods (in the bottom left of the diagram) that is associated with the `args` attribute, a group of methods (in the upper left) that are associated with the `indirectionMarker`, `allowItemIndirect`, and/or `items` attributes, central functionality associated with the `process` method, and several outlying groups.

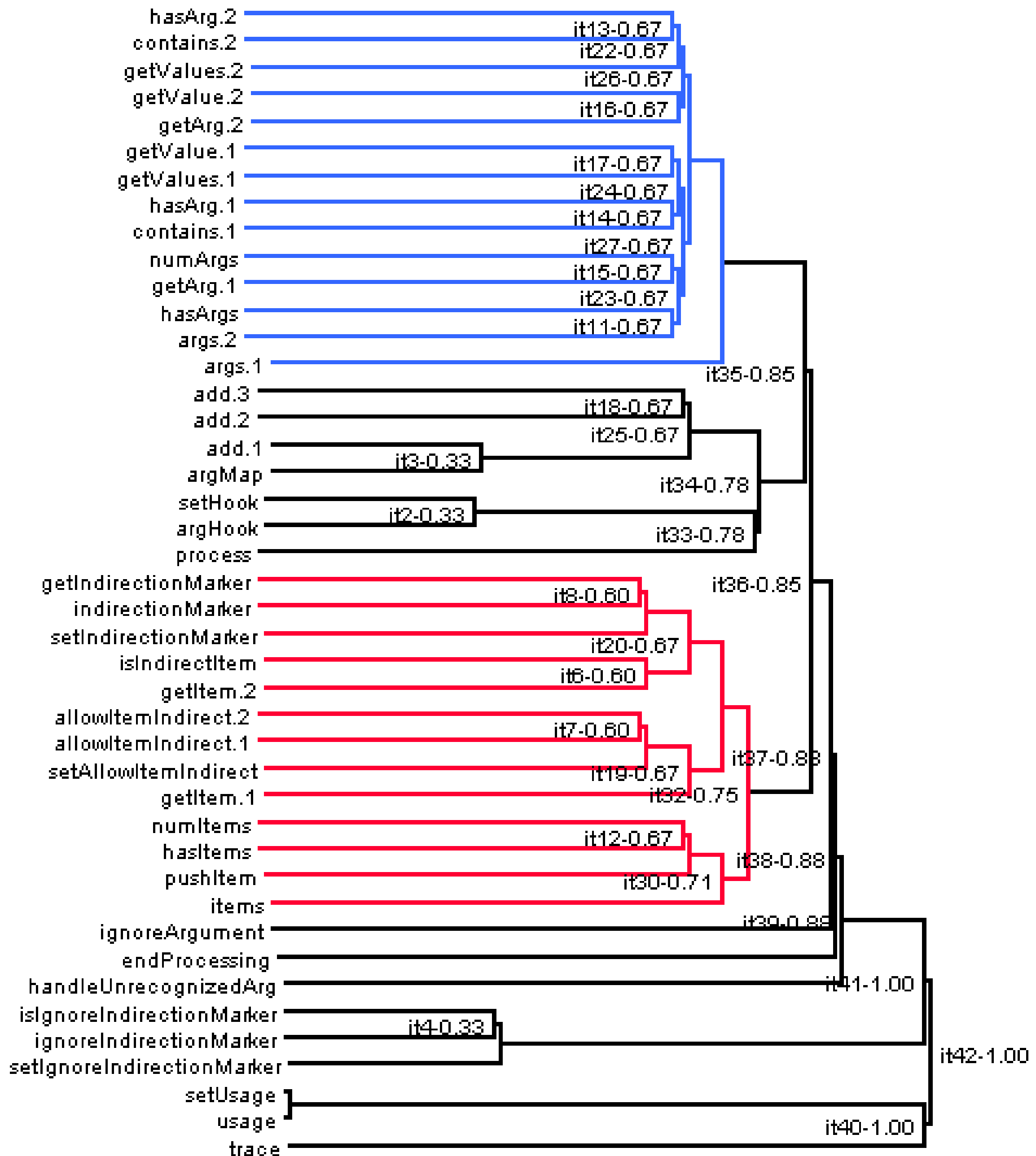
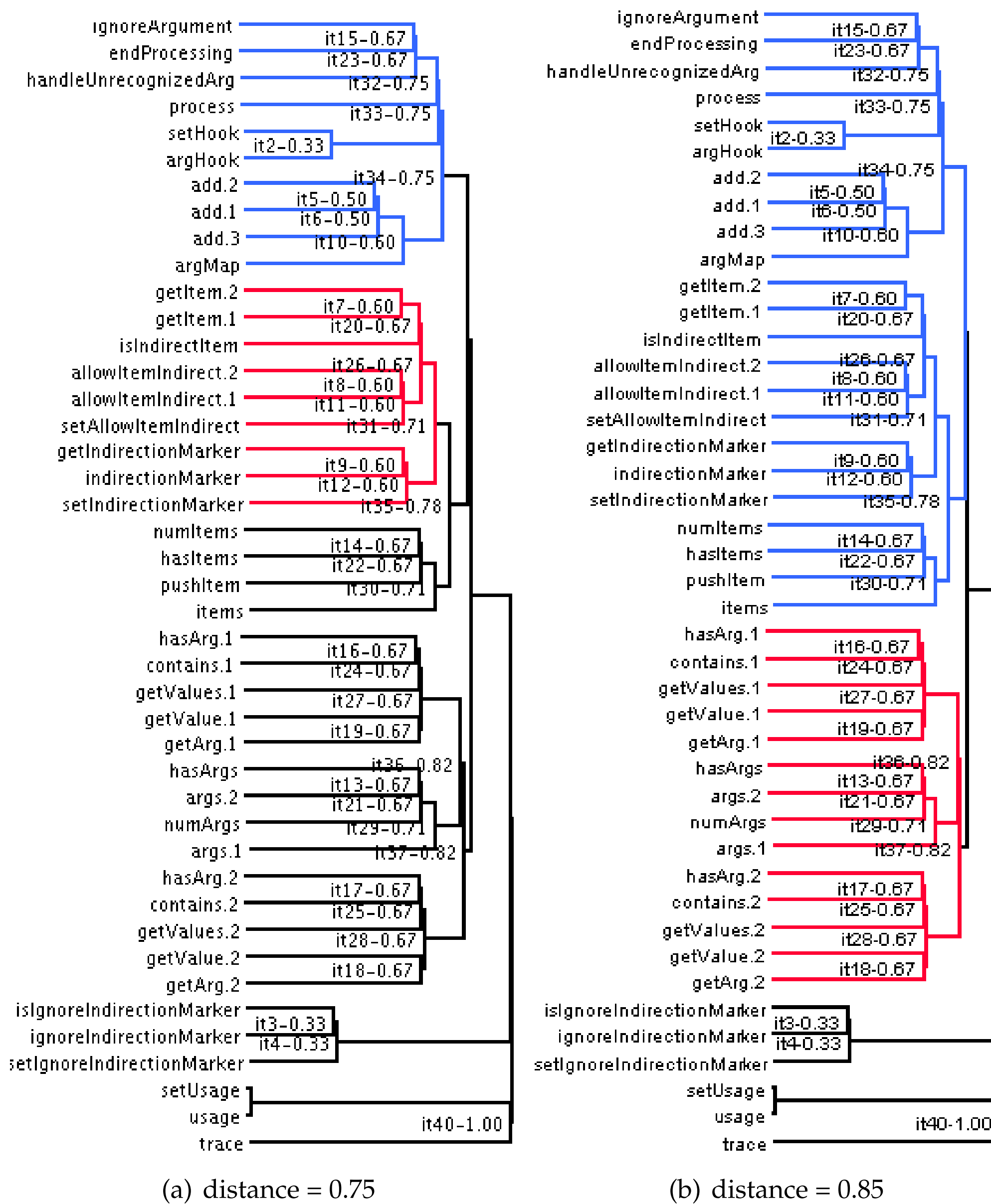


Figure 5.12: Single link clustering of CommandLine (Sim01)

Figure 5.12 shows a dendrogram for `CommandLine` after its class members were clustered using single link agglomerative clustering using a Jaccard distance function and the `Sim01` property set. The two largest clusters at a cutoff of 0.75 are shown in blue and red. The largest blue cluster contains the class members that are associated with the `args` attribute, while the red cluster corresponds to a group of members in the top left of Figure 5.11. At a distance of 0.85, these two clusters merge with each other and a third subcluster, and there are no longer multiple clusters of seven or more members.



(a) distance = 0.75 (b) distance = 0.85

Figure 5.13: Single link clustering of CommandLine (Nhood)

Figure 5.13 shows dendrograms for `CommandLine` after its class members were clustered using single link agglomerative clustering using a Jaccard distance function and the `Nhood` property set. Figure 5.13(a) depicts the two largest clusters at a cutoff of 0.75 in blue and red. The blue cluster corresponds to the group of members associated with the `process` method, while the red cluster corresponds to a group of some of the members in the top left of Figure 5.11. The majority of the class members are outside of the two largest clusters.

Figure 5.13(b) helps illustrate a potential problem when using the two largest clusters from an arbitrary distance cutoff to form the basis of refactored classes. It depicts the two largest clusters at a cutoff of 0.85 in blue and red. The largest blue cluster at a distance of 0.85 contains both clusters that were the largest at a distance of 0.75, whereas the new red cluster at 0.85 contains the class members that were associated with the `args` attribute.

We extracted classes based on the clusters shown in Figures 5.12 and 5.13, and measured the cohesion of those classes. Table 5.5 shows the values for six structural cohesion metrics – LCOM, LCOM*, TCC, DC_D, LCC, and DC_I, together with the C3V semantic cohesion metric for those classes. The "Class" column contains entries for the *original* class (before it was refactored), and the refactored classes produced based on the clusterings provided by single link agglomerative clustering. One refactoring was based on clustering with the `Sim01` property set, corresponding to Figure 5.12. The other two refactorings were based on clustering with the `Nhood` property set – corresponding to Figures 5.13(a) and 5.13(b). For each of the clusterings, the *modified* class was formed from the largest cluster plus outliers, while the *extracted* class was formed from the second largest cluster. The Δ *modified* rows contain the improvement in the given measurement between the original and modified classes, and the Δ *extracted* rows contain the improvement in the given measurement between the original and extracted classes. Due to rounding, the improvement rows may sometimes appear to be off by 0.01.

The refactoring based on the clusters produced by `Sim01` and `Nhood` at a distance of 0.75 both seem to be slight improvements over the original class. The refactoring produced by `Nhood` at 0.75 has a slightly less cohesive modified class, but a more cohesive extracted class than both the original class and the one produced by `Sim01` at 0.75. The refactored classes based on the clusters produced by `Nhood` at a distance of 0.85 have generally higher cohesion than the refactored classes based on the clusters produced by `Sim01` and `Nhood` at a distance of 0.75. This is primarily due to the addition of loosely associated class members to the

Table 5.5: Cohesion metrics - refactored CommandLine

Class	LCOM	LCOM*	TCC	DCD	LCC	DCI	C3V
Original	507	0.93	0.22	0.22	0.64	0.64	0.17
Sim01 (dist = 0.75)							
modified	449	0.91	0.26	0.26	0.64	0.64	0.17
extracted	3	0.63	0.60	0.60	1.00	1.00	0.34
Δ modified	58	0.02	0.04	0.04	0.00	0.00	0.00
Δ extracted	504	0.30	0.38	0.38	0.36	0.36	0.17
Nhood (dist = 0.75)							
modified	483	0.93	0.21	0.21	0.40	0.40	0.17
extracted	0	0.61	0.71	0.72	1.00	1.00	0.37
Δ modified	24	0.00	-0.01	-0.01	-0.24	-0.24	0.00
Δ extracted	507	0.32	0.49	0.50	0.36	0.36	0.20
Nhood (dist = 0.85)							
modified	355	0.90	0.22	0.22	0.64	0.64	0.14
extracted	61	0.00	1.00	1.00	1.00	1.00	0.56
Δ modified	152	0.03	0.00	0.00	0.00	0.00	-0.03
Δ extracted	446	0.93	0.78	0.78	0.36	0.36	0.39

clusters that existed at a distance of 0.75. As previously mentioned, there was only one cluster with at least seven members produced by `Sim01` at a distance of 0.85, so no classes were extracted.

For average link agglomerative clustering using a Jaccard distance function, neither `Nhood` nor `Sim01` produced two clusters of at least seven class members at the 0.9 distance cutoff. However, at a distance of 0.99, both property sets generated the same top level clusters as `Nhood` did for single link clustering at a distance cutoff of 0.85 (Figure 5.13(b)).

For complete link agglomerative clustering using a Jaccard distance function, neither `Nhood` nor `Sim01` produced two clusters of at least seven class members at any distance below the 1.0 maximum. As Table 5.4 shows, this is a common problem with the use of complete link clustering with these property sets.

This case study illustrates some of the difficulties of using agglomerative clustering as a basis for determining how to refactor classes. While the clusters produced can lead to the production of more cohesive refactored classes, it is not obvious which cutoff distance produces the clusters that result in the highest quality classes. However, manual inspection of the dendrograms in conjunction with a visual analysis of the class's structure can be helpful for guiding the refactoring.

5.5 Visualizing agglomerative clustering

Among the weaknesses of using agglomerative clustering for refactoring is the difficulty of understanding why clusters are being combined. Distance functions tend to be opaque, i.e., as the agglomerative clustering proceeds, it is often difficult to see why clusters are being combined. To partially address this shortcoming, we devised a novel visualization of agglomerative clustering in the context of the class's structure [CAGA11].

This section shows the visualization of clustering using an example distance function we created based on structural connectedness. While examining open source classes [CAGA11], we noticed that classes sometimes had chains of method calls. Based on the idea that a class member that was exclusively linked to another member could only be joined to that member, we created a novel distance function that produces a small distance for nodes that have few links except to each other. For two nodes representing class members, the distance equals the shortest

undirected path between the two nodes plus an added fractional "exclusivity" distance based on the number of incident edges on the two nodes. The exclusivity distance equals 1 when the node have no common edges, otherwise

$$dist(a, b) = 1 - (2/(edgeCount(a) + edgeCount(b))) \tag{5.3}$$

where the *edgeCount* function returns the total number of edges incident to a node. In this agglomerative clustering example, we can recalculate distances based solely on the properties of the underlying graph representation. This distance function has the advantage of simplicity. It is calculated based purely on the evolving graph – it does not require any of the cluster linkage mechanisms described in Section 5.1.3.

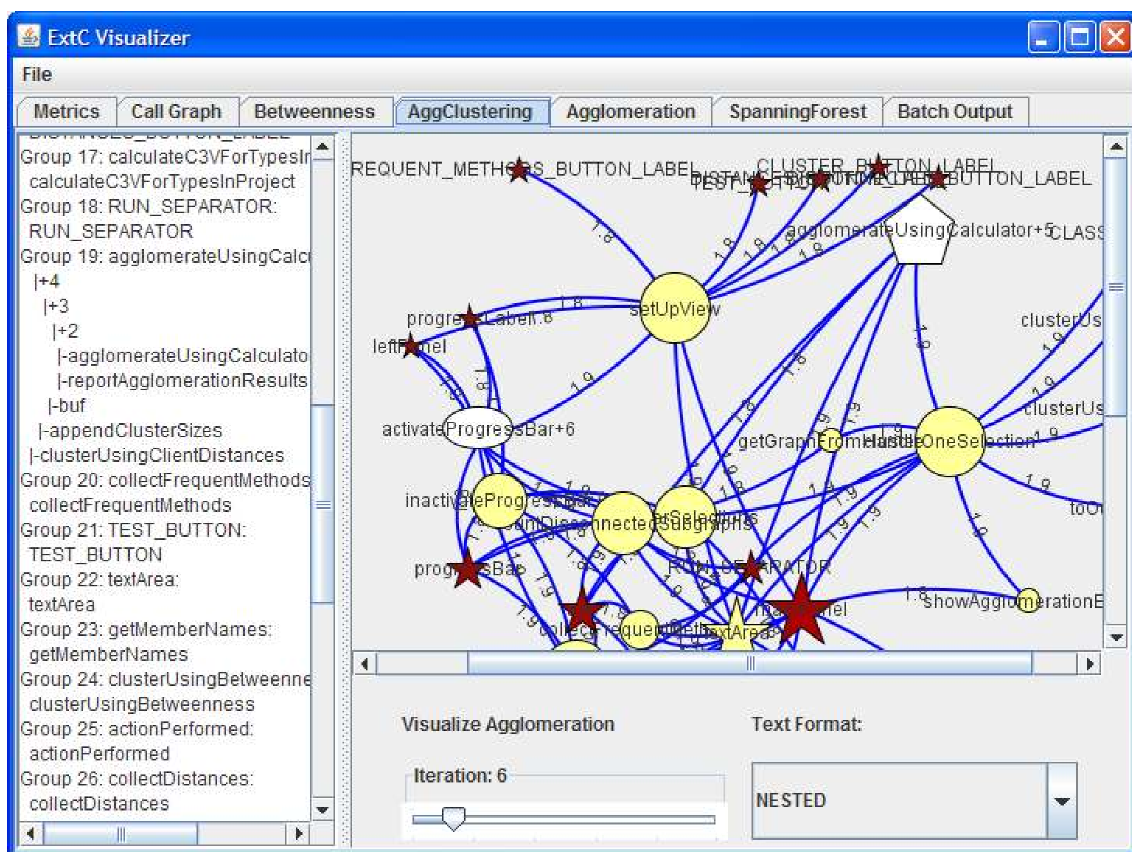


Figure 5.14: Agglomerative clustering view

Figure 5.14 shows the agglomerative clustering view. Due to the importance of intraclass structure, the basis of the visualization is an undirected version of an intraclass dependency graph, where stars represent unclustered attributes, circles represent methods, and white shapes represent clusters consisting of two or more

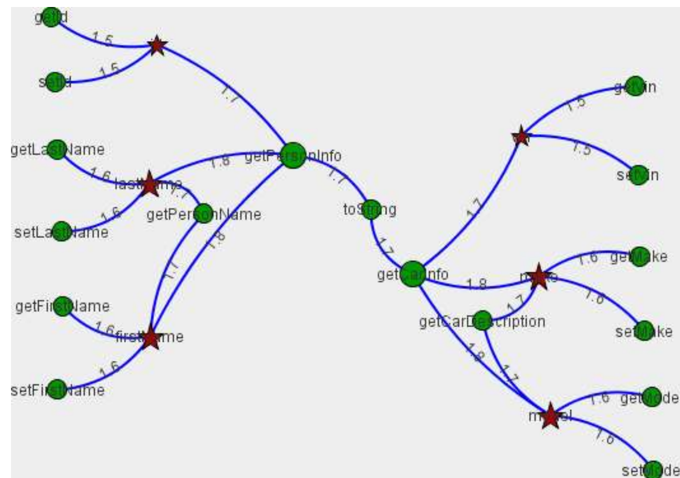
members. Clusters of two members are represented as ovals, while larger clusters are represented by polygons, where the number of sides is equal to the number of members in the cluster. Cluster nodes are labeled with the name of one of their members followed by the iteration in which the cluster was formed.

Edges represent either a method calling a method or a method accessing an attribute. The weights of the edges are the distances between the linked class members. While implementations of agglomerative clustering typically maintain distances between all clusters, our graph only shows the distances between structurally linked nodes, rather than for all node pairs, emphasizing the significance of the class's structure.

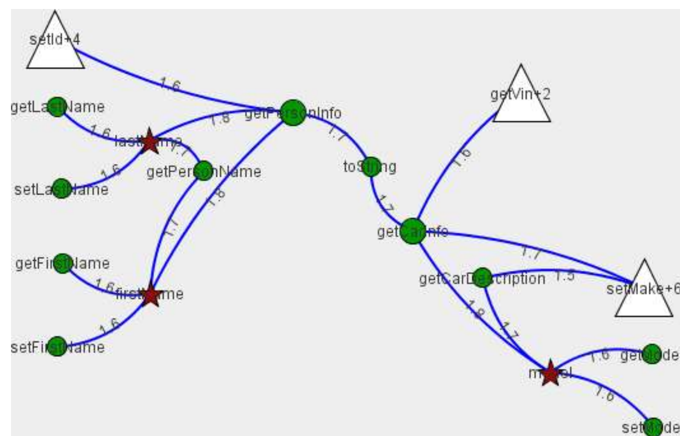
The user modifies the clustering iteration using a slider. Moving the slider to the right causes the clustering iteration to advance, and the distance values on the edges to change. At each iteration, the two nearest nodes are merged. The node that is farthest from the center is removed, while the more central one "absorbs" it and changes shape. After the merge step is completed, the edge weights (distances) are recalculated. While this is happening, the pane on the left displays a textual representation of the hierarchical structure of the clusters as they evolve.

Figure 5.15 shows agglomerative clustering in action. (The figure only shows the pane containing the graph, to conserve space.) Figure 5.15(a) shows the dependency graph before the first cluster is formed. The effects of the first six iterations of clustering are relatively uninteresting as the nodes on the outskirts of the graph are being merged with their only neighbors, and no cluster has more than three members. Figure 5.15(b) shows the graph after the sixth iteration, where there are three clusters of three members. Each of these clusters consists of an attribute together with its two accessors, and the distance function seems to be working as intended. At this point, we expected to see more clustering of attributes with their accessors. Instead, the cluster consisting of `make` and its accessors merges with `getCarDescription`. Figure 5.15(c) shows the graph after seven clustering iterations, when this group of four is formed.

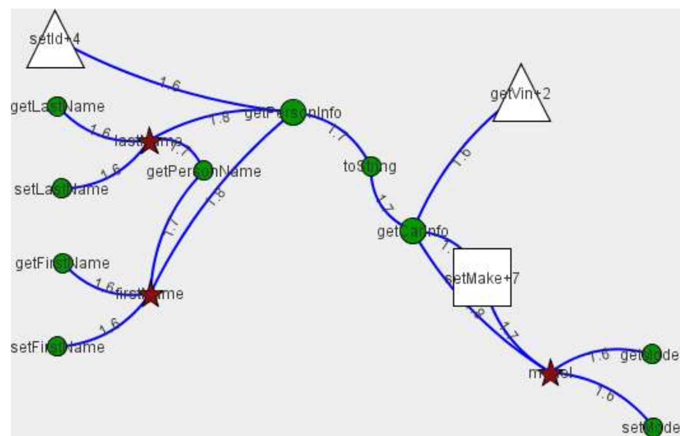
This merger was an unintended consequence of our representation combined with our distance function. As clusters merged, edges disappeared, which affected the distances between nodes. For example, in Figure 5.15(a), `getCarDescription` is connected to `make` via an edge with weight 1.7, because `make` is linked to several other nodes. In Figure 5.15(b), `getCarDescription` is connected to the cluster containing `make` (`setMake+6`) via an edge with weight 1.5, because `setMake+6` only has one other edge.



(a) Iteration 0



(b) Iteration 6



(c) Iteration 7

Figure 5.15: Agglomerative clustering visualization

It was relatively easy to identify the problem in our representation using the visualization, because the visualization highlights the structural relationships between class members and displays a select few of the possible inter-member distances. While the clustering could have been debugged using a combination of dendrograms and traces of the distance matrix produced during clustering, that would likely have been a more time consuming and tedious task.

While this visualization is helpful for debugging some problems in agglomerative clustering, it has limitations. Chief among them is the difficulty of visualizing clustering involving a large number of entities. In addition to the problem of screen clutter, there is a potential problem with tedium – there can be many clustering steps to observe. Consequently, this visualization is perhaps most useful for initial exploration of agglomerative clustering with a small number of entities.

5.6 Evaluation of distance-based techniques

The previous sections have discussed several distance-based clustering techniques, their uses for refactoring, and some of their individual strengths and weaknesses. This section evaluates these techniques at a higher level, and identifies common strengths and weaknesses.

One of the biggest strengths, and one of the biggest weaknesses, of distance-based clustering algorithms is the distance functions themselves. Distance functions are extremely flexible – anything that can be coded to produce a numeric result can be used as a distance function. While this flexibility is capable of producing excellent results, it also has drawbacks, particularly when the functions become complicated. Distance functions tend to be opaque – when clustering produces unexpected results, it can be difficult for analysts to determine why.

The difficulty of designing good distance functions is exacerbated by some characteristics of the object-oriented domain. It is hard to create useful distance functions that cover diverse entities like attributes and methods, and it is hard to embed knowledge about preferred (software) characteristics of the cluster, e.g., that the class produced from the cluster should be of a certain size and cohesion. Without such domain knowledge, distance-based clustering is liable to produce clusters unsuitable for forming classes.

In the approaches discussed earlier in this chapter, the researchers have chosen to create distance functions that treat attributes and methods nearly identically

during clustering. Some of these researchers concentrated on the structural neighborhoods of the entities, while others concentrated on the informal features – the words used in the code and comments. In both cases, the distance functions capture some of the information important to creating a good class (some aspects of structural or semantic cohesion), while ignoring others. Consequently, these functions produce useful results in some cases, but not others.

5.6.1 Evaluation of agglomerative techniques

Earlier sections have shown how agglomerative clustering can identify useful clusters of classes that facilitate the *Extract Class* and *Move Method* refactorings. Although there has been some success, agglomerative clustering has a number of drawbacks in regards to refactoring in general, and extracting classes in particular.

Agglomerative clustering is not a good cognitive fit with the *Extract Class* refactoring, where the programmer's goal is to divide a class into more maintainable ones. For *Extract Class*, a programmer wants to divide a class's members, while maintaining many of the interconnections between members within each of the resulting classes. (Chapter 6 discusses refactoring using graph-based divisive clustering, which seems more conceptually suited to *Extract Class*.) With agglomerative clustering, one breaks up a class into its members and reassembles them, hoping that the distance functions are good enough to recombine cohesive groups of class members. While agglomerative clustering can produce good results in this manner, its method for doing so may be different from how most programmers would produce the same results.

Evaluating clusters

It can also be difficult for a programmer to decide how to use agglomerative clustering results, particularly for large real-world classes. Consider the clusters produced from the members of Weka's `RegOptimizer` class using a semantic distance function, shown in Figure 5.16. Unlike for the simple case of `PersonCarDisjoint` illustrated in Figure 5.9, merely choosing the last two unmerged clusters does not provide a good result. For `RegOptimizer`, the last clusters to be merged are outliers that have little apparent similarity to any of the other class members. Programmers are unlikely to be impressed with a suggestion to extract a new class consisting of the single private attribute `serialVersionUID`.

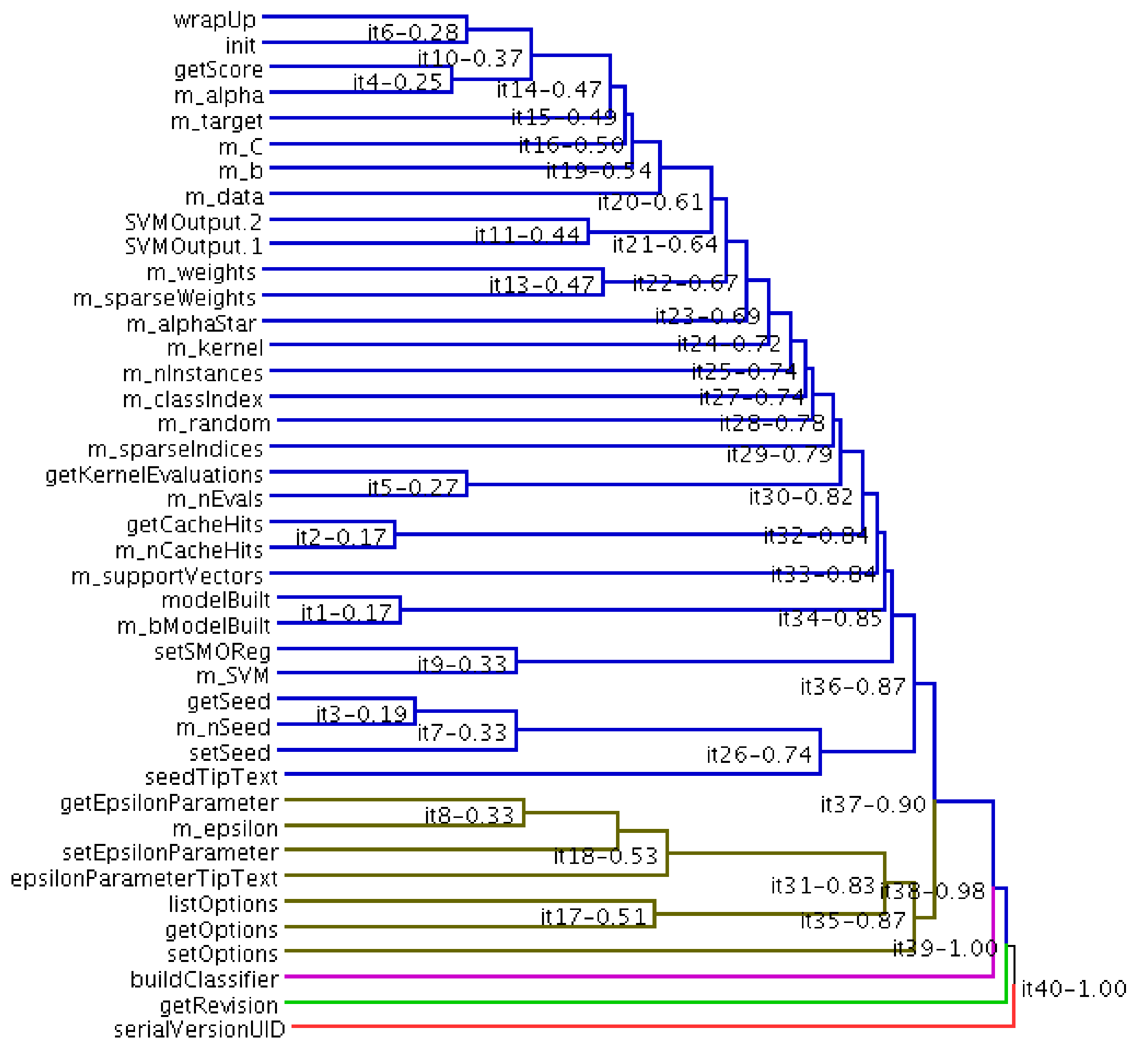


Figure 5.16: RegOptimizer dendrogram

In some cases, it is possible to choose clusters through the choice of a particular cutoff distance. In the dendrogram in Figure 5.16, it looks as though there may be useful clusters at a distance of 0.90 – a large blue cluster consisting of the members `wrapUp` through `seedTipText`, a seven member brown cluster consisting of `getEpsilonParameter` through `setOptions`, and three single member outlier clusters, `buildClassifier`, `getRevision`, and `serialVersionUID`. However, without experimentation, or a sound theoretical explanation, there is no way of knowing whether such a cutoff value has any validity. There is not some universally useful cutoff distance that indicates which clusters might serve as the bases for classes; however, through experimentation, researchers can establish useful cutoffs for particular clustering techniques. For example, for the first step

of their two-step technique, the researchers at Salerno chose to remove all edges from their similarity graph with a weight less than 0.1 (which is equivalent to collecting the clusters at a distance of 0.9) based on experimental results with different thresholds [BDLMO10b].

However, unless the intent is to have fully automated refactoring, it is not necessary to have a criterion for deciding which clusters are most suitable for creating clusters. Programmers can manually inspect results and choose clusters that seem promising as the bases for new classes. Hierarchical clustering has advantages over partitional techniques here, because programmers can examine more closely related subclusters and apply their judgment regarding the potential utility of the subclusters. For example, a programmer might look at the dendrogram of Figure 5.16, and decide that the first eight members, `wrapUp` through `m_data`, might make a good class. However, this manual inspection requires more effort on the part of the programmers than when programmatic criteria can be used to identify the most promising clusters.

Evaluating cluster linkage

Based on the flawed results produced by complete link clustering for the simple test cases presented in Section 5.3.1, and on the clustering results for open source classes presented in Section 5.4 we feel that both single link and average link clustering are preferable to complete link clustering for extracting classes. For deciding where to split classes, it is important to maintain existing connectivity between class members. Because single link clustering extends clusters based on shared properties, it tends to maintain connectivity when the distance function is based on local structure. Complete link clustering, on the other hand, will not form clusters (before a distance of 1.0) between entities that have no shared information. Class members that are connected only through methods chains whose length is greater than two will never be in the same cluster (prior to the clusters formed at a distance of 1.0).

Our recommendation to use single link or average link clustering contrasts with the preliminary conclusion reached by Anquetil and Lethbridge [AFL99] to use complete link clustering for large scale modularization; however, that modularization task may be less concerned with non-local interactions. We suspect that further investigation will show average link to be preferable to single link clustering, because single link clustering is extremely sensitive to the placement of

individuals; however, this has not yet been verified.

5.6.2 Evaluation of k-means and k-medoids

K-means and k-medoids appear to be worse choices for most refactoring tasks than the agglomerative distance-based approaches. Besides sharing the burden of having opaque distance functions, k-means and k-medoids both build clusters relative to some average, or prototypical, point in space, and it is not clear what spatial dimensions are suitable for object-oriented software.

It can also be difficult to choose a proper value for the number of partitions, `k`. For some refactoring tasks, the choice of the number of partitions is relatively clear, for example, setting `k` to 2 for redistributing methods between two classes. It is harder to set `k` for some other refactoring tasks. Setting `k` to 2 for *Extract Class* seems a reasonable strategy, but can produce poor results, for example, for a class that was composed of three distinct spatially-related subsets of attributes and methods, as in Figure 5.2.

Also, being partitional algorithms, k-means and k-medoids do not provide an easy means of using partial results. Agglomerative clustering produces nested clusters, so it is possible for programmers to examine more closely related subclusters when the higher level clusters are unsatisfactory, and use the members of the subclusters as the foundation for new classes. There are no such subclusters produced by partitional algorithms. Of course, an analyst could always calculate the closest members of a partition after the clustering, but it is unclear what advantage this would have over using agglomerative clustering from the start.

5.7 Contribution summary

The idea of using agglomerative clustering to organize software is not new; however, important aspects of clustering have been overlooked, particularly in regard to refactoring object-oriented classes. This chapter has analyzed several approaches to refactoring object-oriented software based on their application to test classes that we have developed and on open source classes. This analysis has identified areas of weaknesses that have been ignored previously, for example, the drawbacks of using local connectivity information with complete link clustering. Consequently, we recommend single link or average link clustering over complete

link clustering when using agglomerative clustering with local connectivity information.

We have also provided a mechanism for restructuring the inputs to clustering algorithms to help eliminate some of the confounding effects of special methods like `toString`, `equals`, etc. By eliminating "noisy" structure that is not related to the main purpose of the class, we can better split noncohesive classes.

Among the weaknesses of using agglomerative clustering for refactoring is the difficulty of understanding why clusters are being combined. To address this shortcoming, we devised a visualization of agglomerative clustering in the structural context of the class. This visualization showed the clusters growing with each iteration of clustering, making it easier to discern whether clusters were forming consistent with the underlying structure of the class.

Most of the approaches discussed in this chapter use local structural information to try to determine whether class members belonged together. In Chapter 6, we present graph-based clustering techniques, an alternative means of clustering using structure, that we feel is more appropriate to the problem of splitting classes. Chapter 7 discusses how a combination of graph-based and agglomerative clustering techniques can produce a result that is better than either technique individually.

Chapter 6

Refactoring Using Graph-Based Clustering Techniques

Graphs are a good choice of representation for many domains where the relationships between entities are important. For these domains, nodes can represent entities, and edges can represent the relationships between the entities, e.g., communication.

Graph-based clustering algorithms separate the entities in a graph into clusters, typically by removing a small number of edges (the *cut set*), which disconnects the graph. Each connected component corresponds to a cluster, and the entities represented by the nodes in the component are the elements of the cluster.

This chapter begins with a discussion of some graph-based clustering algorithms that have been applied to refactoring software, followed by discussions of other people's research in refactoring object-oriented software based on graph-based clustering. Section 6.3 discusses our approach for applying a graph-based clustering technique known as betweenness clustering to discovering subsets of class members suitable as the basis of an extracted class, and Section 6.3.1 describes our visualization of betweenness clustering operating on object-oriented classes. Section 6.3.2 discusses how we use betweenness clustering results as the basis for splitting large open source classes and how that results in extracted classes with improved cohesion. In section 6.4, we evaluate the relative strengths and weaknesses of graph-based clustering techniques compared to other clustering approaches for refactoring classes. The chapter concludes with a summary of our contributions.

6.1 Background – graph-based clustering

Graph-based clustering algorithms work on an underlying graph representation of the data, splitting the graphs by removing edges. For graph-based clustering algorithms, the quality of the result depends on what is represented by the graph's nodes and edges, as well as the algorithm used to partition the graph.

Researchers from a variety of fields are interested in graph-based clustering algorithms. Biological and social network analysis researchers often use graph-based clustering algorithms to find interconnected communities of individuals, an activity known as *community detection* [For10, MW03, RB07, GN02]. The community detection algorithms they use look for heavily interacting groups of individuals who do not interact with many individuals outside the group. Transferring these ideas to the software domain, the groups of highly interacting individuals correspond to cohesive systems which are loosely coupled to other cohesive systems.

This section contains a brief overview of some graph-based clustering algorithms that have been used for refactoring. There are more extensive sources of information available on graph-based clustering and community detection algorithms [New10, For10, Sch07].

6.1.1 Splitting a minimum spanning tree

Undirected graphs can represent groups of entities, where nodes represent entities and edges represent the distances between them. For connected graphs, there are algorithms [Pri57, Har75] to construct *minimum spanning trees*, trees that contain all of the original nodes with the lowest possible sum of edge weights. Figure 6.1 shows a minimal spanning tree for a collection of five mammals.

If an algorithm then iteratively removes the edges with the largest distances, it separates the spanning tree, and the disjoint subtrees form clusters of entities that are relatively close together. In the example of Figure 6.1, the first iteration removes the edge between gorilla and wolf, because that edge has the largest distance (12). This divides the initial cluster into two clusters – the primates (lemur, human, and gorilla) and the canines (wolf, dog). The next iteration separates the lemur from the human and gorilla, and so forth.

This iterative process of splitting the tree produces the same results as single link agglomerative clustering, only in a divisive fashion, rather than an

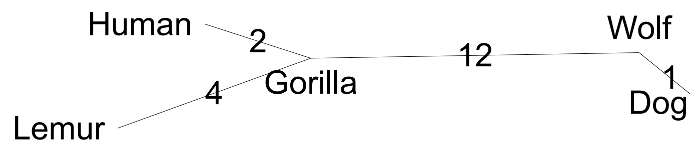


Figure 6.1: Minimal spanning tree – mammals

agglomerative one [GR69]. Because splitting a minimal spanning tree is equivalent to single link agglomerative clustering, it suffers from the same drawbacks as the agglomerative clustering techniques that were discussed in Section 5.6.1. Section 6.2.2 describes a technique somewhat similar to this that is used for determining clusters of class members that may serve as the basis for the *Extract Class* refactoring.

6.1.2 Betweenness clustering

Betweenness clustering algorithms [GN02, Bra01] are graph-based divisive clustering techniques that separate a connected graph into disconnected subgraphs. Betweenness clustering gets its name because it operates by iteratively removing the relatively few edges that separate (are most between) densely interconnected clusters (a.k.a *communities*) of nodes.

Betweenness clustering has been applied to many domains, including social networks [GN02, RB07, For10], scientific collaboration networks [New01], biological food webs [GN02], and software componentization [DYM+08]. In social networks, the graphs typically consist of nodes that represent people or organizations, and edges that represent some kind of communication of information from one entity to another. Groups of nodes with many communication connections are communities.

The Girvan-Newman betweenness clustering algorithm [GN02] works by iteratively removing the edge of a graph that has the highest betweenness value, where the *betweenness value* of an edge is the number of shortest paths between pairs of nodes that pass through it. (If there are multiple paths between two nodes that tie for the shortest distance, each edge on each path accrues an equal, fractional weighting.) Figure 6.2 contains two graphs whose edges are labeled with betweenness values. In both the connected graph on the left and the disconnected

graph on the right, the edge between F and G has a value of 4, because it is on the shortest paths between (F, G), (G, F), (G, H), (H, G), but no others.

If the graph on the left represented a collaboration network, nodes A through D might represent researchers in artificial intelligence (AI) and nodes E through H researchers in refactoring. The edges might indicate that the people represented by the nodes were co-authors of a paper, and communicate information relevant to the paper. Thus, the edge from AI researcher D to refactoring researcher E would represent the rare occurrence of co-authorship of a paper (and communication of information) between the AI and refactoring communities.

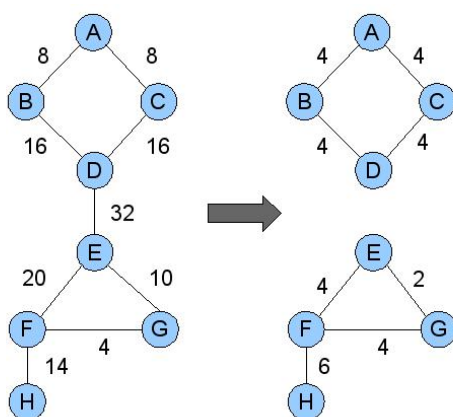


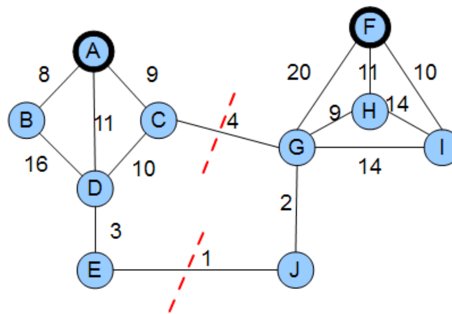
Figure 6.2: Betweenness clustering – edge betweenness values

Figure 6.2 shows the graph on the left being transformed into the graph on the right by removing the edge with the highest betweenness. For this example, the first iteration of the betweenness clustering algorithm removes the edge between D and E, leading to two disconnected subgraphs/clusters, as shown on the right side of the figure. Depending on the graph's structure, many edge removals may be required before a cluster of nodes becomes disconnected from the rest of the graph. After each edge removal, the algorithm recalculates the betweenness values of the edges prior to the next edge removal. The successive removal of edges will disconnect the connected portions of the graph and form additional clusters.

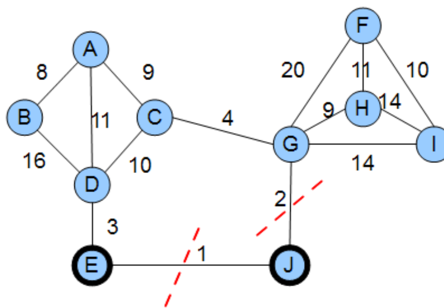
In the variant of betweenness clustering described above, the edges are undirected and unweighted – the edge labels indicate the betweenness values as though all edges had a weight of 1.0. Newman [New10] describes additional variants for directed and weighted edges that we may investigate in the future.

6.1.3 Max flow/min cut

The maximum flow problem is to determine the maximum amount of something (e.g., information) that can flow through a network from a source to a destination, and along which paths. This can be cast as a graph problem. Given a weighted graph that contains a designated source node and destination (or *sink*) node, maximum flow algorithms [FF09] determine the maximum flow from source to destination and how that flow is distributed across the various edges. In general, maximum flow algorithms can also be used to determine the minimum cuts in a graph, that is, the edges with the lowest combined weights that can be removed to separate subgraphs containing the source and destination.



(a) Separating A and F



(b) Separating E and J

Figure 6.3: Minimum cuts

The minimum cut for a graph can vary depending on the choices for the source and destination. Figure 6.3 shows two different determinations of a minimum cut for a graph, produced by using two different sets of source and destination nodes. Assume that the graph depicts information flow. There are two groups that communicate relatively much information – nodes `A` through `D`, and nodes `F` through `I`. Nodes `E` and `J` are relatively isolated outliers that do not communicate much information.

When the source and destination are representative members of distinct groups, e.g., `A` is the source and node `F` is the destination, then the minimum cut algorithm can often detect the groups of highly interacting nodes. Figure 6.3(a) shows the minimum cut between `A` and `F`, the edges (`C`-`G`) and (`E`-`J`). The newly separated graph components contain the groups of nodes that most communicate information.

When the source and destination are not representative members of distinct groups, e.g., the source and destination are outliers, then the minimum cut algorithm is less likely to detect the groups of highly interacting nodes. Figure 6.3(b) shows the minimum cut between the `E` and `J`, the edges (`E`-`J`) and (`G`-`J`). The newly separated graph components do not separate the highly interacting groups.

Determining the minimum cut is somewhat similar to separating a graph using betweenness clustering. In both cases, one is determining a cut set that disconnects a graph. However, these algorithms are useful in different circumstances. Max flow/min cut is useful for disconnecting a graph when a meaningful source and destination can be chosen; betweenness clustering can separate a graph without a specified source and destination.

6.2 Related work – applying graph-based clustering to software

A few research groups have applied graph-based clustering techniques to refactoring software. This section discusses their approaches and the strengths and weaknesses of those approaches. Section 6.4 provides further evaluation, comparing some graph-based and agglomerative clustering approaches to refactoring.

6.2.1 Modularization

Dietrich, et al. [DYM^{+}08] created the BARRIO Eclipse plug-in as part of a tool that will suggest a list of refactorings to make programs more modular by better allocating classes to packages and containers. BARRIO detects and visualizes clusters of Java classes in dependency graphs. The nodes in the graphs are Java classes, and the edges indicate *extends*, *implements*, or *uses* relationships between the classes. BARRIO provides filters that affect the graph display, but the use of

filtering with clustering was listed as future work.

They use a variant of betweenness clustering to form clusters. They modified the betweenness clustering algorithm to remove multiple edges in a single iteration, when all of those edges have the same weight. This is not an optimization over the Girvan-Newman betweenness algorithm; it produces different results.

These researchers do not report on their criteria for determining what constitutes a good cluster, e.g., they do not stop clustering when a cluster exhibits any particular properties. Instead, their clustering algorithm runs for a specified number of iterations. Consequently, an iteration may break apart a cohesive package.

They recognize two refactorings. When multiple clusters exist in a package, they recommend splitting the package into multiple packages. When a cluster contains multiple packages, they recommend merging those packages into a single package. Their paper emphasizes visual aspects of component determination and does not indicate whether they performed any refactorings based on the betweenness clustering results.

6.2.2 Extracting classes

Researchers at the University of Salerno [BDLO11] use max flow/min cut to help determine how to split large classes. Their technique constructs a fully-connected graph whose nodes are the class's methods and whose edge weights are the similarity scores between the connected nodes. (The similarity function is discussed in Section 5.2.3.) Next, all edges with a weight below a certain threshold are removed. In contrast to their "two step" clustering technique described in Section 5.2.3, the edge removal step is not intended to disconnect the graph. They do not state how they avoid disconnecting the graph.

After edge removal, they form clusters using a max flow/min cut algorithm. After setting the source and destination nodes to the nodes of the most dissimilar methods, they remove the minimum edge cut set (a set of edges whose removal disconnects the graph, and the sum of whose weights has the minimum possible value) as determined by max flow/min cut. The two subgraphs form the basis for the new classes. They acknowledge that their choice of using the most dissimilar nodes as the source and destination can cause problems, because the source and destination may be outliers.

As with their two-step technique, they evaluated their *Extract Class* approach

by randomly combining pairs of classes from well-designed systems and then seeing whether their technique would extract the original classes when applied to the hybrid classes. In this study, they took some LCOM cohesion measurements on the classes before the merger, after the merger (the hybrid), and after the extract class was performed. The LCOM measurements indicated that the hybrid classes were significantly less cohesive than the original classes and the extracted classes, and that the extracted classes were nearly as cohesive as the originals. However, as mentioned in Section 2.1.3, LCOM scores correlate with class size. In general, one would expect the larger, hybrid classes to have higher LCOM scores than the smaller classes to which their LCOM scores are compared, particularly when the classes that were merged were unlikely to be highly related.

It is not clear what the underlying conceptual basis is for the Salerno algorithm – the graph's edges indicate similarity measures, not structural connectedness/information flow. Their approach somewhat resembles the separation of a minimum spanning tree as described in Section 6.1.1, which gives the same result as single link agglomerative clustering. Instead of separating a minimal spanning tree by removing the edges with the greatest distances, their algorithm separates a graph by removing the cut set determined by max flow/min cut, which typically consists of a small number of edges with low similarities.

6.3 Experiments – applying betweenness clustering to Java classes

Based on the idea that interactions between methods and attributes are analogous to the interactions between people, we use betweenness clustering to try to identify the clusters of methods and attributes within a large class that most belong together. In the graphs of large classes, the nodes represent the class's members, and the undirected edges represent dependencies between members, e.g., a method calling a method or accessing an attribute. The next section explains how betweenness clustering works on the members of a class using our interactive betweenness view. Section 6.3.2 discusses some experiments we ran on open source classes to determine whether betweenness clustering could effectively determine how to split large classes.

6.3.1 Visualizing betweenness clustering

ExtC provides a betweenness clustering view for visualizing the activity of the betweenness clustering algorithm on undirected intraclass dependency graphs. The view shows the algorithm removing edges from the graph, changing betweenness values of the remaining edges, and forming clusters. It is based on a demo provided by the JUNG graph framework [OFWB03] that illustrates betweenness clustering using social network data.

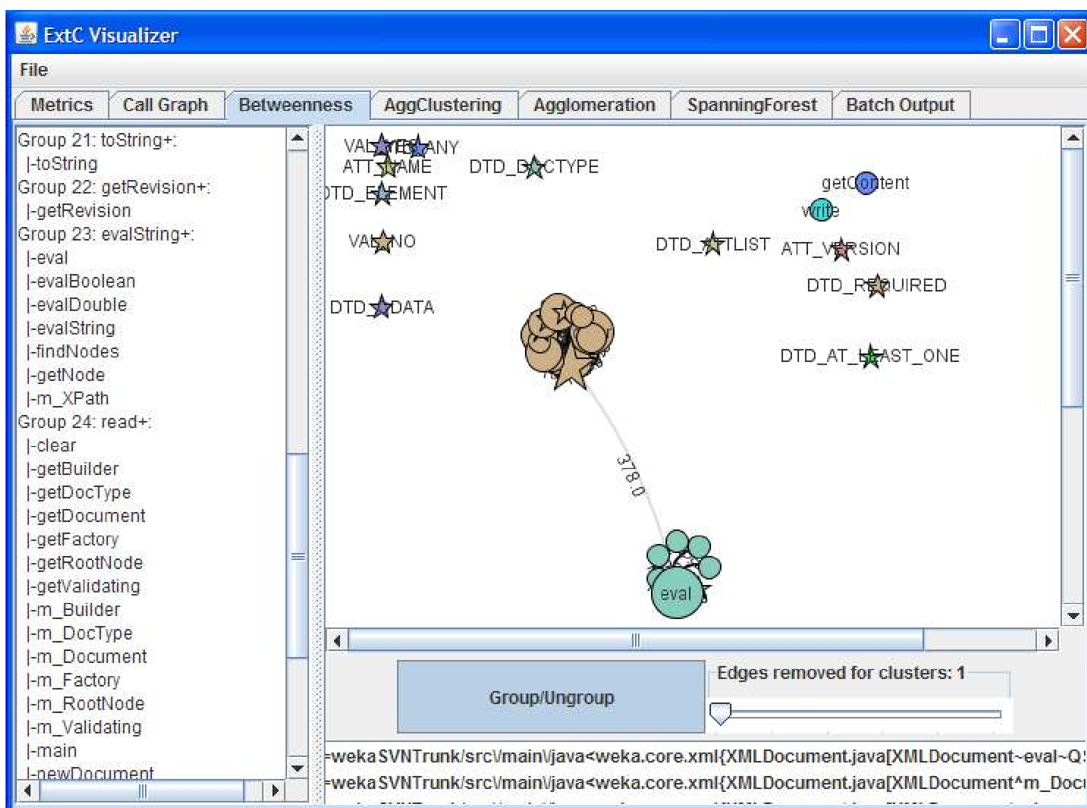


Figure 6.4: Betweenness view

Figure 6.4 shows the betweenness clustering view, whose primary display components are an intraclass dependency graph, and a text area. In the graph, all of the nodes in a cluster (i.e., connected subgraph) are shown in the same color, and different clusters have different colors. The text area on the left displays the class members in each cluster.

The view has a slider at the bottom that allows the user to specify the number of edges to remove. Dragging it causes edges to be removed, which causes the betweenness values on the remaining edges to change. When the removed

edges constitute a cut set, the appearance of a new color notifies the user of the appearance of a new cluster.

As a simple example, consider the `PersonCarDirect` class from Appendix A.3, that contains the mixed characteristics of both a person and a car. Its primary methods (`getPersonName`, `getPersonInfo`, `getCarDescription`, and `getCarInfo`) ultimately access the fields one would expect. From the source code of this class, ExtC creates an intraclass dependency graph like the one shown in Figure 6.5, where the edges are labeled with betweenness values. (Some of these nodes have been manually repositioned for readability.)

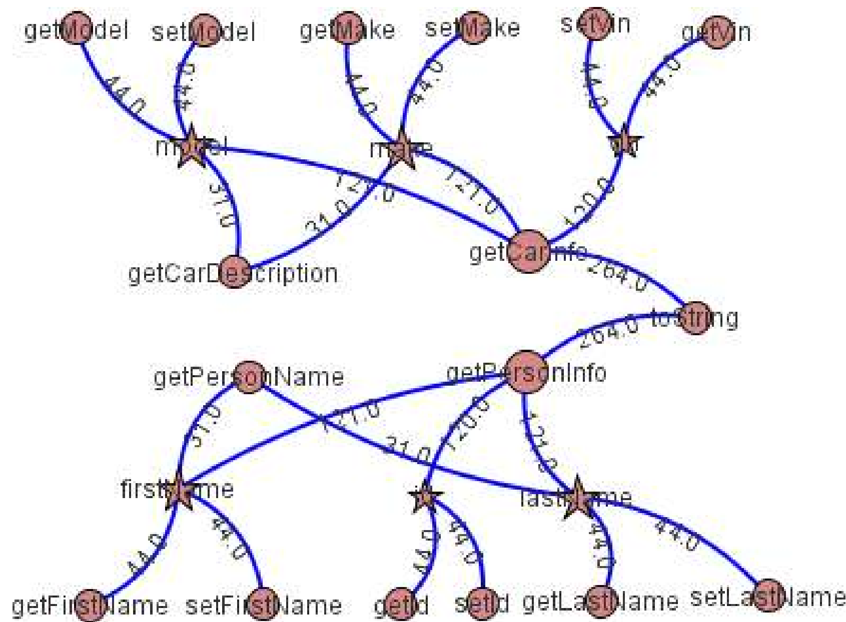
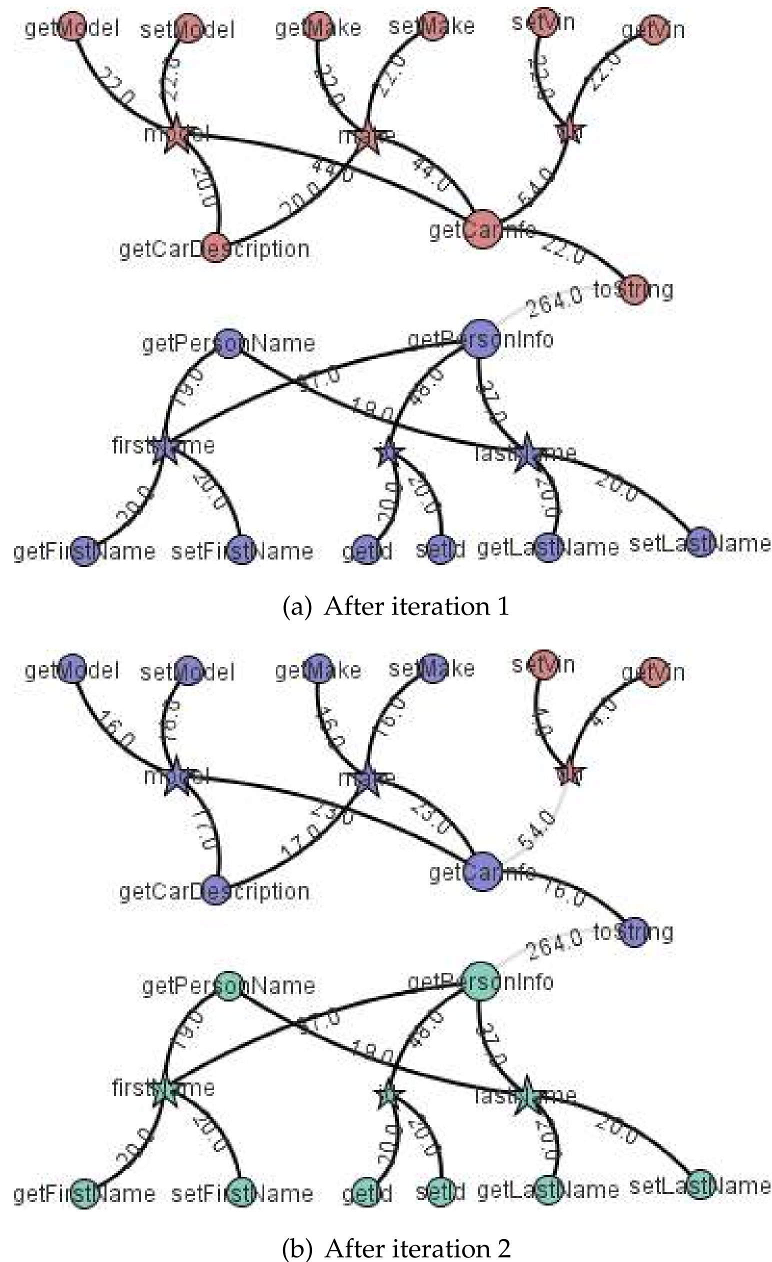


Figure 6.5: Betweenness clustering – PersonCarDirect before start

As shown, the top half consists of a subgraph containing the car-related methods and attributes, while the bottom half contains the person-related methods and attributes. Tying these together is the `toString` method, which calls `getCarInfo` and `getPersonInfo`. In Figure 6.5, the edges with the maximum betweenness are the ones leading from `toString`, near the center right of the graph. (When two edges have the same betweenness score, the JUNG implementation of the betweenness algorithm chooses one arbitrarily. When this occurs, multiple tests may produce slightly different results, and there is no clear reason to prefer one result over the alternatives. This situation occurs here, because the example is a symmetric, artificially constructed test case. With large, real-life classes, like the open source classes used in the experiments in Section 6.3.2,



(a) After iteration 1

(b) After iteration 2

Figure 6.6: Betweenness clustering – PersonCarDirect iterations 1 and 2

such situations are rare for the edges with the initial highest betweenness scores.)

As the analyst drags the slider, the betweenness clustering algorithm will first remove one of the edges leading from `toString`. This will effectively break the graph in two (Figure 6.6(a)). One subgraph will be composed primarily of the methods and attributes pertaining to a person, and the other subgraph will have methods and attributes pertaining to a car. At this point, the algorithm will recompute the betweenness scores for the edges and once again remove the

highest scoring edge.

After removing the second edge, there are three clusters, as shown in Figure 6.6(b):

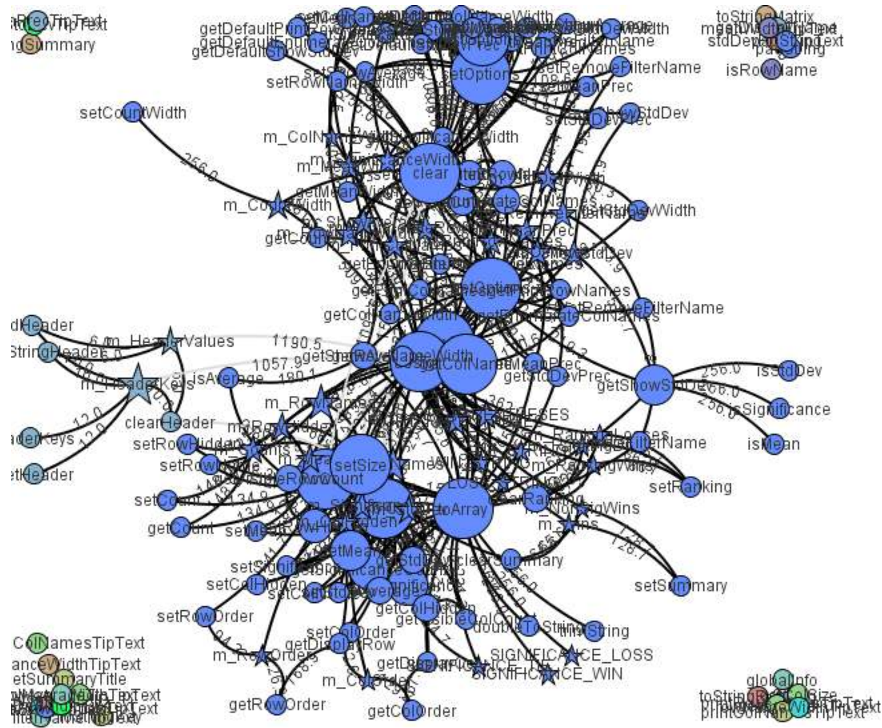
1. The top left part of the graph, consisting of the methods and attributes pertaining to cars, except for those pertaining to the car's `vin`.
2. The top right part of the graph, consisting of the methods and attributes pertaining to the car's `vin`.
3. The bottom part of the graph, consisting of the methods and attributes pertaining to a person.

A user can choose to display a graph in either "ungrouped" or "grouped" form. When ungrouped, each node in the ungrouped view is labeled with the corresponding member name, and the edges between the nodes are labeled with their betweenness values. The nodes are laid out using a force-directed algorithm [FR91]; however, analysts can reposition nodes manually to better show relationships between particular class members. It is relatively easy to discern clusters in the ungrouped view when the graph is composed of fewer than 50 nodes, as in Figure 6.6.

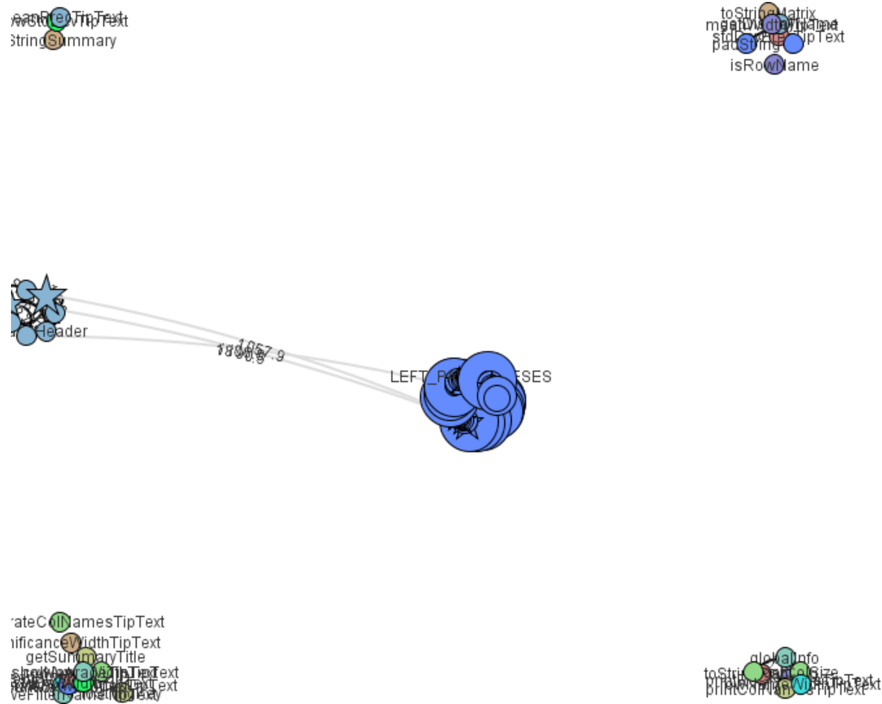
However, larger intraclass dependency graphs tend to make the unclustered graph view resemble a ball of yarn, so the grouped view may be preferable. Consider Weka's `ResultMatrix` class, which has over 300 class members and whose graph has 27 disconnected subgraphs prior to any edges being removed. The graph's many nodes and edges can make analysis difficult. For example, Figure 6.7(a) shows the ungrouped view after eight edge removals has produced a new cluster. The two largest clusters are slightly different shades of blue and close together spatially.

By clicking a button, analysts can toggle to the grouped view (Figure 6.7(b)). This lays out the nodes of a cluster in a tight circular pattern. In the grouped view, only a single node in a cluster is labeled to help eliminate clutter, and most edges within the cluster are obscured. The most noticeable edges in the grouped view are the gray inter-cluster edges, the ones that were removed to form the clusters. If the clusters are used as the basis for forming revised classes, the relationships represented by the gray edges represent coupling between the classes. Because there is a preference for low coupling between classes, a large number of gray edges is a counter-indicator for a potential refactoring.

Certain methods and fields obscure the fundamental structure of the class.



(a) Ungrouped



(b) Grouped

Figure 6.7: Betweenness clustering views

As discussed in Section 4.2, the graph view provides options for whether to include `toString`, loggers, etc., and those options apply to the betweenness view also. However, there may be less predictable causes of clutter in graphs that obscure the fundamental purpose of the class. For example, a particular company might mandate that its programmers use certain information gathering or debug methods. To help alleviate this problem, the betweenness clustering view provides a list by which the user can optionally remove nodes from the graph. The nodes on the list are ordered based on their betweenness values, because nodes that are connected to many other nodes tend to have high betweenness. Because high placement on the list does not necessarily indicate noise, the selection of nodes to filter out is left to the user. By filtering out noisy nodes, analysts can obtain better clustering results.

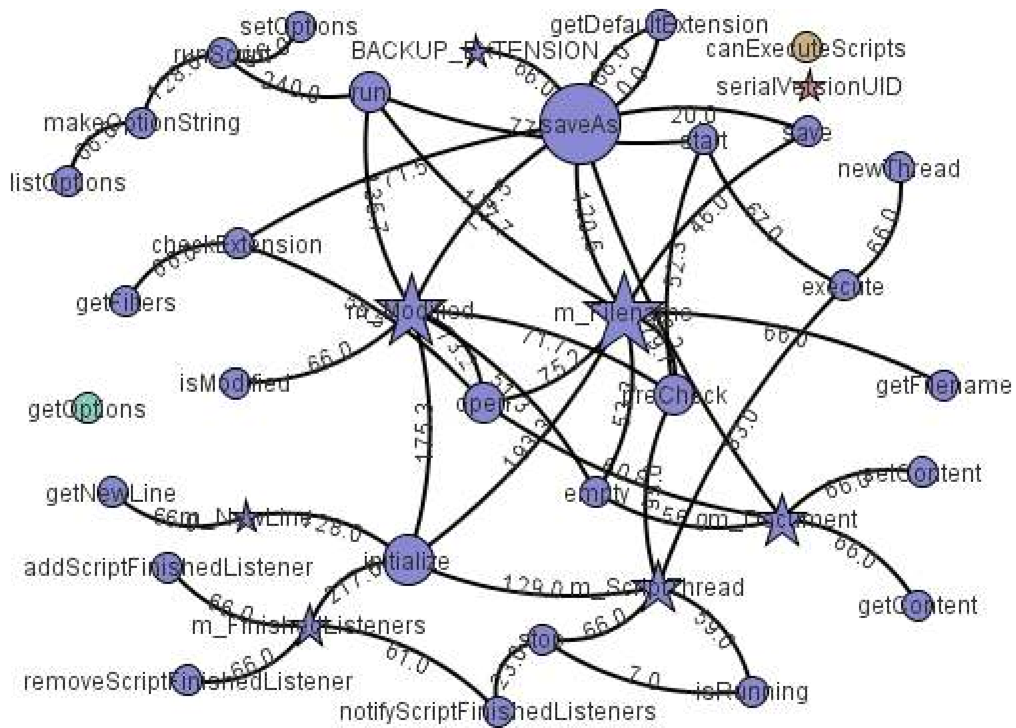
Consider Weka's `Script` class. Figure 6.8 shows betweenness clustering on the members of `Script` when no members are filtered out. Figure 6.8(a) shows the initial state. Figure 6.8(b) shows the formation of the first new cluster after one edge has been removed. This cluster appears in the upper left and contains four members – `listOptions`, `makeOptionString`, `runScript`, and `setOptions`.

The `Script` class contains an `initialize` method that only sets the values of five attributes. If we filter out `initialize`, betweenness clustering produces different results, as shown in Figure 6.9. Figure 6.9(a) shows the initial state, without `initialize`. Figure 6.9(b) shows the state after two edges have been removed and the first additional cluster appears – a cluster of nine members that does not include the members produced by the non-filtered betweenness clustering illustrated in Figure 6.8.

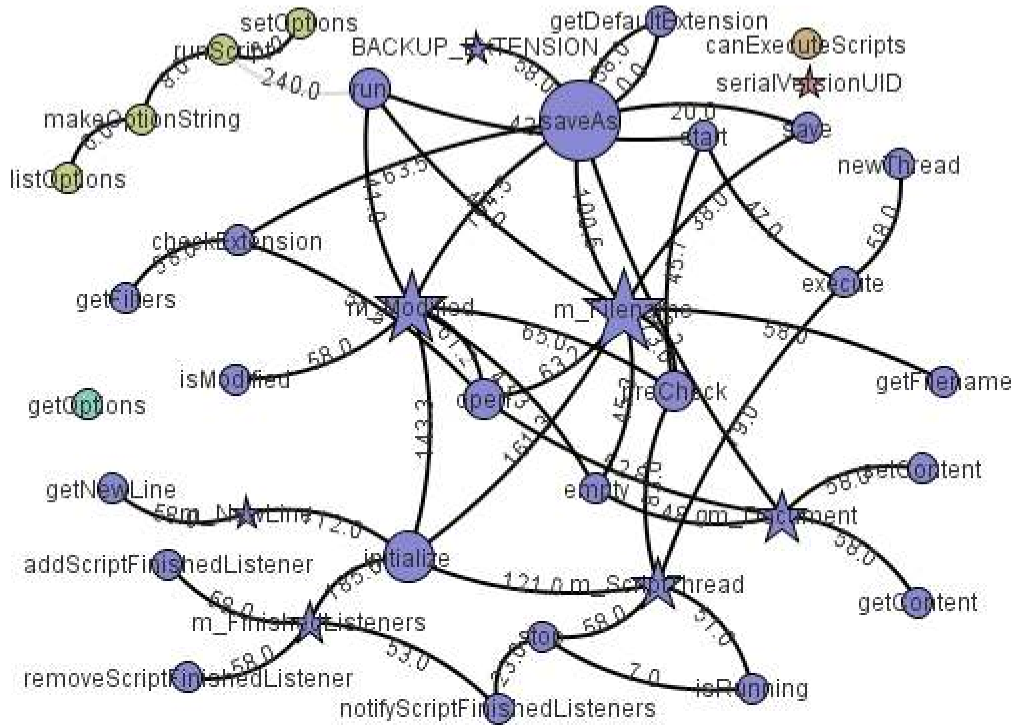
In general, it requires some domain expertise to interpret which are the noisy nodes and which are the "best" clustering results. Our betweenness visualization helps with that analysis.

6.3.2 Refactoring god classes using betweenness clustering

Our visualizations convinced us that betweenness clustering was useful for determining how to split some classes. For example, betweenness clustering easily identifies the preferred divisions of the `PersonCarDisjoint` and `AnonymousPersistence` classes that were used as a preliminary test for analyzing the effectiveness of agglomerative clustering algorithms in Section 5.3.1. The next

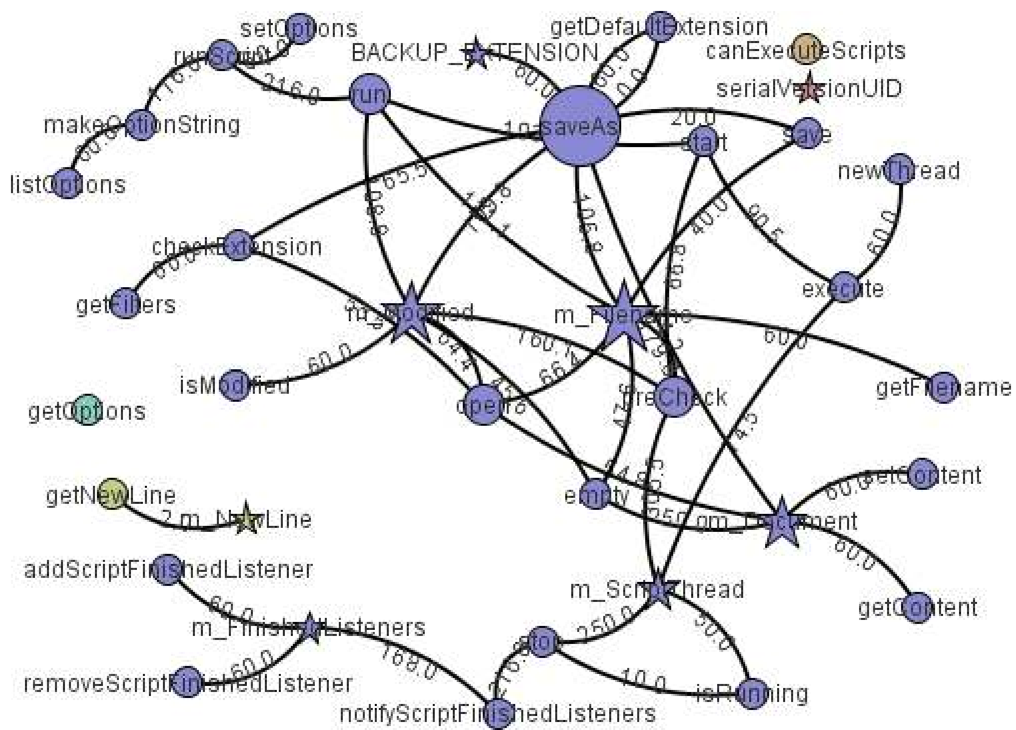


(a) Iteration 0

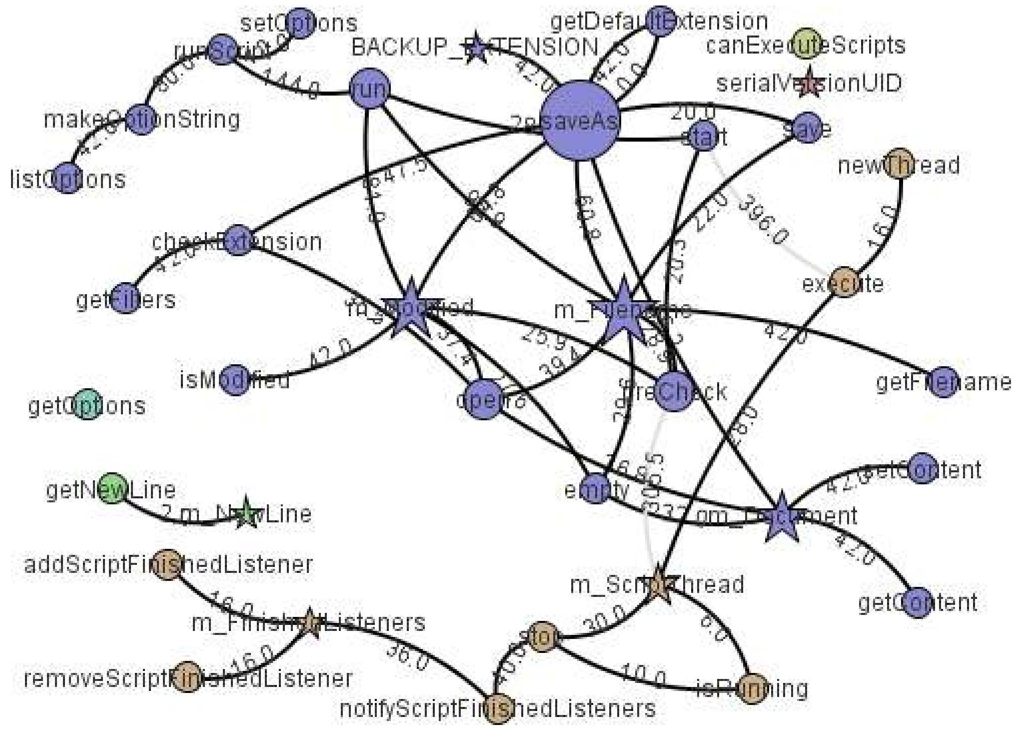


(b) Iteration 1

Figure 6.8: Betweenness clustering – Weka's Script class



(a) Iteration 0



(b) Iteration 2

Figure 6.9: Betweenness clustering with filtering – Weka's Script class

step was to test betweenness clustering on a less controlled input set, four mature open source Java projects – FreeCol, Heritrix, Jena, and Weka. Appendix B.2 provides detail on these open source projects.

The tests were performed on a low end PC, a Dell Vostro 1000 PC with an AMD Athlon(tm) 64 X2 Dual-Core Processor TK-57 1.90 GHz with 1.87 GB of RAM. Because other applications were running at the same time as the tests, any execution times reported are approximate and likely to be on the high side.

The tests consisted of the following steps:

1. We used software metrics to identify large noncohesive classes in the open source projects.
2. Betweenness clustering separated each class's members into at least two clusters.
3. With the two largest clusters as input, we used IntelliJ's *Extract Class* refactoring to split each of the original classes into two classes.
4. Software metrics quantified the changes caused by the refactoring.

Identifying large, noncohesive classes

We used Metrics2 to collect the size and structural measurements on the classes, and ExtC to collect the semantic measurements. All measurements were stored in a database. The query described in Section 4.1 identified thirty god classes out of approximately 3000 top level public classes. These classes are listed in Appendix B.2.

Clustering the members

An examination of the structure of the thirty matching classes revealed that there was a high degree of disconnectedness in their call graphs, i.e., even before edge removal, there were distinct structural clusters. After removing some special methods (see Appendix C.1 for detail) that might link together functionally unrelated members, we checked the classes to determine the number of connected subgraphs and the size of those subgraphs. Because seven is approximately the average number of methods per class [LM06], we used that as a threshold for large and small clusters. For the thirty classes:

1. None had a completely connected dependency graph.

2. The average number of clusters per class was 11.2, and the median was 8.5 clusters per class.
3. Multiple large clusters were relatively rare. Only six (20%) of the classes had more than one cluster of seven or more members, and no class had more than two such clusters.
4. Multiple small clusters were extremely common. Fifteen (50%) of the classes had seven or more clusters of fewer than seven members.

Appendix C.3.1 contains detailed data about the cluster sizes.

ExtC provides a batch mode of operation to cluster the members of the classes that match the god class query. The results determine whether the classes are good candidates for an *Extract Class* refactoring. There are two conditions that trigger a refactoring: (1) when betweenness clustering identifies two clusters of seven or more without removing any edges, (2) when the first new cluster formed by edge removals consists of seven or more members. On average, the clustering step took less than six seconds per class. Based on the results of the clustering, 12 of the 30 classes did not warrant refactoring, because they did not produce two clusters with at least seven members each. It is worth noting that, had the criterion for the number of members in a cluster been reduced from seven to six, an additional four classes would have warranted refactoring.

Refactoring the large classes

The two largest clusters formed the basis for the revised classes. The class members of the second largest cluster were the basis for the extracted class, and they served as inputs for IntelliJ's Extract Class refactoring, as described in Section 4.4. Except to correct compilation errors, no manual editing was done to the classes generated by IDEA.[1] It typically took from 15 to 30 minutes to perform each refactoring, with most of the time spent specifying the class members to be extracted by the IntelliJ GUI.

Quantifying the changes caused by refactoring

Metrics2 calculated most of the measurements on the software. The data collected included the number of fields (NF), number of methods (NM), weighted method

[1] E.g., see the defects at `http://youtrack.jetbrains.com/issue/IDEA-24608` and `http://youtrack.jetbrains.com/issue/IDEA-50421`

count using computational complexity (WMC), and six structural cohesion metrics (LCOM, LCOM*, TCC, DC_D, LCC, and DC_I). ExtC provided the semantic cohesion (C3V) of the classes. Appendix C.3 contains the detailed metric data.

Table 6.1: Average metric values (excluding cohesion)

Class	NF	NM	WMC
original	26	54	132
modified	24	52	114
extracted	3	14	29

Table 6.2: Cohesion measurements

Class	LCOM	LCOM*	TCC	DCD	LCC	DCI	C3V
Original	1392	0.97	0.19	0.25	0.48	0.51	0.09
Modified	1117	0.96	0.21	0.25	0.49	0.51	0.09
Extracted	133	0.47	0.65	0.70	0.83	0.89	0.32

Table 6.1 summarizes the metric results for the number of fields, number of methods, and weighted methods per class. It shows a decrease in the average values for number of fields, number of methods, and weighted methods per class for both the modified and extracted classes. Almost all classes showed a decrease in complexity as measured by WMC. However, the sum of the values for the modified and extracted classes exceeds the values for the original classes, especially for the number of methods. As discussed in Section 3.1.3, this is an expected consequence of strict refactoring.

Table 6.2 summarizes the cohesion results for the same classes. (Remember that LCOM and LCOM* are "lack of cohesion" metrics, so higher numbers for these indicate less cohesion.)

Figure 6.10 uses box plots to summarize the cohesion distributions of the normalized cohesion metrics for the original, modified, and extracted classes. The box boundaries are the values for the first and third quartiles; the lines within each box represent the median. The lines that extend from the boxes reach to the furthest data value that is no more than 1.5 times the size of the box. The outliers

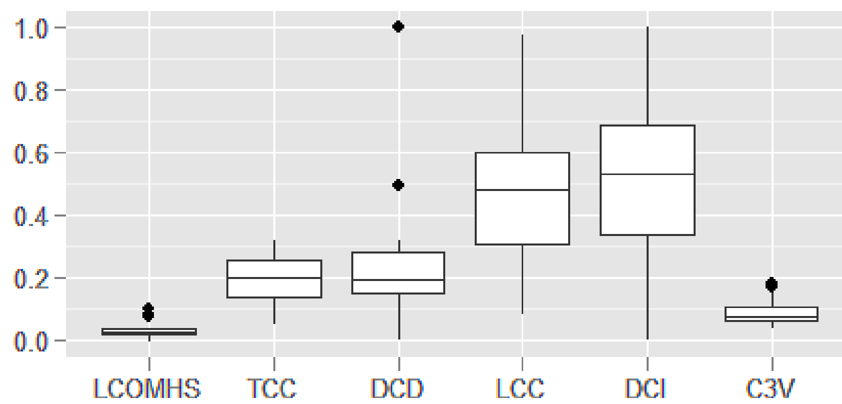
beyond this limit are shown as dots. Because LCOM* is a lack of cohesion metric, the values shown in the figures for the column labeled "LCOMHS" is (1 - LCOM*). This makes it easier to compare the cohesion metrics – better cohesion scores will always be towards the top.

The cohesion values for the original classes (Figure 6.10(a)) and the modified classes (Figure 6.10(b)) are not much different. For the normalized metrics (all except LCOM), the average cohesion results for the modified classes differ from those of the original classes by 0.02 or less. There are two main reasons for the overall lack of improvement in cohesion for the modified classes. First, the classes being refactored are highly noncohesive, typically having many structurally disconnected parts. After the refactoring, many of the small, structurally disconnected portions of the original class remain as small, structurally disconnected portions of the modified class. Second, for every public method that was part of the original class, there is a proxy method in the modified class. These proxy methods do not interact with most of the attributes of the modified class, so they contribute to the apparent lack of cohesion.

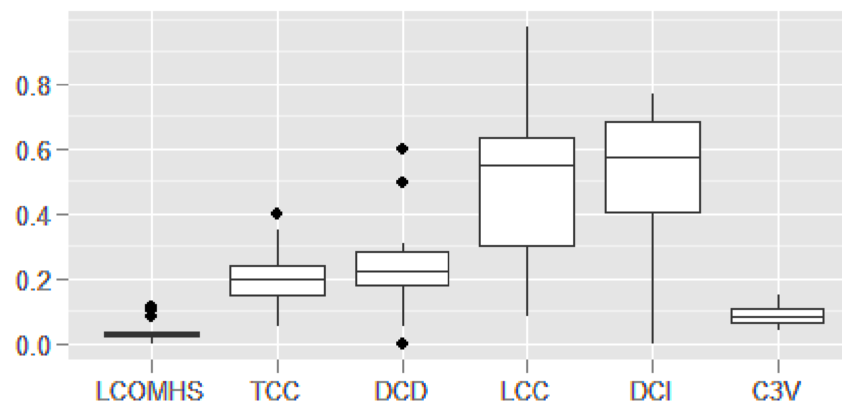
On the other hand, the average cohesion scores for the extracted classes (Figure 6.10(c)) are generally much better than those of the original class. For the normalized metrics, the average improvements ranged from 0.23 for C3V to 0.5 for LCOM*. This increased cohesion for the extracted class is due to betweenness clustering identifying a structurally well-connected portion of the original intraclass dependency graph, and this well-connected subgraph is used to identify the members that form the extracted class.

Only three (8.3%) of the 36 modified and extracted classes had a majority (four or more) of the seven cohesion scores worsen by 0.01 or more, and even in these cases, at least two cohesion scores improved. For the other 33 (91.7%), the majority of the cohesion scores improved or stayed the same. None of the original classes were split into classes where both the modified and extracted classes had a majority of cohesion metrics show a decrease in cohesion. Based on these results, it appears that betweenness clustering provides useful assistance for the programmer who is trying to break apart god classes.

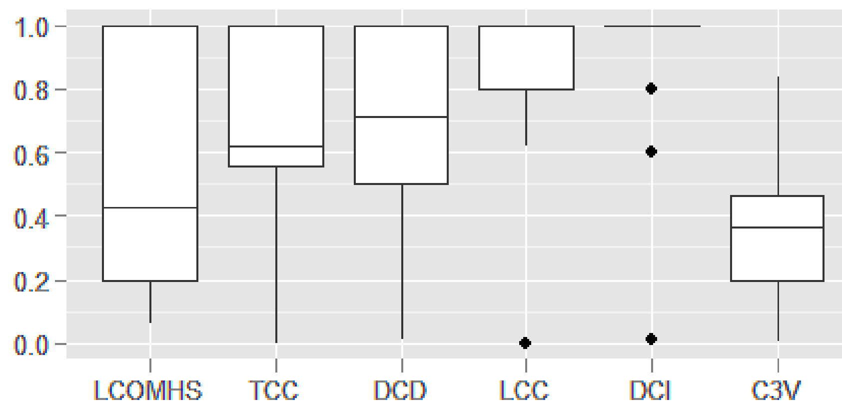
It is worth examining some of the classes whose cohesion decreased. These do not necessarily indicate a bad outcome of refactoring. Consider Weka's `XML-Document` class, which was one of the classes whose modified class had lower cohesion than the original. The betweenness clustering, shown in Figure 6.11, identified one cluster that produced a highly cohesive extracted class. This cluster



(a) Original class



(b) Modified class



(c) Extracted class

Figure 6.10: Cohesion distributions

is brown and towards the upper right of the figure. The cluster contains five methods that all call the `eval` method, which in turn accesses the sole attribute – `m_XPath`. Because of this structure, the extracted class produced from this cluster

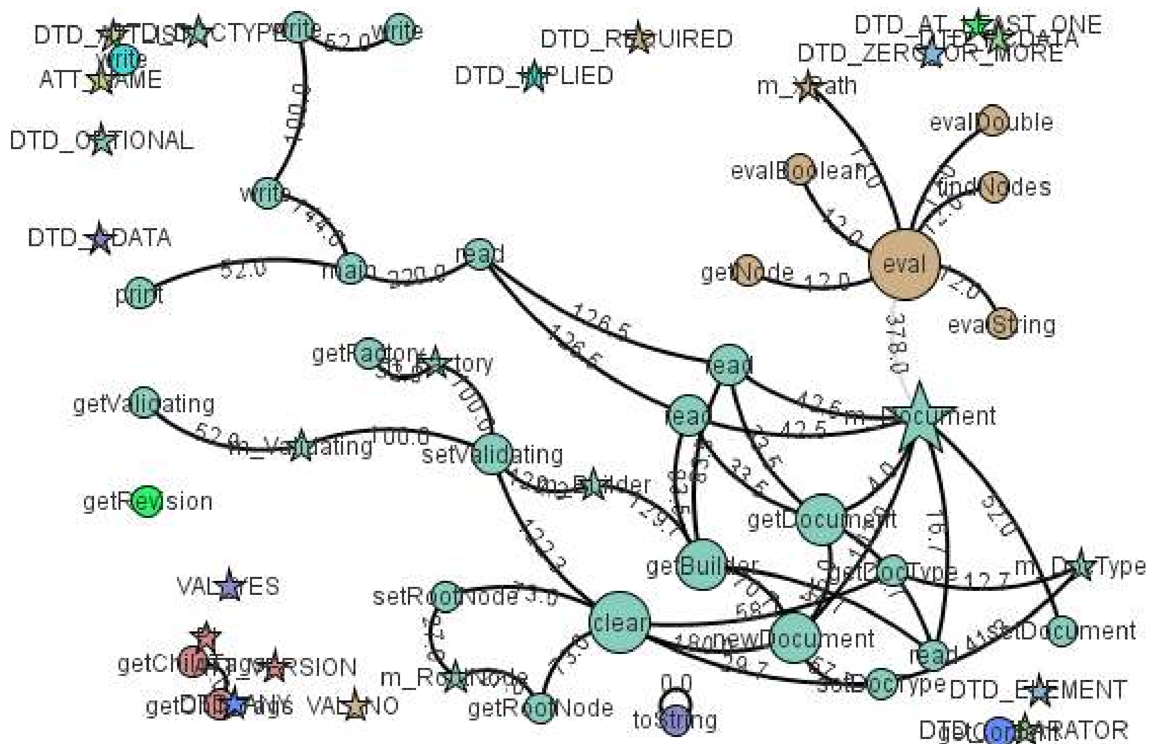


Figure 6.11: Betweenness clustering – XMLDocument

has maximal TCC, DC_D, LCC, and DC_I cohesion scores. On the other hand, the modified class is formed from the remainder of the class members. Because there are many small clusters, and because a cohesive portion of the original class (the brown members) has been extracted, the modified class now has lower cohesion than before. Nevertheless, this refactoring is arguably a better class structure than existed previously.

There are also some exceptional extracted classes. The extracted class for Heritrix's `CandidateURI` class has a cohesion of 0.0 for many of the metrics. It is composed solely of methods, except for a field that references the modified class, that is used only by the constructor. The extracted class for Jena's `Rule` has a similar problem – it is composed exclusively of static methods – those metrics that require access to attributes give it a score of 0.0, even though the methods are inter-related. These oddities are due more to idiosyncrasies of the cohesion metrics than they are to poor refactoring.

6.3.3 Using betweenness clustering on directed graphs

The betweenness clustering experiments discussed in Section 6.3.2 operated on undirected versions of intraclass dependency graphs. We also investigated betweenness clustering working on directed graphs.

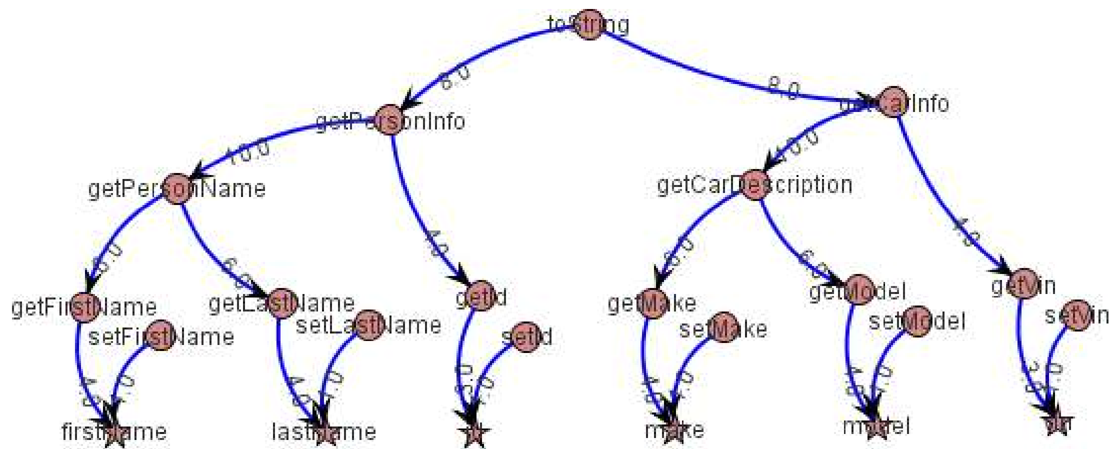
Betweenness clustering with directed intraclass dependency graphs forms clusters that seldom produce effective object-oriented classes. Consider Figure 6.12 which shows betweenness clustering on the `PersonCarIndirect` class, whose source code can be found in Appendix A.4. As with the other `PersonCar` classes discussed previously, `PersonCarIndirect` should split into two clusters, one having the class members pertaining to persons and the other having class members pertaining to cars. Instead, the first iteration of this clustering (Figure 6.12(b)) produces a cluster that contains a subset of the person functionality. Due to results such as these, we recommend using betweenness clustering on undirected graphs over betweenness clustering on directed graphs for extracting classes.

6.4 Evaluation of graph-based clustering techniques

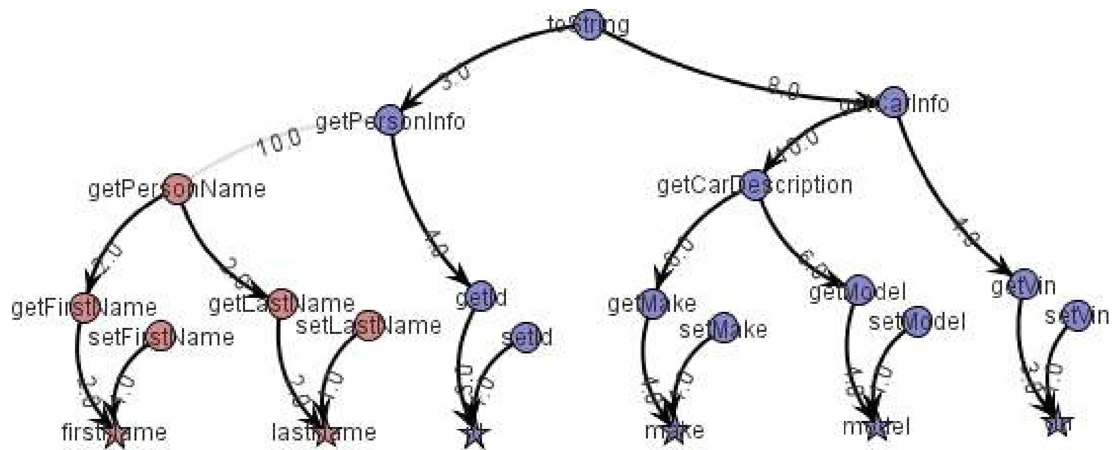
In contrast to agglomerative clustering, graph-based divisive clustering is a cognitive match for *Extract Class*, where the goal to split a class into more maintainable ones. Graph-based clustering can also be useful for determining inputs to other refactorings, e.g., *Move Method* and *Move Field*, as discussed in Section 5.2.2.

In Section 2.1.3, we noted that cohesion researchers value connectivity between a class's members and often represent that connectivity using graphs. In particular, it is worth noting that betweenness clustering and max flow/min cut are similar in spirit to the previously discussed (Section 2.1.3) ICBM technique [ZXZY02] for measuring cohesion, which relies on determining the cut sets for a dependency graph. In fact, the creators of ICBMC mention that it could be used as a basis for class restructuring.

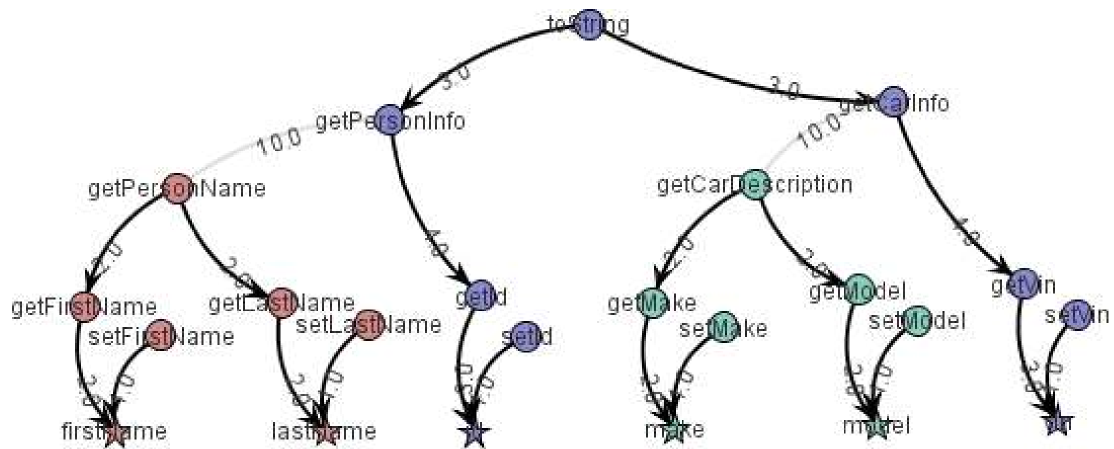
The betweenness clustering and max flow/min cut algorithms form clusters by splitting graphs. In effect, these algorithms use extended structural information in addition to local structural information, i.e., the extended relationships of members to other members is embedded in the intraclass dependency graphs on which the algorithms work, and the graph-based clustering algorithms rely on



(a) Iteration 0



(b) Iteration 1



(c) Iteration 2

Figure 6.12: Betweenness clustering on directed graphs

this information when determining the clusters.

Furthermore, these graph-based algorithms retain most existing intraclass member relationships. These divisive techniques will not combine entities (attributes and methods) that were not previously together; instead they divide those entities that were already together into meaningful sets. This is advantageous for maintaining existing interfaces, particularly for structure-based refactorings including *Extract Class*, *Move Method*, and *Move Field*.

The edges that were removed from the graph to form a cut set represent calling relationships that will be re-established between the members of the newly created classes. Because the graph-based divisive clustering algorithms tend to produce small cut sets, there will be few interclass calling relationships (except for those created through the use of proxy methods), and consequently, relatively little coupling between the refactored classes.

The clusters produced by techniques that use distance functions (e.g., many agglomerative techniques and k-means) have no innate mechanism for maintaining existing structural relationships, nor do they have any built-in way of limiting interclass connections. The members in the clusters are a result of the operation of the distance functions, and the functions previously discussed in Section 5.2 only use local connectivity information in determining how to cluster.

Graph-based algorithms are not a panacea; for instance, they do not provide a way of associating small, dissociated parts of the graph (outliers) with larger parts. Chapter 7 discusses how we can compensate for this deficiency by combining graph-based clustering techniques with agglomerative clustering techniques.

6.5 Contribution summary

Despite their potential benefits, graph-based clustering techniques have been underutilized by the refactoring community. As discussed in Section 2.1.3, many researchers who are studying object-oriented cohesion represent class structure using graphs. Meanwhile, researchers interested in detecting highly associated communities of individuals have been developing clustering techniques based on partitioning graphs. Yet, we believe that our use of betweenness clustering [CAGN09] is the first application of graph-based techniques for determining how to refactor classes based on their structure.

Betweenness clustering is good at forming groups based on structure, but it

lacks the capability of using domain knowledge. Programmers may have difficulty discerning class structure, but have applicable domain knowledge. Our ExtC Eclipse plug-in helps bridge the gap by providing programmers an interactive visualization of betweenness clustering operating on class structure. The formation of clusters within a visual graph display helps the programmer locate important interrelationships between class members, and a filtering capability allows iterative exploration of the class structure as clutter is removed.

The filtering capability has the same foundation as the graph restructuring discussed in Section 2.4.1. In that section, the graph was restructured to remove noise from the cohesion measurements. It has a similar purpose in the context of graph-based clustering; it removes relationships that obscure the primary intent of the class and facilitates more meaningful clustering.

Section 6.3.2 discussed our semi-automated betweenness clustering approach to refactoring god classes from open source projects. It indicates that betweenness clustering provides an effective mechanism for determining how to partition the members of a god class for an *Extract Class* refactoring. In general, the extracted classes' cohesion scores were considerably better than the original classes' scores.

Chapter 7

Refactoring Using Multiple Clustering Techniques

Previous chapters have discussed the strengths and weaknesses of various clustering techniques relative to refactoring object-oriented classes. This chapter discusses how multiple clustering techniques can be combined, so the strengths of one clustering technique can compensate for the weaknesses of another.

Section 7.1 briefly reviews the only previous research we are aware of that combines multiple clustering techniques – the approach taken by the University of Salerno researchers [BDLMO10b, BDLMO10a]. Section 7.2 discusses our dual clustering technique for the *Extract Class* refactoring. This technique uses betweenness clustering, combined with agglomerative clustering based on "semantics", to create two clusters of class members that form the basis for revised classes. Section 7.2.2 presents a case study that shows how dual clustering works on a large, open source class. Section 7.2.3 discusses *Extract Class* refactoring experiments we did using dual clustering to split large open source classes and the resulting changes in quality metrics. It then compares the results of refactoring based on dual clustering to the results of refactoring based on betweenness refactoring. The chapter concludes with a summary of our contributions.

7.1 Related work

The University of Salerno researchers [BDLMO10b, BDLMO10a] are the only other researchers we know of who have combined multiple clustering techniques to determine how to refactor classes. Their technique uses structural and semantic

information to determine clusters of members that serve as the basis for performing the *Extract Class* refactoring. Section 5.2.3 introduced their "two-step" technique for refactoring large classes and their use of distance functions. This section further evaluates it.

The fundamental data structure underlying their technique is a graph, whose nodes are the class's methods and whose edges are weighted with the similarity scores between the connected nodes. The similarity function that produces the edge weights combines a measure of semantic similarity with measures of structural similarity. They weight the semantic term more highly than the structural terms.

The first step divides the graph by removing edges with low weights. The second step recombines the graph by attaching subgraphs with fewer than three members to larger ones. This step uses average link agglomerative clustering with the same similarity function that was originally used to determine the similarities between the members of the clusters.

While their two-step technique combines a graph-based divisive step with an agglomerative step, it is basically an agglomerative technique. The graph-splitting step breaks a graph whose edges represent similarities. Thus, it is equivalent to separating a minimum spanning tree by removing the edges with the greatest distances (see Section 6.1.1), which is equivalent to single link agglomerative clustering [GR69]. The second step attaches outliers to the main clusters using average link agglomerative clustering.

As a consequence, their technique suffers from the same problems relative to refactoring classes as the agglomerative techniques that were discussed in Section 5.6.1. Because their graphs are not based solely on existing call relationships within classes, the clusters produced by their techniques do not necessarily conserve those calling relationships, i.e., their techniques are not inherently biased towards producing revised classes that have high structural cohesion and low structural coupling. Rather, they produce highly cohesive classes relative to their similarity function, which takes into account both structure and semantics.

7.2 Refactoring god classes using dual clustering

Our initial approach to extracting classes [CAGN09] used betweenness clustering to split intraclass dependency graphs. The clusters produced by this approach retain most of the existing intraclass relationships, so using them as the basis for revised classes tends to produce classes that are structurally cohesive. The results in Section 6.3.2 support this claim.

However, inspection of the dependency graphs of large open source classes revealed that a significant percentage of the graphs were composed of disconnected subgraphs, and many of these subgraphs consisted of three or fewer members (see Section 6.3.2). Betweenness clustering does not assist in determining how those isolated members should be assigned to classes. Lacking any guidance, the implementation based on betweenness clustering kept the class members represented by the small, disconnected subgraphs with the original class.

While the intraclass structural information should be the primary guide for refactoring, there is additional information available that can guide refactoring. For example, a number of researchers have used semantic information embedded in identifier names and comments to measure cohesion [CEJ06, Etz06, DLOV08], to identify topics in source code [Mar04, SDGP10], or to help refactor [BDLMO10b, DLOV08]. Because clustering based on semantics relies on meaningful terms being embedded in the identifiers, they are less reliable than structural information, but can still be useful as a secondary source of information for refactoring.

7.2.1 Dual clustering approach

This section presents an approach that combines two different kinds of clustering algorithms – a graph-based divisive clustering based on structural information, followed by an agglomerative clustering based on semantic information. The first clustering algorithm splits the members of the class into at least two clusters. Afterward, the second clustering algorithm connects any additional small clusters with the two largest clusters. The two resultant clusters form the basis for splitting the original class into two classes, providing they meet certain criteria (to be discussed). The process can be repeated for those cases in which at least one of the revised classes is still large and noncohesive.

Phase 1 – betweenness clustering

The first phase is betweenness clustering (see Section 6.1.2), which operates on an undirected version of the intraclass dependency graph of the class to be refactored. If the initial intraclass dependency graph has two disjoint subgraphs with at least seven members each, then no further divisions of the graph need to be made. Otherwise, we run betweenness clustering on the graphs and stop the clustering once the first new cluster is formed. The two largest clusters serve as seeds for the agglomerative clustering.

Phase 2 – agglomerative clustering using semantics

The second phase is average link agglomerative clustering based on semantics. The initial clusters are the clusters produced by betweenness clustering. The two largest clusters are kept separate and the smaller clusters merge with these.

The semantic distance function used by the agglomerative clustering is the one described in Section 5.3.2, which is useful for clustering documents. Each of a class's methods and attributes is treated as a document, and the document's contents are the stemmed words present in the identifiers, string constants, and non-Javadoc comments. The distance function is based on the cosine similarity of the words present in two given documents.

For each iteration of agglomerative clustering, our algorithm assigns the two largest clusters a maximum distance to prevent them from being merged. All other pairs of clusters are assigned a distance that is the average distance between the members of one cluster and the members of the other. The clustering stops when there are only two clusters remaining. These final two clusters specify the class members that will be present in the refactored classes.

7.2.2 Case study

This section discusses our application of the dual clustering approach to refactoring a noncohesive open source god class, Weka's `RegOptimizer`. `RegOptimizer` is a good example for showing some of the weaknesses of betweenness clustering and agglomerative clustering relative to the *Extract Class* refactoring, and how a combination of those techniques can be useful.

`RegOptimizer` has a large group of members that are connected via calling or accessing relationships, plus eight class members that are not used within

the class – `m_sparseWeights` and `m_sparseIndices` appear to be dead code, `epsilonParameterTipText` and `seedTipText` are used by the user interface via reflection, `serialVersionUID` is used by Java's internal serialization code, `getRevision` and `listOptions` are public methods required by interfaces, and `buildClassifier` appears to be a placeholder method to be overridden by subclasses.

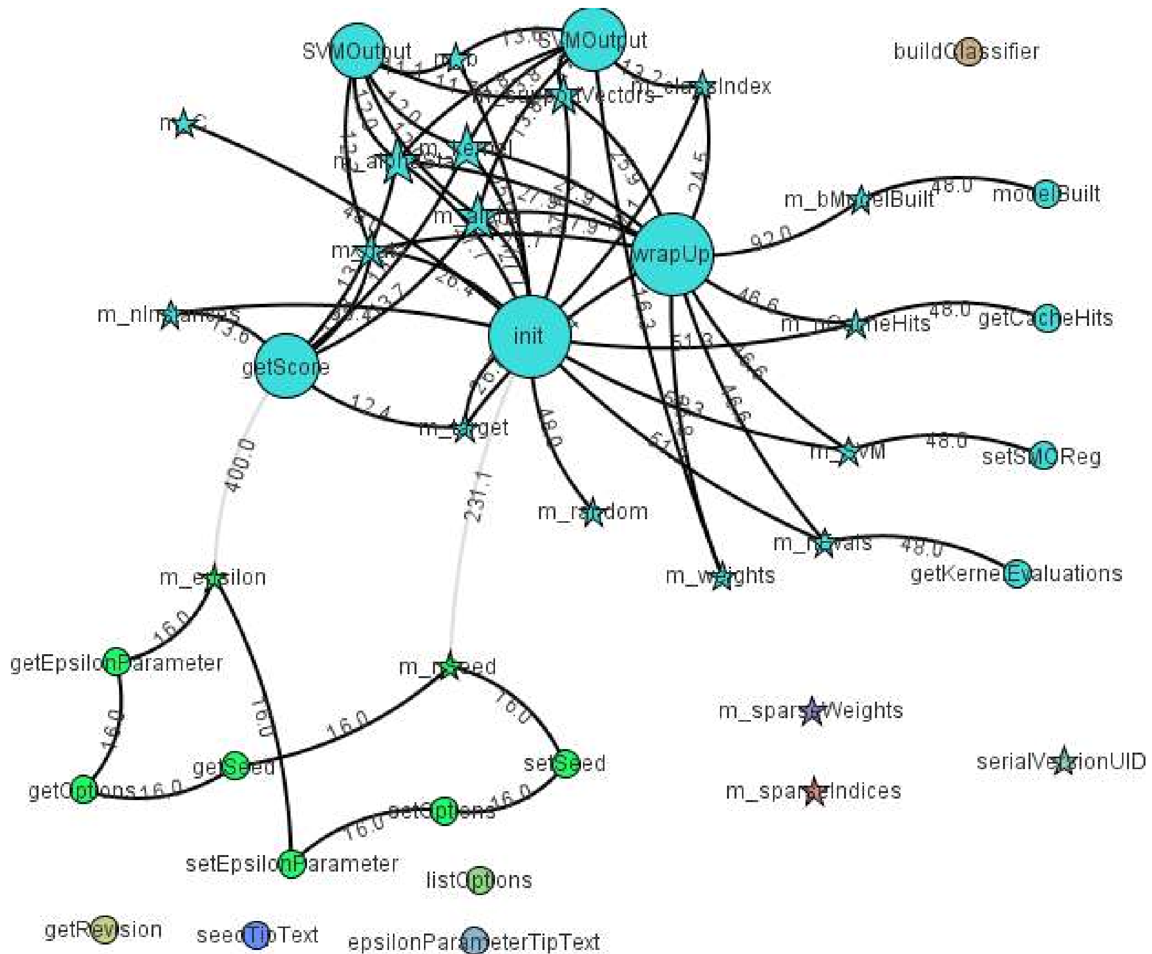


Figure 7.1: RegOptimizer class after betweenness clustering

Figure 7.1 shows the intraclass dependency graph for `RegOptimizer` after betweenness clustering split the large cluster of members into two smaller clusters by removing two edges. Note that the words that comprise the identifiers of the some of nodes in the singleton clusters overlap with some of the words present in the identifiers in the multi-member clusters, e.g., “epsilon”, “seed”, and “options”.

Figure 7.2 shows the dendrogram produced from `RegOptimizer`, using agglomerative clustering based on semantics as described in Section 5.3.2. It is not

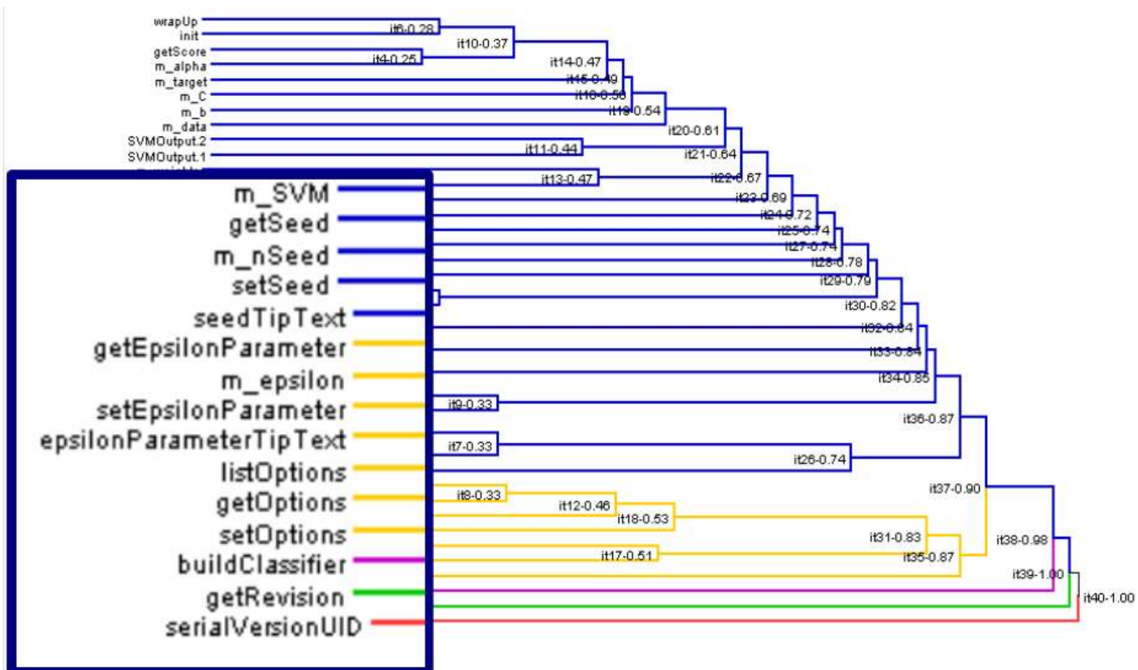


Figure 7.2: RegOptimizer class after agglomerative clustering

clear at what distance this dendrogram produces the most useful clusters; however, if a cut is made at a distance of 0.9, the clusters formed have some similarity with those produced by betweenness clustering. The largest cluster (shown in blue) contains most of the class members; a smaller cluster (shown in yellow) contains seven class members, and there are three leftover singleton clusters. The smaller, yellow cluster consists of four class members with “epsilon” in their names and three with “options” in their names. In contrast with the betweenness clusters, the smaller group does not contain `m_seed`, `getSeed`, or `setSeed`, which are structurally connected to the class members in the yellow cluster, but does contain `listOptions`, and `epsilonParameterTipText`, which are not structurally connected to members in the yellow cluster.

Figure 7.3 shows the result of dual clustering by adding the results of agglomerative clustering to the betweenness clusters. The two largest clusters produced by betweenness clustering are the light blue cluster at the top of the figure and the light green cluster in the bottom left. Red lines indicate the merger of clusters during agglomerative clustering. The lines have boxed labels that indicate the agglomerative clustering iteration in which the clusters were combined. For example, in the first iteration, the singleton `listOptions` was clustered with `epsilonParameterTipText`. This may seem surprising, given that many of the

Figure 7.3: RegOptimizer class after dual clustering

other class members have names that are similar. However, a look at the code in Figure 7.4 shows that, while the two method names are dissimilar, the two method bodies have many tokens in common – “epsilon”, “parameter”, “insensitive”, etc. Indeed, with a slight rewriting of the code, `listOptions` could have used the result of a call to `epsilonParameterTipText`, in which case there would have been a structural connection between the two methods.

The next few clustering iterations can largely be explained based on the words present in the identifiers (although the words in the bodies of the methods are equally significant). In the second agglomerative iteration, the two singleton clusters, `m_sparseIndices` and `m_sparseWeights`, both contain “sparse”. The third iteration combines the `listOptions`, `epsilonParameterTipText` pair with the smaller betweenness cluster, several of whose member names contain “epsilon”, “parameter”, or “options”. The fourth iteration combines clusters, some

```
public String epsilonParameterTipText() {
  return "The epsilon parameter of the epsilon
    insensitive loss function.(default 0.001).";
}

public Enumeration listOptions() {
  Vector result = new Vector();
  result.addElement(new Option(
    "\tThe epsilon parameter in epsilon-insensitive loss
       function.\n"
    + "\t(default 1.0e-3)","L",1,"-L <double>"));

  result.addElement(new Option(
    "\tThe random number seed.\n"
    + "\t(default 1)", "W", 1, "-W <double>"));
  return result.elements();
}
```

Figure 7.4: Source code for epsilonParameterTipText and listOptions

of whose member names contain "weights"; the fifth combines clusters some of whose member names contain "build", and the sixth iteration combines clusters containing "seed", "tip" and "text". The seventh and eight iterations arbitrarily assign outliers that have nothing in common with the other clusters.

The final result of the dual clustering is two clusters. The smaller cluster contains eight structurally connected class members, plus three other members that have similar names to the structurally connected members, plus one extreme outlier (`getRevision`). The larger cluster contains everything else.

7.2.3 Experiments on open source god classes

We ran an experiment to try to determine whether the combination of betweenness clustering and agglomerative clustering provided better results than betweenness clustering alone. The experiment applied our dual clustering technique to the same open source classes that were selected by the god class query in Section 6.3.2. We refactored god classes when the dual clustering produced two clusters containing at least seven members each. Based on the results of the clustering, seven of the thirty classes did not warrant refactoring, because the second largest cluster had too few members.

We refactored the candidate classes using the process described for betweenness clustering in Section 6.3.2, with the dual clustering results serving as inputs to IntelliJ IDEA's *Extract Class* refactoring. Any compilation errors that were present in the refactored classes were corrected by manual editing.

As in the betweenness clustering experiments, Metrics2 provided measurements for the number of fields (NF), number of methods (NM), weighted method count using computational complexity (WMC), and six structural cohesion metrics (LCOM, LCOM*, TCC, DC_D, LCC, and DC_I). ExtC provided the semantic cohesion (C3V) of the classes. Appendix C.4 contains the complete tables of metric values collected for the classes before and after refactoring. Tables 7.1 and 7.2 summarize the average measurements for the 22 refactored classes.[1]

Refactoring based on the results of dual clustering generally had positive effects on cohesion. Refactoring the 22 candidate classes produced 44 revised classes. All of these showed improvement in at least one of the seven cohesion metrics. Only five (11%) showed a decrease in cohesion for more than half of the cohesion metrics.

[1] IDEA's Extract Class failed to operate on one of the classes to be refactored.

Table 7.1: Average metric values (excluding cohesion)

Class	NF	NM	WMC
original	23	52	119
modified	21	51	108
extracted	4	15	27

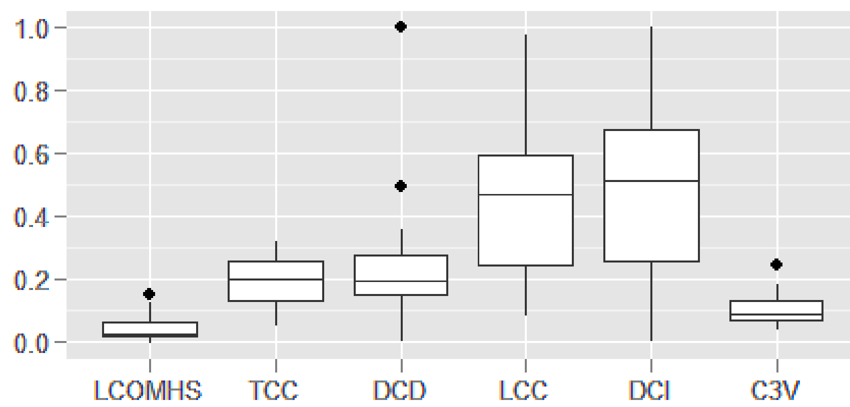
Table 7.2: Cohesion measurements

Class	LCOM	LCOM*	TCC	DCD	LCC	DCI	C3V
Original	1,249	0.96	0.19	0.24	0.43	0.48	0.10
Modified	1,126	0.95	0.23	0.29	0.51	0.56	0.10
Extracted	85	0.61	0.42	0.46	0.54	0.56	0.30

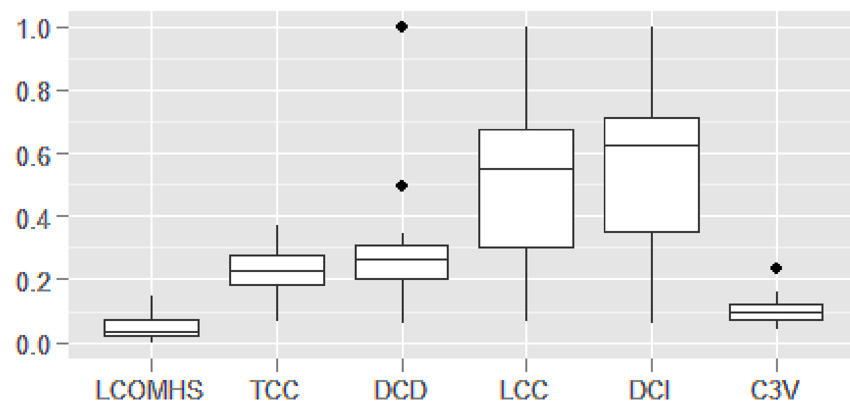
Figure 7.5 uses box plots to summarize the cohesion distributions of the normalized cohesion metrics for the original, modified, and extracted classes. The box boundaries are the values for the first and third quartiles; the lines within each box represent the median. The lines that extend from the boxes reach to the furthest data value that is no more than 1.5 times the size of the box. The outliers beyond this limit are shown as dots. Because LCOM* is a lack of cohesion metric, the values shown in the figures for "LCOMHS" is (1 - LCOM*). This makes it easier to compare the cohesion metrics – better cohesion scores will always be towards the top.

As with the results of refactoring based on betweenness clustering discussed in Section 6.3.2, there is not much difference between the metric values of the original classes (Figure 7.5(a)) and the modified classes (Figure 7.5(b)), although there is generally slight improvement. There is more improvement shown in the extracted classes (Figure 7.5(c)).

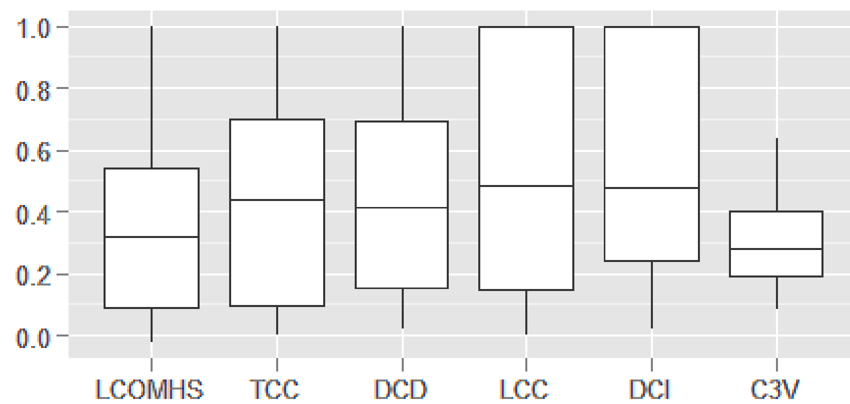
Figure 7.6 compares the average cohesion values for the classes that were refactored based on the results of betweenness clustering to the average cohesion values for the classes refactored based on the results of dual clustering. The average cohesion values for the original god classes to be refactored based on betweenness clustering (betw-orig) and dual clustering (dual-orig) are nearly the same. They



(a) Original class



(b) Modified class



(c) Extracted class

Figure 7.5: Cohesion distributions

are not identical, because dual clustering produces 23 candidates for refactoring (based on clusters of seven or more members) whereas betweenness clustering produces only 18 candidates. The average cohesion values for the modified classes

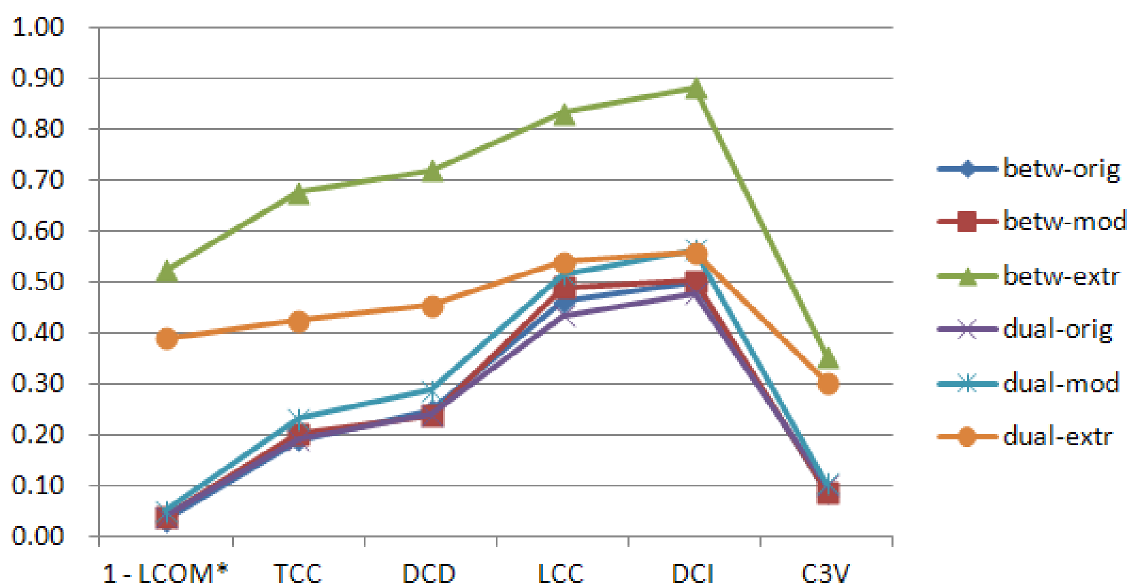


Figure 7.6: Average cohesion values

(betw-mod and dual-mod) are not much different, with the cohesion values for the classes that were modified based on dual clustering being slightly better. On the other hand, the cohesion values for the classes extracted based on betweenness clustering (betw-extr) are much better than those produced by dual clustering (dual-extr).

The differences are explained by the redistribution of the many small, structurally noncohesive clusters, many consisting of a single class member. Refactoring based on betweenness clustering puts all of the small clusters into the modified class, so the modified class may have many structurally disconnected parts, whereas the extracted class is structurally connected.

In contrast, refactoring based on dual clustering distributes the class members from small, noncohesive clusters to both the modified and extracted classes. With dual clustering, the refactoring forms the extracted classes based on the smaller of the two largest clusters produced by betweenness clustering, together with some of the small, structurally disconnected clusters. The presence of structurally disconnected, small groups of members in the extracted classes has a relatively large effect on lowering the cohesion scores relative to the classes produced based on the results of betweenness clustering alone.

Refactoring based on dual clustering forms the modified classes based on the larger of the two largest clusters produced by betweenness clustering, together

with some of the small, structurally disconnected clusters. While there are fewer structurally disconnected parts of the extracted class relative to the ones created by betweenness clustering, the increase in cohesion is less pronounced than the decrease in cohesion of the extracted classes, because the modified class is larger than the extracted class.

7.3 Evaluation of refactoring using dual clustering

While the ideas behind dual clustering are intuitively attractive, it is unclear whether the dual clustering approach to extracting classes is any better than the betweenness clustering approach alone. Based on the criterion that the extracted class should contain at least seven class members, dual clustering enabled the refactoring of more classes than betweenness clustering (22 vs. 18) in our experiments. On the other hand, based on the average cohesion scores of the modified and extracted classes, betweenness clustering generally produces more cohesive classes than does dual clustering.

7.4 Contribution summary

Previous chapters have discussed several approaches to refactoring object-oriented classes using various clustering techniques and the limitations of these techniques. This chapter discussed our dual clustering technique for refactoring god classes, which goes beyond previous attempts by using two distinct, complementary clustering phases. Dual clustering identifies structurally cohesive core members for new classes using betweenness clustering and then uses agglomerative clustering based on semantic information to add outliers to the core members.

To test the hypothesis that our clustering techniques offer useful advice for splitting classes, we ran the algorithm on 30 open source classes and collected data both on the original classes and on the refactored classes. Refactoring based on dual clustering enabled us to refactor 22% more classes than when we refactored based on betweenness clustering alone, although there was more improvement in the cohesion values with betweenness clustering.

Chapter 8

Conclusions and Future Work

Many of the refactorings from Fowler's book [FBB+99] pertain to rearranging the members of object-oriented classes such that the members that most belong together co-exist in the same class. One of the challenges to programmers attempting to refactor classes is deciding which members belong together.

Clustering techniques form clusters from entities that belong together, and researchers have been examining their use for better modularizing object-oriented systems for over twenty years [MU90, Lak97, AFL99, KE00, MM06, Wig97]. We have applied clustering techniques to refactoring large classes in new ways [CAGN09, CAG11]. Concurrent with our research, other researchers [SC08, BDLO11, FTCS09] have applied different clustering techniques to refactoring. While our clustering techniques have some advantages over competing techniques for determining how to refactor certain kinds of problematic classes, the main contribution of this thesis is an analysis of the strengths and weaknesses of various clustering techniques as applied to refactoring classes.

Section 8.1 reviews our research contributions in a narrative format, emphasizing the issues as they arose, and how we addressed them. Section 8.2 gives a concise summary of the contributions. Section 8.3 discusses potential future work.

8.1 Contribution review

Critical to any clustering activity is the choice of entities to cluster. Among the researchers who have applied clustering techniques to refactoring object-oriented classes, there is general agreement that the entities to be clustered are the methods and attributes of the classes [SC08, BDLO11, CAG11, FTCS09], although some

researchers [CS06, CC08, SC08] also include the classes themselves.

The next requirement for effective clustering is to determine which characteristics of the entities determine whether they belong together. Consistent with much of the cohesion research (see Section 2.1.3), class refactoring research has concentrated on the structural interactions between methods and attributes, and the semantic content of code and comments. Some information used by a smaller number of cohesion metrics (e.g., method arguments and calling relationships from clients) has been ignored by the cluster-based refactoring researchers, but is worth investigating in the future.

Of the researchers who have investigated the use of clustering for determining how to refactor object-oriented classes, most have used agglomerative clustering in combination with a Jaccard distance function that compares local structural information encoded as property sets. The particular structural information used varies between the approaches. Section 5.3.1 evaluated the effectiveness of these approaches for determining how to refactor two simple classes in our test suite, `PersonCarDisjoint` and `AnonymousPersistence`, using various property sets. None of them, except for one created by us, and one used by Simon and Fokaefs et al. [SSL01], were able to generate the preferred clusters. Our property set used the entire local structural neighborhood of a class member (itself and all directly calling and directly called class members), whereas theirs omitted the calling methods from a method's property set. They do not explain this omission. We recommend using as much structural information as possible when clustering to decide how to refactor.

Semantic information has also been used as a basis for determining how to refactor classes. Some researchers [AFL99] feel that, in general, semantic information gives better results than structural information when creating packages; however, they acknowledge that there are cases where code lacks much semantic information, in which case, clustering based on semantics is not useful. Section 5.3.2 discussed some experiments on clustering using semantic information for `PersonCarDisjoint` and `AnonymousPersistence`, both of which contain meaningful identifiers. The clustering produced acceptable clusters only for `PersonCarDisjoint`. While clustering based on the word usage in identifiers and comments is informative, it can also be unreliable, so we recommend using structural information as the primary basis for determining how to reorganize classes, with semantic information being secondary.

The conclusions above were based on agglomerative clustering. One of the

important decisions when using agglomerative clustering is when to combine clusters. Section 5.3.1 discusses why we recommend average link or single link clustering over complete link.

Not all distance-based approaches for determining clusters of class members have been agglomerative. Some researchers [CS06] have used k-means, a partitional clustering technique. Agglomerative clustering, k-means, and k-medoids all use distance functions, but in different ways. Agglomerative clustering determines whether two clusters belong together based on a distance function, and typically connects together clusters that are "nearest neighbors" according to the linkage criterion. K-means and k-medoids generally rely on multi-dimensional spatial distance functions, and instead of clustering based on nearest neighbors, they cluster based on the nearness of an entity to some central points in space (k-means) or to prototypical members (k-medoids). Consequently, k-means and k-medoids tend to produce compact clusters of approximately the same diameter, which is sometimes undesirable. Section 5.2.2 discusses another problem with using k-means or k-medoids for refactoring object-oriented programs – the difficulty in determining what constitutes the dimensions of space for object-oriented software. Furthermore, partitional approaches give no indications of which parts of the cluster are more highly related than others. This can be useful information to a programmer who is trying to refactor, but is not satisfied with the highest level clusters. Based on the limitations of partitional approaches, we discourage their use for determining how to refactor classes.

While there has been some success with distance-based clustering techniques, they have the following drawbacks as a basis for refactoring classes:

- Distance functions are opaque, making it hard to debug the clustering process.
- It is hard to compose a good distance function that incorporates non-local structural information.
- Distance-based clustering does not inherently maintain existing relationships between class members. This makes refactoring more difficult and tends to introduce more change than graph-based clustering techniques do.

To help with the problem of distance function opacity, we built a novel visualization that shows the steps of agglomerative clustering superimposed over the structure of an intraclass dependency graph. Section 5.5 described this visualization and how it helps the researcher.

Graph-based divisive clustering techniques remedy some of the problems with the distance-based approaches. Graphs more easily capture the idea of connectedness between class components than do distance functions based on property sets. (The property sets of Section 5.2.4 are all based on the local connections of a class member.) Also, graph-based divisive clustering techniques tend to produce clusters that maintain existing relationships between class members, so refactoring based on those clusters tends to produce structurally cohesive clusters with little coupling between them. Section 6.3 describes our use of betweenness clustering to refactor god classes, which appears to be the first use of graph-based clustering using primarily structural information as the basis for refactoring classes.

Our experiments showed that betweenness clustering provides a basis to successfully refactor some god classes, but not others. To help understand why, we created the visualization described in Section 6.3.1. That visualization helped reveal "noisy" nodes and edges in the intraclass dependency graphs that affected the quality of the clustering. After filtering these noisy nodes and edges, the clustering results improved.

The same class members that can cause problems for betweenness clustering can also cause problems for cohesion metrics and other clustering techniques. Section 2.4.1 described our novel approach for restructuring intraclass dependency graphs to help ameliorate the effects of these special nodes. This graph restructuring technique has also been incorporated into ExtC's clustering tools.

Graph-based divisive clustering addresses some of the problems with agglomerative clustering, but has its own problems. For example, half of the large classes discussed in Section 6.3.2 had seven or more disconnected subgraphs with fewer than seven members. Graph-based clustering techniques can not provide guidance on placement of these seemingly isolated class members. However, in some of those cases, there are structural relationships that are not readily apparent, e.g., those induced via reflection, that may be addressable using other techniques. Section 7.2 described how we augment betweenness clustering based on structure with agglomerative clustering based on semantics to partially address the problem of structurally disconnected subgraphs.

8.2 Contribution summary

The primary contribution of this research is the identification of the strengths and weaknesses of various clustering techniques for identifying groups of class members that can be used as the basis for refactoring object-oriented classes. As part of this work, we introduced two new applications of clustering for identifying what belongs in the refactored classes:

1. Betweenness clustering – Clustering based on the analysis of communication patterns within a class (Section 6.3).
2. Dual clustering – An initial clustering step based on intraclass communication forms the major clusters, and a subsequent step based on semantics handles outliers (Section 7.2).

Both of these techniques produce clusters that serve as the basis for *Extract Class* refactorings. Additional contributions include:

- An analysis of cohesion metrics and their limitations (Section 2.3).
- The development of a technique that can make certain structural cohesion metrics more accurate. This technique eliminates specified "special" class members and relationships from consideration and includes additional relationships in the calculations (Section 2.4).
- The use of the above technique for restructuring inputs to clustering, e.g., the filtering of certain "noisy" entities and relationships within classes (Section 5.7).
- The creation of the open-source ExtC Eclipse plug-in (Chapter 4). ExtC provides an environment for exploring the use of clustering for refactoring, including interactive visualizations of clusters being formed via agglomerative clustering and betweenness clustering (Sections 5.5 and 6.3.1).
- The creation of a test suite to use for evaluating cohesion metrics and for evaluating the effectiveness of the *Extract Class* refactoring (Section 3.2.2).

8.3 Future work

The clustering techniques described in this thesis are helpful for quickly determining how to refactor some poor quality classes; however, there are other faulty classes that these techniques do not handle well. This section discusses some

potential future work that may assist in the detection and correction of these problematic classes.

8.3.1 Cohesion metrics – comparative study and sensitivity analysis

Despite the amount of research that has been done, making good use of object-oriented cohesion metrics is difficult. There are many different cohesion metrics, and many of them do not correlate well [DJ03, MP05, BT07]. It is also often hard to determine how to best use the metrics for a given purpose, e.g., detecting classes in need of refactoring.

It would be useful to perform an empirical study, similar to one performed by Barker and Tempero [BT07], that analyzes many cohesion metrics. Barker and Tempero's study showed significant differences between the results of different families of cohesion metrics. An updated study should analyze the major sources of differences between the metrics. The study should also determine how sensitive the metrics are to the sources of illusory and hidden cohesion that were discussed in Section 2.3. Some of these seemingly minor variations in class structure can have large effects on cohesion scores. For example, Section 2.4.3 discussed how two classes that were designed to be noncohesive had LCC values of 0.47 and 1.0 where the only difference between them was the presence of a `toString` method in the maximally cohesive class, that connected two parts of the intraclass dependency graph that were disconnected in the less cohesive class. Such a study would provide software engineers the ability to more effectively use existing cohesion metrics and give researchers insights into how to develop more robust metrics.

8.3.2 Handling inheritance

The techniques discussed in this thesis have concentrated on refactoring classes that do not inherit from anything besides `Object`; however, the presence of additional inheritance makes software development, including refactoring, more complicated [MS98, KS08]. Changes to classes that involve inheritance may need to be propagated up and down the inheritance hierarchy. There has been some research on moving class members up or down the inheritance hierarchy [ST98, SS04, MHVG08]; however, we know of no refactoring research about extracting or moving functionality outside of the hierarchy. For example, examining the internal

structure of a class may suggest that it be split into two classes. However, such a split might break subclasses, unless the subclasses can be split in a similar fashion. There are many such issues, involving superclasses, subclasses, and implemented interfaces.

Inheritance also complicates the use of clustering techniques. For example, a superclass may call a placeholder method, where all of the connectivity and behavior for that method is in the subclass, or, a subclass may have a method that is seemingly disconnected, but actually calls superclass members. In both of these cases, clustering based merely on the members defined within a single class is likely to give misleading results. However, it is not clear how members of related classes in the class hierarchy might best be combined to allow usage of most traditional clustering algorithms. Because inheritance plays such a large part in object-oriented languages, these issues need to be investigated further.

8.3.3 Additional applications of clustering algorithms

There are many clustering algorithms that are potentially applicable for helping to determine how to refactor object-oriented classes. This section discusses two of them.

Max flow/min cut based refactoring

Although the University of Salerno researchers have used max flow/min cut to separate a graph where the edges are weighted with similarity scores, we are not aware of anybody who has used max flow/min cut on a purely structural graph to determine how to refactor object-oriented classes. Max flow/min cut is a good cognitive match for the problem of determining how to redistribute a set of class members into two classes using either the *Move Method*, *Move Field*, or *Extract Class* refactorings. The structural graph represents the flow of information (e.g., via method parameters and return values) and the minimum cut represents a likely class boundary.

When the intent of clustering is to identify the members involved in *Move Method* or *Move Field* refactorings, the graph can be comprised of the intraclass dependency graphs of the two examined classes conjoined with additional edges representing the calls from the members of one class to the members of the other. The source node would come from one class and the destination node from the

other. The source and destination nodes should each be "core" members of the respective classes.

It is not yet clear how to best determine the core class members. In cases where there is not a knowledgeable programmer available to make the choice, it is possible to use any of a number of graph-based algorithms to identify candidate source and destination nodes based on their positions in the graph [New10, Kle99]. For example, a heavily connected node or centrally located node within a class's graph might be a good candidate for the source or destination. This task requires care. For example, a logger field or a debug method might be the most heavily used member of a class, but they might make poor choices as the basis for determining how to split functionality between classes.

Max flow/min cut may also be applicable to splitting an intraclass dependency graph to determine an *Extract Class* refactoring; however, it is harder to determine how to choose source and destination nodes when attempting to perform *Extract Class*. The two nodes from a given class with the highest score on some importance metric may well be closely connected with each other, in which case they probably should not be separated. One alternative for source and destination determination is to use "semantics" to determine which are the two main topics (i.e., core semantic clusters) present in a class [SDGP10, LPF+09], and use one class member from each of these topics as a source and destination.

Frequent pattern analysis

Many of the clustering techniques discussed in this thesis were based on analyzing the dependencies within one or more classes that may be in need of refactoring. A different strategy is to determine attributes and methods within classes that tend to be used together by external clients. For refactoring, these class members that are commonly used by external classes may indicate members that might best belong together in an extracted class, or they may indicate the potential for an interface. As an example, consider the `PersonCarDirect` class. If there were 100 client classes that all used just the "person methods" and a different 100 classes that all used just the "car methods", that would seem a good indication to split `PersonCarDirect` into `Person` and `Car` classes, or to extract `Person` and `Car` interfaces.

The basic idea is to find a significant subset of clients who commonly use a specific subset of a given class's methods. The idea is analogous to how retail

businesses employ *market basket analysis* to find sets of sales items that commonly occur together in customers' purchases. In this analogy, a "purchase" corresponds to the collection of method calls made by a particular client, and each sales item corresponds to a method call.

There are many algorithms developed by the *frequent pattern mining* [HCXY07], *subspace clustering* [KKZ09, PHL04], and *biclustering* [BPP08] communities that are applicable to this task. As far as we know, nobody in the refactoring community is using these algorithms as the basis for refactoring classes. We have done a small amount of exploratory research that makes use of one such algorithm, FPGrowth [HPY00], to examine client usage of a class's members. This research has raised a number of questions. For example, what degree of common usage of a class's members indicates that the members constitute a potentially desirable new class? It seems easy to say that if 100 client classes all use exactly the same eight "person methods" and a different 100 classes all use exactly the same eight "car methods", then two new classes or interfaces should be created, one with the eight person methods, and one with the eight car methods. The situation becomes muddled when the client classes use different subsets of methods. Some client classes may just use the `get-` accessors of the person attributes, while other classes may just use the `set-` accessors of the person attributes. Other clients may use different subsets of the `get-` accessors and `set-` accessors, etc. There needs to be some way of reconciling slight differences in client usage of methods, perhaps by combining client usage patterns with structural or semantic information.

Another question to be resolved is how to determine when clusters are worthy of being the basis for interfaces. The design criteria for what constitutes a meaningful and useful interface specification are even less clear than the rules for what constitutes a good class [SM07]. Some advocates of interface-based programming argue for having many interfaces, each of which contains the minimal functionality required for a given client [Ste07]. On the other hand, Steimann [Ste07] points out that the Java libraries do not themselves make heavy use of interfaces, e.g., there was no interface corresponding to its heavily used `String` class until Java 1.4. It would make an interesting study to see how well the client usage of popular libraries, e.g., those provided by Java, corresponds to the interfaces provided by those libraries.

8.3.4 Adding domain knowledge

For almost any computation, the quality of the output is dependent on the quality of the input. Most of the techniques that have been discussed in this thesis have used a relatively small amount of the information available to a programmer to determine how classes might be refactored. This section discusses additional information that might be used and how it can be represented.

Semantics

The "semantic" similarity measures discussed in Section 5.3.2 are statistical techniques based on similar usage of words in identifiers and within comments, and on patterns of co-occurrence of words in the corpus. One of the advantages of the statistical techniques is that they are inexpensive to set up, relative to the cost of manually constructing a knowledge base. On the other hand, the statistical measures provide only a weak approximation of true semantic similarity, especially when the corpus from which the words are extracted is small, as when the corpus is based on the code of a small number of projects.

The usefulness of statistical techniques (like LSI [DDF+90]) can be increased through the use of a larger or a domain-specific corpus. For example, the Google similarity distance [CV07] computes "semantic" distances using a non-specialized corpus, i.e., all of the documents that the underlying search engine has indexed. For software applications, a better refinement might be to use a more specialized software corpus, e.g., one built from the Qualitas Corpus [TAD+10]. Then, relative word frequencies of software terms would come into play. Words like "exception", "stack", "factory", etc. occur with a different frequency in code than they do in general speech, and they typically have more specific meanings.

An alternative approach to using statistical approximations of meaning is to use explicit representation of meaning, e.g., knowledge bases or ontologies, that can be either general-purpose or application domain specific. Building knowledge bases by hand is expensive; however, it should be possible to use existing knowledge bases, or to programmatically construct one. There are some existing general-purpose knowledge-bases, e.g., Cyc [Len95] and WordNet [BH06, Mil95], to choose from, as well as several repositories for domain-specific ontologies [SAR+07, oM08]. The cost of integration for these knowledge sources will vary.

It is possible to automate some aspects of the construction of a software-specific

knowledge base. For example, a Java library has built-in information about how classes are inter-related through inheritance, type information, etc., and the code analysis capabilities provided by many IDEs are capable of providing knowledge about how these code entities inter-relate. When one method uses a `Vector` and another method uses a `List`, statistically-based semantic techniques may or may not note similarity, depending on the word patterns in the underlying corpus. On the other hand, it can easily be determined programmatically that `Vector` and `List` are related, because `Vector` implements `List`. One of the research issues to be resolved is how to numerically represent this similarity, and how these similarities might be integrated with scores produced by other functions, e.g., the scores produced by statistical techniques.

Acknowledging differences in class roles

There is more to the software engineering of object-oriented software than the idealized development of some prototypical class. Often, software engineers are encouraged to build classes of a certain size, with certain complexity, etc.; however, not all classes fulfill the same roles. Classes in design patterns [GHJV94, Ker05, OCS10] have particular roles, and depending on the roles, the characteristics of the classes may differ greatly. For example, classes involved in a Model-View-Controller (MVC) pattern may seem to exhibit feature envy when the view classes access data in the model; however, this is by design. The MVC pattern explicitly separates data members from the UI code that uses them.

Bad smell detectors need to be improved to acknowledge the various roles a class can play. Similarly, a system that recommends refactorings will be more useful if it has knowledge about how the classes should be restructured relative to the role of the classes. There is some work already underway; for example, Marinescu [Mar06] takes enterprise application context into account when determining whether a class emits the data class or feature envy smells. There are many more smells and application contexts that have not been analyzed, and as far as we know, there is no overarching theory about how to handle role-specific refactoring.

Adding knowledge to graphs

While the clustering algorithms previously described have sufficient information to generate good recommendations for refactoring some classes, the use of additional

information about the classes and their members should enable more effective recommendations. The graphs used in betweenness clustering (Section 6.1.2) have nodes that represent methods or attributes, and undirected, unweighted links that represent calling or accessing relationships. In addition, the basic graph representation can be restructured by removing specified nodes and condensing others, as discussed in Section 2.4.1.

There is additional software information that can be incorporated in the graphs and used by graph-based clustering techniques. For example, we have coarse-grained boolean control of the class members and their relationships – they are either considered in the clustering, or they are not. There is no weighting of nodes or edges based on software knowledge. Some enhancements to the representation might include:

- Weighting the importance of nodes and edges based on characteristics of the underlying code, e.g., modifying node weights based on their number of lines of code.
- Distinguishing relationships between entities, e.g., inheritance, uses, implements, reads, writes, etc.

As more information gets added, the underlying graph representation becomes less like simple, generic graphs upon which generic algorithms (like betweenness clustering) work, and becomes more like *semantic networks* [BS91] upon which more specialized knowledge-based algorithms work.

The use of semantic networks and knowledge-based techniques may produce more accurate results, but these techniques incur costs associated with the development of the knowledge base and associated algorithms. The relative usefulness and cost of semantic techniques will need to be evaluated.

8.4 Conclusions

The high level goal of this research is to decrease the overall maintenance cost of software by facilitating refactoring of object-oriented classes. This thesis discusses experiments that refactored classes based on suggestions provided by clustering techniques, where the refactored classes generally showed improvements in several object-oriented software metrics associated with maintainability.

The clustering techniques described in this thesis provide a relatively low cost refactoring aid that provide immediate benefit to a programmer. The clustering

tools are available now as an open source Eclipse plug-in, and many of the algorithms can be run in seconds to minutes. However, while the tools are useful to programmers performing maintenance, these are auxiliary benefits.

The main contribution of the research was the analysis of clustering techniques, and their application to refactoring object-oriented classes. This analysis included the determination of the usefulness of the information from which the clusters are derived (e.g., structural information and "semantic" information), as well as the determination of the characteristics of the clustering algorithms that made them more or less suited to generating useful refactoring suggestions.

There are many ways this research can be extended. Some of the potential extensions described above may provide better guidance for people interested in refactoring. Future research may determine whether the potential added benefit of these proposed enhancements is worth the added cost.

Appendices

Appendix A

Test Suite Source Code

The cohesion-tests [CAGN10] (`http://code.google.com/p/cohesion-tests/`) project provides a collection of Java test classes for use in evaluating object-oriented cohesion metrics. The classes vary in the amounts of connectivity within the classes' members, and in the types of connectivity. These classes are also useful in evaluating the effectiveness of automated *Extract Class* refactoring tools. The code for the classes referenced in this thesis is included below.

A.1 AnonymousPersistence

```
package nz.ac.vuw.ecs.kcassell.testclasses.anonymous;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
```

```
 * This class has no fields. It has some methods that store
    and retrieve objects
 * from files and some that store and retrieve from
   databases.
 * @author Keith Cassell
 */
public class AnonymousPersistence {

  public void saveSerializableToFile(Object obj, String
     file) {
    FileOutputStream foStream = null;
    ObjectOutputStream ooStream = null;
    try {
      foStream = new FileOutputStream(file);
      ooStream = new ObjectOutputStream(foStream);
      ooStream.writeObject(obj);
      ooStream.close();
    } catch (IOException e) {
      handleSerializationException("Unable to write to " +
         file, e);
    }
  }

  public static Object readSerializableFromFile(String file
    ) {
    Object obj = null;
    FileInputStream fis = null;
    ObjectInputStream ois = null;
    try {
      fis = new FileInputStream(file);
      ois = new ObjectInputStream(fis);
      obj = (Object) ois.readObject();
      ois.close();
    } catch (IOException e) {
      handleSerializationException("Unable to read from " +
          file, e);
```

```
    } catch (ClassNotFoundException e) {
      handleSerializationException("", e);
    }
    return obj;
  }

  protected static void handleSerializationException(String
      msg, Exception e) {
    System.err.println(msg + ": " + e);
    e.printStackTrace();
  }

  public Object readFromDB() {
    Object obj = null;
    String url = "org.somewhere.Driver";
    loadDriver(url);
    Connection connection = null;
    Statement statement = null;
    ResultSet resultSet = null;
    try {
      connection = DriverManager.getConnection(url);
      statement = connection.createStatement();
      resultSet = getDatabaseValues(statement);
      // TODO reconstruction via reflection
      resultSet.close();
    } catch (SQLException sqle) {
      handleSQLException(sqle);
    } finally {
      releaseResources(connection, statement, resultSet);
    }
    return obj;
  }

  public void saveToDB(Object obj) {
    String url = "org.apache.derby.jdbc.ClientDriver";
    loadDriver(url);
```

```
  Connection connection = null;
  PreparedStatement statement = null;
  try {
    connection = DriverManager.getConnection(url);
    // TODO populate tables using values obtained from
       reflection
    String sqlString = "INSERT CLASS_TABLE VALUES (?, ?)"
       ;
    statement = connection.prepareStatement(sqlString);
    saveDatabaseValues(obj, statement);
  } catch (SQLException sqle) {
    handleSQLException(sqle);
  } finally {
    releaseResources(connection, statement, null);
  }
}

public void loadDriver(String driver) {
  try {
    Class.forName(driver).newInstance();
  } catch (Exception e) {
    System.err.println("Unable to load the JDBC driver "
       + driver);
  }
}

private ResultSet getDatabaseValues(Statement statement)
  throws SQLException {
  ResultSet resultSet;
  String sqlString = "SELECT * FROM CLASS_TABLE";
  resultSet = statement.executeQuery(sqlString);
  return resultSet;
}

private void saveDatabaseValues(Object obj,
   PreparedStatement statement)
```

```
    throws SQLException {
    statement.setString(1, obj.getClass().getSimpleName());
    statement.setString(2, obj.getClass().getPackage().
       getName());
    statement.executeUpdate();
  }

  public static void handleSQLException(SQLException e) {
    while (e != null) {
      System.err.println("\n——— SQLException: " + e.
         getMessage());
      System.err.println("  Error Code: " + e.getErrorCode
         ());
      e.printStackTrace(System.err);
      e = e.getNextException();
    }
  }

  public void releaseResources(Connection conn, Statement
     statement,
                ResultSet result) {
    try {
      if (result != null) result.close();
      if (statement != null) statement.close();
      if (conn != null) conn.close();
    } catch (SQLException sqle) {
      handleSQLException(sqle);
    }
  }

}
```

A.2 PersonCarDisjoint

```
package nz.ac.vuw.ecs.kcassell.testclasses.persons;

/**
 * This class has characteristics of both a Person and a
   Car. No method accesses
 * both Person and Car fields. All methods access fields
   directly.
 * @author Keith Cassell
 */

public class PersonCarDisjoint {
  // Person fields
  private String firstName;
  private String lastName;
  private int id;

  // Car fields
  private String make;
  private String model;

  /** The vehicle id number */
  private int vin;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }
```

```
public void setLastName(String surname) {
  this.lastName = surname;
}

public int getId() {
  return id;
}

public void setId(int id) {
  this.id = id;
}

public String getMake() {
  return make;
}

public void setMake(String make) {
  this.make = make;
}

public String getModel() {
  return model;
}

public void setModel(String model) {
  this.model = model;
}

public int getVin() {
  return vin;
}

public void setVin(int vin) {
  this.vin = vin;
}
```

```
// //////////

  public String getPersonInfo() {
    return "" + id + ": " + firstName + " " + lastName;
  }

  public String getPersonName() {
    return firstName + " " + lastName;
  }

  public String getCarInfo() {
    return "" + vin + ": " + make + " " + model;
  }

  public String getCarDescription() {
    return make + " " + model;
  }

}
```

A.3 PersonCarDirect

```
package nz.ac.vuw.ecs.kcassell.testclasses.persons;

/**
 * This class has characteristics of both a Person and a
   Car.
 * ToString is the only method that accesses both Person
   and Car fields.
 *  It accesses them indirectly through other methods.
 * All other methods access fields directly.
 * @author Keith Cassell
 */

public class PersonCarDirect {
  // Person fields
  private String firstName;
  private String lastName;
  private int id;

  // Car fields
  private String make;
  private String model;

  /** The vehicle id number */
  private int vin;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
```

```
    return lastName;
  }

  public void setLastName(String surname) {
    this.lastName = surname;
  }

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getMake() {
    return make;
  }

  public void setMake(String make) {
    this.make = make;
  }

  public String getModel() {
    return model;
  }

  public void setModel(String model) {
    this.model = model;
  }

  public int getVin() {
    return vin;
  }

  public void setVin(int vin) {
```

```
    this.vin = vin;
  }
/////////////

  public String getPersonInfo() {
    return "" + id + ": " + firstName + " " + lastName;
  }

  public String getPersonName() {
    return firstName + " " + lastName;
  }

  public String getCarInfo() {
    return "" + vin + ": " + make + " " + model;
  }

  public String getCarDescription() {
    return make + " " + model;
  }

  public String toString() {
    return getPersonInfo() + " owns a " + getCarInfo();
  }

}
```

A.4 PersonCarIndirect

```
package nz.ac.vuw.ecs.kcassell.testclasses.persons;

/**
 * This class has characteristics of both a Person and a
   Car. ToString is the
 * only method that accesses both Person and Car methods.
   Only the accessors
 * access fields directly. The other methods access them
   via the accessors.
 *
 * @author Keith
 */

public class PersonCarIndirect {
  // Person fields
  private String firstName;
  private String lastName;
  private int id;

  // Car fields
  private String make;
  private String model;

  /** The vehicle id number */
  private int vin;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
```

```
  public String getLastName() {
    return lastName;
  }

  public void setLastName(String surname) {
    this.lastName = surname;
  }

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getMake() {
    return make;
  }

  public void setMake(String make) {
    this.make = make;
  }

  public String getModel() {
    return model;
  }

  public void setModel(String model) {
    this.model = model;
  }

  public int getVin() {
    return vin;
  }
```

```
  public void setVin(int vin) {
    this.vin = vin;
  }

  public String getPersonInfo() {
    return "" + getId() + ": " + getPersonName();
  }

  public String getPersonName() {
    return getFirstName() + " " + getLastName();
  }

  public String getCarInfo() {
    return "" + getVin() + ": " + getCarDescription();
  }

  public String getCarDescription() {
    return getMake() + " " + getModel();
  }

  public String toString() {
    return getPersonInfo() + " owns a " + getCarInfo();
  }

}
```

A.5 PersonCarSpecial

```
package nz.ac.vuw.ecs.kcassell.testclasses.persons;

import java.util.logging.Logger;

/**
 * This class has characteristics of both a Person and a
    Car.
 * ToString, hashCode, equals, and a constructor access
 *  both Person and Car fields.
 * Most methods access fields directly.
 * @author Keith Cassell
 */

public class PersonCarSpecial {
  // Person fields
  private String firstName;
  private String lastName;
  private int id;

  // Car fields
  private String make;
  private String model;

  /** The vehicle id number */
  private int vin;
////////////

  private final Logger logger =
    Logger.getLogger(getClass().getSimpleName());

  public String getFirstName() {
    logger.entering("PersonCarSpecial", "getFirstName");
    return firstName;
  }
```

```
public void setFirstName(String firstName) {
  logger.entering("PersonCarSpecial", "setFirstName");
  this.firstName = firstName;
}

public String getLastName() {
  logger.entering("PersonCarSpecial", "getLastName");
  return lastName;
}

public void setLastName(String surname) {
  logger.entering("PersonCarSpecial", "setLastName");
  this.lastName = surname;
}

public int getId() {
  logger.entering("PersonCarSpecial", "getId");
  return id;
}

public void setId(int id) {
  logger.entering("PersonCarSpecial", "setId");
  this.id = id;
}

public String getMake() {
  logger.entering("PersonCarSpecial", "getMake");
  return make;
}

public void setMake(String make) {
  logger.entering("PersonCarSpecial", "setMake");
  this.make = make;
}
```

```
public String getModel() {
  logger.entering("PersonCarSpecial", "getModel");
  return model;
}

public void setModel(String model) {
  logger.entering("PersonCarSpecial", "setModel");
  this.model = model;
}

public int getVin() {
  logger.entering("PersonCarSpecial", "getVin");
  return vin;
}

public void setVin(int vin) {
  logger.entering("PersonCarSpecial", "setVin");
  this.vin = vin;
}

public String getPersonInfo() {
  return "" + id + ": " + firstName + " " + lastName;
}

public String getPersonName() {
  return firstName + " " + lastName;
}

public String getCarInfo() {
  return "" + vin + ": " + make + " " + model;
}

public String getCarDescription() {
  return make + " " + model;
}
```

```
public String toString() {
  return getPersonInfo() + " owns a " + getCarInfo();
}

@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result
    + ((firstName == null) ? 0 : firstName.hashCode());
  result = prime * result + id;
  result = prime * result
    + ((lastName == null) ? 0 : lastName.hashCode());
  result = prime * result + ((make == null) ? 0 : make.
     hashCode());
  result = prime * result + ((model == null) ? 0 : model.
     hashCode());
  result = prime * result + vin;
  return result;
}

public PersonCarSpecial(String firstName, String lastName
   , int id,
           String make, String model, int vin) {
  super();
  this.firstName = firstName;
  this.lastName = lastName;
  this.id = id;
  this.make = make;
  this.model = model;
  this.vin = vin;
}

@Override
public boolean equals(Object obj) {
  if (this == obj)
```

```
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    PersonCarSpecial other = (PersonCarSpecial) obj;
    if (firstName == null) {
      if (other.firstName != null)
        return false;
    } else if (!firstName.equals(other.firstName))
      return false;
    if (id != other.id)
      return false;
    if (lastName == null) {
      if (other.lastName != null)
        return false;
    } else if (!lastName.equals(other.lastName))
      return false;
    if (make == null) {
      if (other.make != null)
        return false;
    } else if (!make.equals(other.make))
      return false;
    if (model == null) {
      if (other.model != null)
        return false;
    } else if (!model.equals(other.model))
      return false;
    if (vin != other.vin)
      return false;
    return true;
  }

}
```

Appendix B

Open Source Repositories

This appendix contains information about the open source software discussed in this thesis, including both the refactoring environment and the open source software that was refactored within that environment. The subsections typically include information about version numbers, web sites, and available documentation.

B.1 Refactoring environment

The Eclipse IDE [SDF+03] (`http://www.eclipse.org/`) provides the framework within which ExtC runs. ExtC and the associated plug-ins and libraries mentioned below were tested on Eclipse version 3.6.2, but should run on Eclipse 3.5 and higher.

The IntelliJ IDEA version 8.1 development environment [Jem08] (`http://www.jetbrains.com/idea/`) provides the automated Extract Class refactoring that was used in the experiments.

B.1.1 Plug-ins

The ExtC [CAGA11] (`http://code.google.com/p/ext-c/`) plug-in contains functionality for visualizing Java classes, clustering the class members of Java classes, visualizing clustering, and other functionality described in this thesis. The latest version is available from `http://code.google.com/p/ext-c/downloads/list` (currently, ClassRefactoringPlug-In_1.0.0.201105311759.jar) and the installation instructions can be found at `http://code.google.com/p/`

ext-c/wiki/ExtractClassPlug-InInstallation. ExtC uses the following plug-ins:

Metrics2 (http://metrics2.sourceforge.net/) calculates metrics for Java code and can store the measurements to a database. As of December, 2011, not all of the functionality required by ExtC had been put into a major release. The latest (alpha version) functionality that ExtC uses is available from http://code.google.com/p/ext-c/downloads/list (currently, net.sourceforge.metrics_2.0.0.201105311759.jar).

Apache Derby [Sch08] (http://db.apache.org/derby/) provides a relational database useful for storing metric values and plug-ins to access that database. ExtC was developed using the org.apache.derby.core 10.5.3, org.apache.derby.plug-in.doc, and org.apache.derby.ui plug-ins.

TopicXP [SDGP10] (http://www.cs.wm.edu/semeru/TopicXP/) provides capabilities for analyzing the semantics of Java code. ExtC uses it for producing the "documents" representing the attributes and methods of a class.

B.1.2 Libraries

ExtC uses the following open-source libraries for core functionality:

Archaeopteryx/forester (http://www.phylosoft.org/archaeopteryx/) provides dendrogram viewing and manipulation functionality to the agglomeration view.

JUNG graph framework [OFWB03] (http://jung.sourceforge.net/) provides graph processing algorithms, graph visualization, and graph layout to multiple views.

S-Space [JS10] (http://code.google.com/p/airhead-research/) provides the vector space models used for semantic comparisons in agglomerative clustering.

There are many more open source libraries that are used for auxiliary functionality, both by ExtC and by the included libraries themselves. These are not listed.

B.2 Open source test classes

Much of the clustering and refactoring research in this thesis used open source Java programs as inputs. These open source Java projects came from a variety of domains and are described below.

B.2.1 Qualitas Corpus

The Qualitas Corpus [TAD+10] (`http://qualitascorpus.com/`), version 20101126 [Gro10], contains most of the tested open source projects. The subsections below note when a version was used that is not available from the Qualitas Corpus and specify where the tested versions can be found.

B.2.2 FreeCol

FreeCol (`http://www.freecol.org`) is a multi-player game. `http://www.freecol.org/documentation/subversion.html` discusses how to access FreeCol source code using Subversion. The code we refactored based on the results of betweenness clustering and dual clustering was from version 0.94, revision 7473. Freecol-0.9.4 is available from QualitasCorpus-20101126 and consists of approximately 450 top level public classes. The files we used came from `http://freecol.svn.sourceforge.net/viewvc/freecol/freecol/trunk/src` and are shown in Table B.1.

Table B.1: FreeCol test files

File	Directory
FreeColClient.java	net/sf/freecol/client
FreeColObject.java	net/sf/freecol/common/model
FreeColServer.java	net/sf/freecol/server
ImageLibrary.java	net/sf/freecol/client/gui
Specification.java	net/sf/freecol/common/model

B.2.3 Heritrix

Heritrix (`http://crawler.archive.org/`) is a web crawler. We used heritrix-1.8.0 from the QualitasCorpus-20101126, which consists of approximately 380 top level public classes. The files we used are shown in Table B.2.

Table B.2: Heritrix test files

File	**Directory**
CandidateURI.java	Heritrix/org/archive/crawler/datamodel
CrawlController.java	Heritrix/org/archive/crawler/framework
Heritrix.java	Heritrix/org/archive/crawler
SettingsHandler.java	Heritrix/org/archive/crawler/settings
WorkQueue.java	Heritrix/org/archive/crawler/frontier

B.2.4 Jena

Jena (`http://jena.sourceforge.net/`) is a framework for building semantic web applications. We used jena-2.5.5 from the QualitasCorpus-20101126, which consists of approximately 1100 top level public classes. The files we used are shown in Table B.3.

Table B.3: Jena test files

File	Directory
BRuleEngine.java	Jena/com/hp/hpl/jena/reasoner/rulesys/ impl/oldCode
CommandLine.java	Jena/jena/cmdline
LPBRuleEngine.java	Jena/com/hp/hpl/jena/reasoner/ rulesys/impl
LPInterpreter.java	Jena/com/hp/hpl/jena/reasoner/ rulesys/impl
N3JenaWriterCommon.java	Jena/com/hp/hpl/jena/n3
Node.java	Jena/com/hp/hpl/jena/graph
ParserBase.java	Jena/com/hp/hpl/jena/n3/turtle
Rule.java	Jena/com/hp/hpl/jena/reasoner/rulesys

B.2.5 JHotDraw

JHotDraw (`http://www.jhotdraw.org/`) is a Java GUI framework that is well-known for its use of design patterns. Many people consider it to be a well-designed system, so it is often used as a gold standard when evaluating the effectiveness of certain refactoring techniques [BDLMO10b, DLOV08, SC07, CS06, SSB06]. JHotDraw 5.3.0 is available from the QualitasCorpus-20101126.

B.2.6 Weka

Weka [HFH+09] (`http://www.cs.waikato.ac.nz/ml/weka/`) is a collection of machine learning algorithms. `http://weka.wikispaces.com/Subversion` discusses how to access Weka source code using Subversion. The code we refactored based on the results of betweenness clustering and dual clustering was from version 3.6.3, revision 6972 and consists of approximately

1100 top level public classes. The files we used came from `https://svn.scms.waikato.ac.nz/svn/weka/trunk/weka/src/main/java/` and are shown in Table B.4.

Table B.4: Weka test files

File	**Directory**
BVDecompose.java	weka/classifiers
BVDecomposeSegCVSub.java	weka/classifiers
DatabaseUtils.java	weka/experiment
Experiment.java	weka/experiment
NearestNeighbourSearch.java	weka/core/neighboursearch
Node.java	weka/gui/treevisualizer
RegOptimizer.java	weka/classifiers/functions/supportVector
ResultMatrix.java	weka/experiment
Rule.java	weka/classifiers/trees/m5
Script.java	weka/gui/scripting
TestInstances.java	weka/core
XMLDocument.java	weka/core/xml

Appendix C

Experimental Data

C.1 Preferences

These are the preferences (see Section 2.4.2) in effect when the metrics were collected:

- Constructors, inherited members, inner class members, object methods were filtered out.
- Static members, loggers and loggers were included.
- Only required methods were condensed.

C.2 Agglomerative clustering

We created clusters of class members for each of thirty open source classes in six different ways – using single, complete, and average link agglomerative clustering with a Jaccard distance measure and either the `Sim01` or `Nhood` property sets. This section contains data about the clusters that existed at each of four distances (0.5, 0.75, 0.9, and 0.999) for the six combinations of linkage and property set.

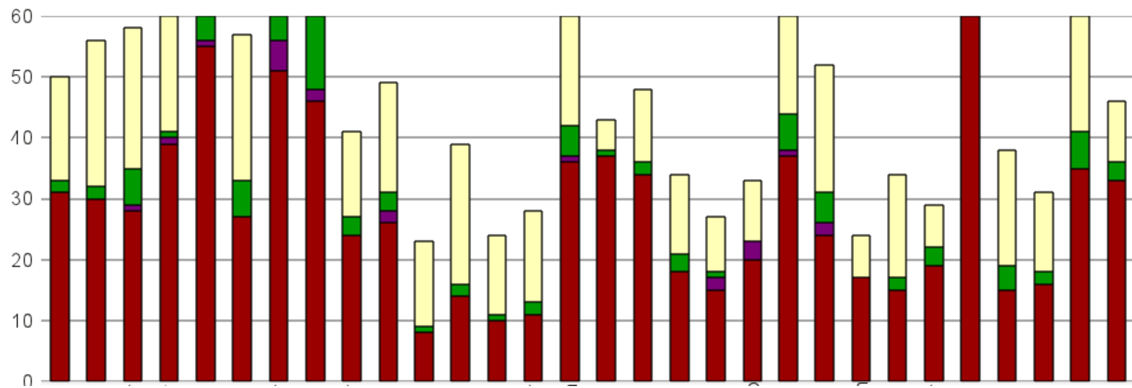
Section C.2.1 summarizes data about the number of clusters for each class, at each of the four cutoff distances, for all six combinations of linkage and property set. Section C.2.2 contains detailed data about the sizes of the clusters for each class, at each of the four cutoff distances, for all six combinations of linkage and property set.

C.2.1 Cluster counts

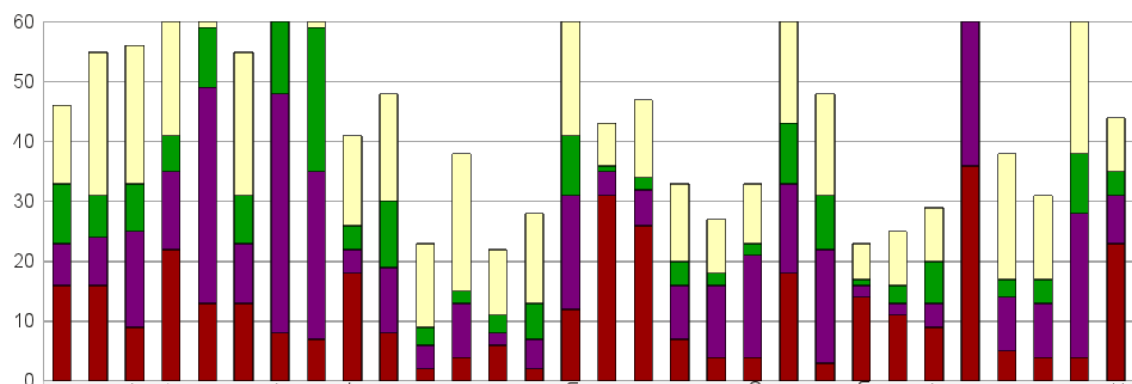
Figures C.1 and C.2 use colored vertical bars to show the number of clusters that exist at various distances when using `Sim01` and `Nhood`, respectively. The vertical bars are truncated at 60 to enhance readability. The distance to color coding is:

- 0.5: yellow
- 0.75: green
- 0.9: purple
- 0.999: red

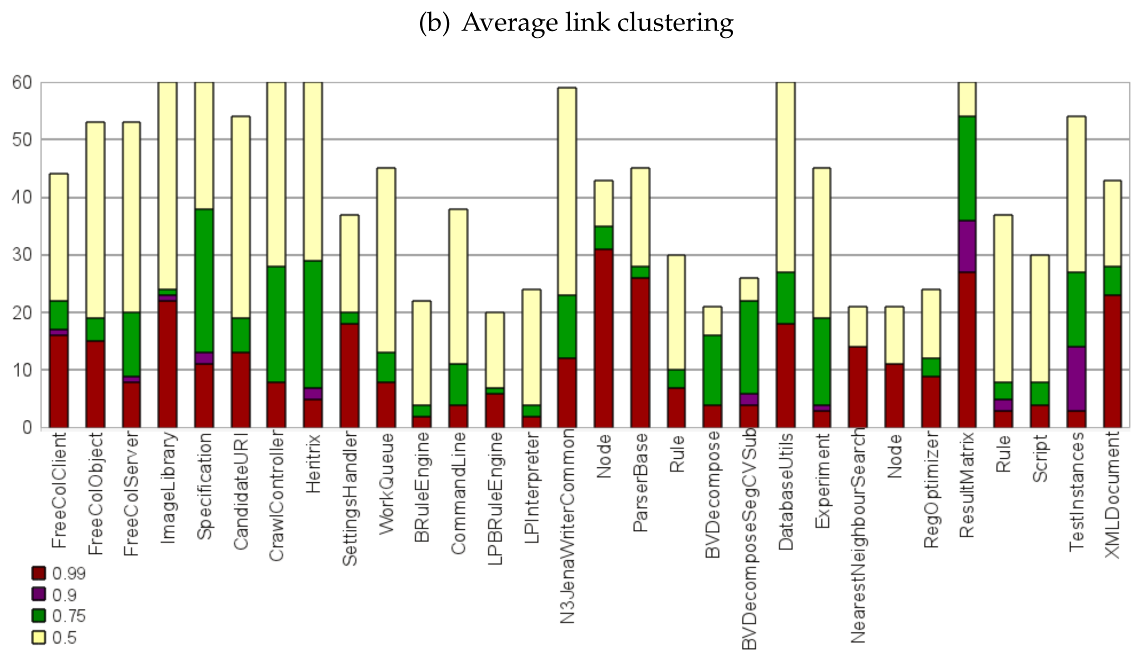
Figure C.1(c) shows single link agglomerative clustering for `Sim01`. For example, the first column shows the number of clusters produced for the members of the `FreeColClient` class, i.e., there are 44 clusters at a distance of 0.5, 22 at a distance of 0.75, 17 at 0.9, and 16 at 0.999. Figures C.1(b) and C.1(a) show the number of clusters for average and complete link agglomerative clustering.



(a) Complete link clustering

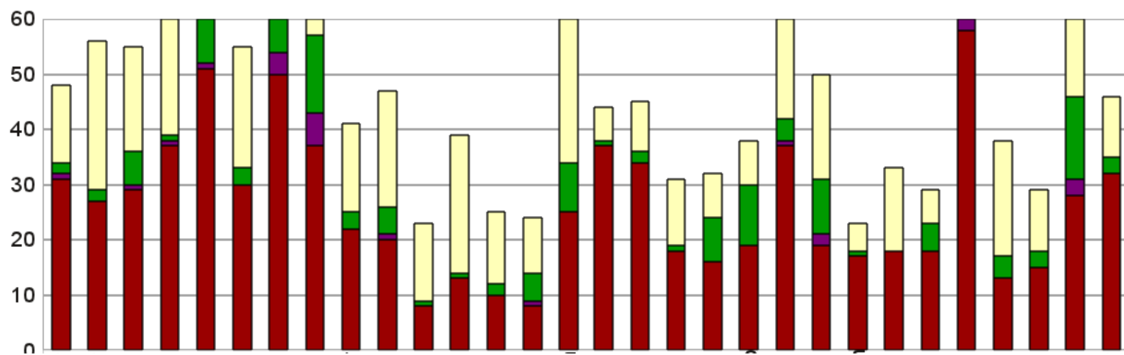


(b) Average link clustering

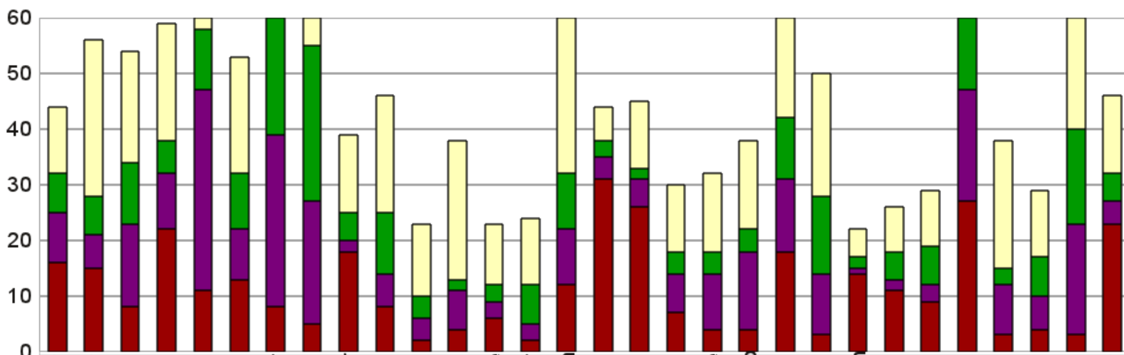


(c) Single link clustering

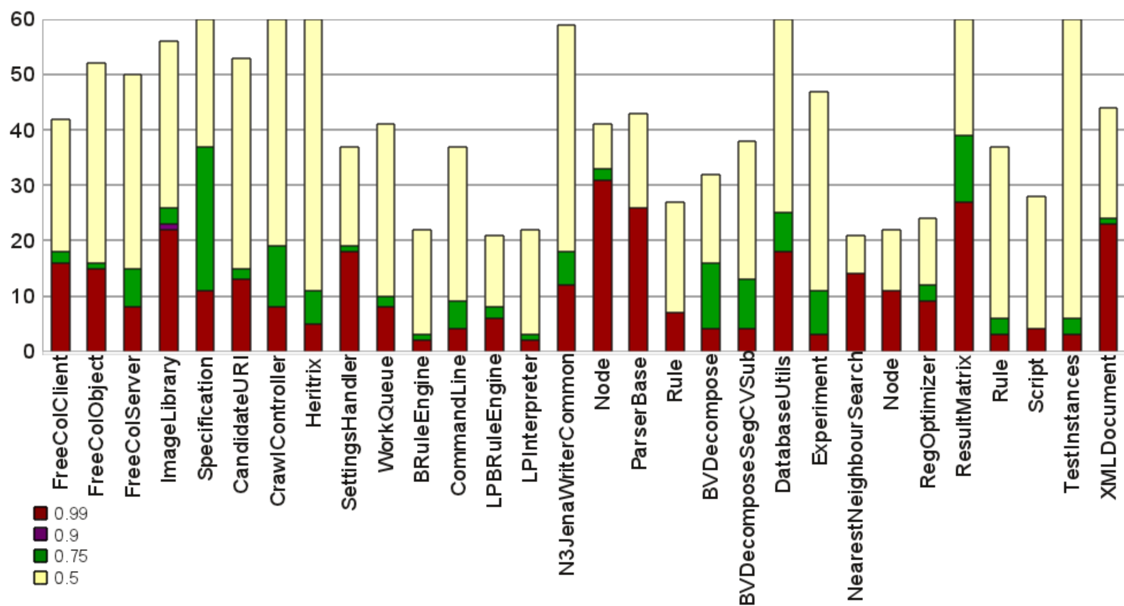
Figure C.1: Number of clusters – agglomerative clustering, Sim01



(a) Complete link clustering



(b) Average link clustering



(c) Single link clustering

Figure C.2: Number of clusters – agglomerative clustering, Nhood

C.2.2 Cluster sizes

The tables in this section contain the sizes of the clusters produced by agglomerative clustering when applied to thirty open source classes. If a class has more than one cluster of a given size, then the number of clusters of that size is given in parentheses.

Sim01 – single link

Table C.1 contains data about the clusters that existed at each of four distances (0.5, 0.75, 0.9, and 0.999) for single link agglomerative clustering using a Jaccard distance function and the `Sim01` property set.

Table C.1: Sim01 – single link

Class	0.5	0.75	0.9	0.99
BRuleEngine	3, 2(4), 1(17)	13, 6(2), 3	22, 6	22, 6
BVDecompose	3(6), 2(5), 1(10)	6, 3(6), 2(5), 1(4)	34, 2, 1(2)	34, 2, 1(2)
BVDecompose-SegCVSub	3(7), 2(7), 1(12)	5, 3(7), 2(7), 1(7)	41, 2, 1(4)	43, 2, 1(2)
CandidateURI	3(3), 2(8), 1(43)	28, 10, 7, 3(2), 2(3), 1(11)	50, 3(2), 2(2), 1(8)	50, 3(2), 2(2), 1(8)
Command-Line	3, 2(3), 1(34)	14, 13, 4, 3, 2(2), 1(5)	37, 3, 2, 1	37, 3, 2, 1
Crawl-Controller	5, 4, 3(3), 2(13), 1(118)	114, 7, 4(2), 3, 2(7), 1(16)	148, 7, 2, 1(5)	148, 7, 2, 1(5)
DatabaseUtils	6, 4, 3(5), 2(2), 1(53)	40, 8, 3(4), 2, 1(20)	58, 8, 1(16)	58, 8, 1(16)
Experiment	3(7), 2(3), 1(35)	25, 4(2), 3(6), 2, 1(9)	59, 1(3)	60, 1(2)
FreeColClient	3(9), 2(13), 1(22)	11, 8, 6(2), 5(2), 3(5), 2(8), 1(3)	29, 8, 6, 5, 3(4), 2(6), 1(3)	35, 8, 5, 3(4), 2(6), 1(3)
FreeColObject	4, 3, 2(5), 1(46)	14, 13(2), 4(2), 2, 1(13)	32, 15, 4, 1(12)	32, 15, 4, 1(12)

Continued on next page

Table C.1 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
FreeColServer	5, 3(3), 2(12), 1(37)	27, 11, 4, 3(3), 2(10), 1(4)	60, 3, 2(5), 1(2)	62, 3, 2(4), 1(2)
Heritrix	16, 3(5), 2(9), 1(91)	37, 26, 24, 7, 5(2), 4, 3(3), 2(4), 1(15)	132, 2(2), 1(4)	135, 2, 1(3)
ImageLibrary	3(2), 2(4), 1(56)	45, 3, 1(22)	46, 3, 1(21)	47, 3, 1(20)
LPBRule-Engine	4(2), 3(2), 2(4), 1(12)	13, 8, 4, 3(2), 2, 1	21, 4, 3(2), 2, 1	21, 4, 3(2), 2, 1
LPInterpreter	5, 3(2), 2(3), 1(18)	18, 11, 5, 1	34, 1	34, 1
N3JenaWriter-Common	3(3), 2(4), 1(52)	33, 8, 4, 3, 2(2), 1(17)	48, 9, 2(2), 1(8)	48, 9, 2(2), 1(8)
Nearest-Neighbour-Search	3(3), 2, 1(17)	12, 3, 2, 1(11)	12, 3, 2, 1(11)	12, 3, 2, 1(11)
Node	2(9), 1(34)	6, 5, 4, 3, 2(3), 1(28)	18, 4, 2, 1(28)	18, 4, 2, 1(28)
Node	5, 4, 3(10), 2, 1(8)	17, 6, 4, 3(7), 1	17, 6, 4, 3(7), 1	17, 6, 4, 3(7), 1
ParserBase	4(2), 3, 2(3), 1(39)	21, 4(2), 2(2), 1(23)	27, 4, 2, 1(23)	27, 4, 2, 1(23)
RegOptimizer	8, 3(2), 2(6), 1(15)	18, 8, 5, 2, 1(8)	33, 1(8)	33, 1(8)
ResultMatrix	3(9), 2(12), 1(113)	47, 46, 7, 5, 4, 3, 2(4), 1(44)	127, 2(2), 1(33)	136, 2(2), 1(24)
Rule	4, 3(3), 2(5), 1(21)	21, 6, 4(2), 3, 2, 1(4)	29, 9, 2, 1(4)	29, 9, 2, 1(4)
Rule	5, 3(2), 2(2), 1(32)	29, 7, 4, 3, 1(4)	43, 1(4)	45, 1(2)
Script	3, 2(5), 1(24)	24, 4, 2(3), 1(3)	34, 1(3)	34, 1(3)
Settings-Handler	4, 3(4), 2(2), 1(30)	12, 8, 6(2), 2(2), 1(14)	18, 9, 6, 2(2), 1(13)	18, 9, 6, 2(2), 1(13)

Continued on next page

Table C.1 – *Continued from previous page*

Class	0.50	0.75	0.90	0.99
Specification	3(8), 2(19), 1(70)	46, 13, 7, 5, 4, 3(7), 2(10), 1(16)	120, 1(12)	122, 1(10)
TestInstances	3(19), 2(5), 1(30)	13, 7(2), 6(2), 3(17), 2(2), 1(3)	62, 3(11), 1(2)	95, 1(2)
WorkQueue	4, 3(2), 2(7), 1(35)	35, 4(2), 3(2), 2(2), 1(6)	43, 4(2), 3, 2, 1(3)	43, 4(2), 3, 2, 1(3)
XML-Document	5, 3(2), 2(6), 1(34)	14, 7(2), 3, 2(2), 1(22)	34, 2, 1(21)	34, 2, 1(21)

Sim01 – average link

Table C.2 contains data about the clusters that existed at each of four distances (0.5, 0.75, 0.9, and 0.999) for average link agglomerative clustering using a Jaccard distance function and the `Sim01` property set.

Table C.2: Sim01 – average link

Class	0.5	0.75	0.9	0.99
BRuleEngine	2(5), 1(18)	6, 5, 4, 3(2), 2(3), 1	7, 6, 5(2), 3, 2	22, 6
BVDecompose	2(11), 1(16)	4, 3(6), 2(5), 1(6)	5, 3(7), 2(4), 1(4)	34, 2, 1(2)
BVDecompose-SegCVSub	2(14), 1(19)	4, 3(7), 2(7), 1(8)	5, 4, 3(6), 2(7), 1(6)	43, 2, 1(2)
CandidateURI	3(2), 2(9), 1(44)	13, 4(3), 3(5), 2(6), 1(16)	13, 7, 5(3), 4(2), 3(2), 2(5), 1(9)	50, 3(2), 2(2), 1(8)
Command-Line	3, 2(3), 1(34)	5(3), 4(4), 3, 2(2), 1(5)	9, 5(2), 4(3), 3(2), 2, 1(4)	37, 3, 2, 1
Crawl-Controller	5, 4, 3(3), 2(13), 1(118)	7, 6(3), 5, 4(3), 3(9), 2(34), 1(25)	16, 13, 7(7), 6, 5(2), 4(3), 3(3), 2(17), 1(13)	148, 7, 2, 1(5)
DatabaseUtils	3(3), 2(7), 1(59)	7, 4(3), 3(7), 2(10), 1(22)	8(3), 5, 4(4), 3(5), 2(2), 1(18)	58, 8, 1(16)

Continued on next page

Table C.2 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
Experiment	3(4), 2(6), 1(38)	5, 4(2), 3(8), 2(5), 1(15)	11, 6, 4(4), 3(5), 2(3), 1(8)	60, 1(2)
FreeColClient	3(7), 2(15), 1(24)	5, 3(11), 2(16), 1(5)	11, 6(2), 5(3), 3(6), 2(8), 1(3)	35, 8, 5, 3(4), 2(6), 1(3)
FreeColObject	3, 2(6), 1(48)	14, 4(2), 3(3), 2(7), 1(18)	15, 5(2), 4(4), 3(2), 2, 1(14)	17, 15(2), 4, 1(12)
FreeColServer	3(2), 2(15), 1(39)	6, 4(2), 3(7), 2(17), 1(6)	8(2), 6, 4(2), 3(7), 2(11), 1(2)	60, 3, 2(5), 1(2)
Heritrix	16, 3(3), 2(11), 1(93)	17, 7, 6, 4(6), 3(8), 2(20), 1(22)	26, 8, 7(4), 6(2), 5(2), 4(5), 3(4), 2(8), 1(8)	131, 3, 2, 1(4)
ImageLibrary	2(6), 1(58)	6, 5, 3(5), 2(10), 1(24)	10, 5(2), 4, 3(6), 2(3), 1(22)	47, 3, 1(20)
LPBRule-Engine	4, 3(2), 2(5), 1(14)	7, 6, 4, 3(2), 2(5), 1	10, 7, 4(2), 3(2), 2, 1	21, 4, 3(2), 2, 1
LPInterpreter	3, 2(5), 1(22)	5(2), 4(3), 2(5), 1(3)	9, 6, 5(3), 4, 1	34, 1
N3JenaWriter-Common	2(7), 1(55)	5, 4, 3(6), 2(9), 1(24)	9, 7, 6, 5, 4(3), 3(2), 2(2), 1(20)	48, 9, 2(2), 1(8)
Nearest-Neighbour-Search	3, 2(3), 1(19)	4(2), 3(2), 2, 1(12)	4(3), 3, 2, 1(11)	12, 3, 2, 1(11)
Node	2(9), 1(34)	6, 4, 3(2), 2(4), 1(28)	6, 5, 4, 3, 2(3), 1(28)	18, 4, 2, 1(28)
Node	3(10), 2(4), 1(11)	5, 4, 3(13), 1	11, 6, 4, 3(9), 1	17, 6, 4, 3(7), 1
ParserBase	3, 2(7), 1(39)	5(2), 4(3), 3, 2(3), 1(25)	6, 5, 4(4), 2(3), 1(23)	27, 4, 2, 1(23)
RegOptimizer	5, 2(8), 1(20)	9, 5, 3(2), 2(5), 1(11)	18, 5, 4(2), 2, 1(8)	33, 1(8)

Continued on next page

Table C.2 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
ResultMatrix	3(3), 2(18), 1(119)	8, 5(3), 4(3), 3(23), 2(5), 1(50)	19, 8(2), 7, 6, 5, 4(6), 3(12), 2(5), 1(41)	127, 2(2), 1(33)
Rule	3(2), 2(7), 1(24)	4(3), 3(6), 2(3), 1(8)	7, 5, 4(2), 3(5), 2(2), 1(5)	29, 9, 2, 1(4)
Rule	5, 3, 2(3), 1(33)	11, 4, 3(7), 2(3), 1(5)	15, 4(2), 3(6), 2, 1(4)	43, 1(4)
Script	2(6), 1(25)	6, 4, 3(2), 2(8), 1(5)	11, 5, 4, 3, 2(5), 1(4)	34, 1(3)
Settings-Handler	3, 2(7), 1(33)	8, 6, 4, 3(3), 2(3), 1(17)	9, 7, 6, 5, 3, 2(3), 1(14)	18, 9, 6, 2(2), 1(13)
Specification	3, 2(26), 1(77)	13, 6, 5, 4(2), 3(12), 2(22), 1(20)	13, 10, 7, 6, 5(2), 4, 3(11), 2(18), 1(13)	120, 1(12)
TestInstances	3(12), 2(12), 1(37)	6, 3(23), 2(8), 1(6)	7(2), 6(2), 5, 4(2), 3(18), 2, 1(2)	92, 3, 1(2)
WorkQueue	3, 2(9), 1(38)	5, 4(2), 3(5), 2(9), 1(13)	11, 9, 5, 4(2), 3(5), 2(2), 1(7)	43, 4(2), 3, 2, 1(3)
XML-Document	4, 3(2), 2(6), 1(35)	7, 6, 3(3), 2(5), 1(25)	7(2), 5, 4, 3(2), 2(3), 1(22)	34, 2, 1(21)

Sim01 – complete link

Table C.3 contains data about the clusters that existed at each of four distances (0.5, 0.75, 0.9, and 0.999) for complete link agglomerative clustering using a Jaccard distance function and the `Sim01` property set.

Table C.3: Sim01 – complete link

Class	0.5	0.75	0.9	0.99
BRuleEngine	2(5), 1(18)	6, 5, 4, 3(2), 2(3), 1	6, 5, 4, 3(3), 2(2)	6, 5, 4, 3(3), 2(2)
BVDecompose	2(11), 1(16)	4, 3(6), 2(5), 1(6)	5, 3(6), 2(5), 1(5)	5, 3(7), 2(5), 1(2)
BVDecompose-SegCVSub	2(14), 1(19)	4, 3(7), 2(7), 1(8)	4, 3(7), 2(7), 1(8)	5, 3(7), 2(9), 1(3)
CandidateURI	2(11), 1(46)	11, 4(2), 3(4), 2(11), 1(15)	13, 4(3), 3(6), 2(8), 1(9)	13, 4(3), 3(6), 2(8), 1(9)
CommandLine	2(4), 1(35)	5(2), 4(4), 3(2), 2(3), 1(5)	5(3), 4(4), 3, 2(3), 1(3)	5(3), 4(4), 3, 2(3), 1(3)
Crawl-Controller	5, 4, 3(2), 2(14), 1(119)	7, 6(3), 5, 4, 3(10), 2(36), 1(26)	7(3), 6(2), 5(4), 4(7), 3(9), 2(23), 1(8)	7(4), 6(4), 5(4), 4(6), 3(7), 2(19), 1(7)
DatabaseUtils	3(2), 2(8), 1(60)	5, 4(2), 3(9), 2(10), 1(22)	8, 5(2), 4(3), 3(7), 2(6), 1(19)	8, 5(3), 4(3), 3(6), 2(5), 1(19)
Experiment	2(10), 1(42)	5, 4(2), 3(8), 2(5), 1(15)	5(2), 4(2), 3(8), 2(6), 1(8)	5(2), 4(3), 3(7), 2(7), 1(5)
FreeColClient	3(3), 2(19), 1(28)	5, 3(11), 2(16), 1(5)	5(3), 3(9), 2(14), 1(5)	5(3), 3(9), 2(14), 1(5)
FreeColObject	2(7), 1(49)	14, 4, 3(3), 2(9), 1(18)	15, 4, 3(4), 2(8), 1(16)	15, 4, 3(4), 2(8), 1(16)
FreeColServer	2(17), 1(41)	6, 3(8), 2(19), 1(7)	8, 6, 5, 3(7), 2(16), 1(3)	8, 6, 5, 4, 3(7), 2(14), 1(3)
Heritrix	16, 3(3), 2(11), 1(93)	17, 7, 6, 4(3), 3(7), 2(26), 1(25)	17, 8, 7, 5(3), 4(5), 3(9), 2(18), 1(10)	17, 8, 7(2), 6, 5(3), 4(3), 3(8), 2(17), 1(10)

Continued on next page

Table C.3 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
ImageLibrary	2(6), 1(58)	6, 5, 3(5), 2(10), 1(24)	6, 5, 3(6), 2(9), 1(23)	7, 5, 3(6), 2(9), 1(22)
LPBRuleEngine	4, 2(7), 1(16)	7, 6, 4, 3(2), 2(5), 1	8, 7, 4, 3(2), 2(4), 1	8, 7, 4, 3(2), 2(4), 1
LPInterpreter	3, 2(5), 1(22)	5(2), 4(2), 3, 2(6), 1(2)	6, 5(2), 4, 3(2), 2(4), 1	6, 5(2), 4, 3(2), 2(4), 1
N3Jena-WriterCommon	2(7), 1(55)	4, 3(6), 2(12), 1(23)	4(4), 3(7), 2(6), 1(20)	5, 4(4), 3(6), 2(5), 1(20)
Nearest-Neighbour-Search	2(4), 1(20)	4(2), 3(2), 2, 1(12)	4(2), 3(2), 2, 1(12)	4(2), 3(2), 2, 1(12)
Node	2(9), 1(34)	4, 3(2), 2(7), 1(28)	4(2), 3(2), 2(5), 1(28)	4(2), 3(2), 2(5), 1(28)
Node	3, 2(13), 1(20)	6, 4, 3(11), 2(2), 1(2)	8, 4, 3(12), 1	8, 4, 3(12), 1
ParserBase	2(8), 1(40)	5, 4(2), 3(3), 2(4), 1(26)	5(2), 4(3), 3, 2(3), 1(25)	5(2), 4(3), 3, 2(3), 1(25)
RegOptimizer	4, 2(9), 1(19)	6, 4, 3(2), 2(7), 1(11)	8, 5, 4, 3(2), 2(4), 1(10)	8, 5, 4, 3(2), 2(4), 1(10)
ResultMatrix	3(3), 2(18), 1(119)	8, 5(3), 4(2), 3(20), 2(12), 1(49)	11, 8, 5(2), 4(5), 3(17), 2(11), 1(42)	15, 8, 7, 5(2), 4(4), 3(18), 2(7), 1(40)
Rule	3, 2(8), 1(25)	4(2), 3(6), 2(5), 1(8)	7, 5, 3(6), 2(4), 1(6)	7, 5, 3(6), 2(4), 1(6)
Rule	5, 3, 2(3), 1(33)	6, 4, 3(8), 2(4), 1(5)	11, 4(3), 3(6), 2, 1(4)	11, 4(3), 3(6), 2, 1(4)
Script	2(6), 1(25)	4, 3(4), 2(8), 1(5)	8, 4, 3(2), 2(7), 1(5)	8, 4, 3(2), 2(7), 1(5)
Settings-Handler	3, 2(7), 1(33)	8, 3(6), 2(4), 1(16)	9, 5(2), 3(4), 2(2), 1(15)	9, 5(2), 3(4), 2(2), 1(15)

Continued on next page

Table C.3 – *Continued from previous page*

Class	0.50	0.75	0.90	0.99
Specification	2(27), 1(78)	13, 6, 5, 4(3), 3(10), 2(22), 1(22)	13, 6, 5, 4(4), 3(10), 2(23), 1(16)	13, 6, 5, 4(5), 3(10), 2(21), 1(16)
TestInstances	2(24), 1(49)	3(23), 2(10), 1(8)	7, 4(3), 3(20), 2(7), 1(4)	7, 4(3), 3(20), 2(7), 1(4)
WorkQueue	2(10), 1(39)	4(2), 3(6), 2(10), 1(13)	5, 4(2), 3(7), 2(7), 1(11)	6, 5, 4, 3(7), 2(7), 1(9)
XMLDocument	4, 2(8), 1(37)	7, 5, 3(2), 2(7), 1(25)	7(2), 4, 3(2), 2(5), 1(23)	7(2), 4, 3(2), 2(5), 1(23)

Nhood – single link

Table C.4 contains data about the clusters that existed at each of four distances (0.5, 0.75, 0.9, and 0.999) for single link agglomerative clustering using a Jaccard distance function and the `Nhood` property set.

Table C.4: Nhood – single link

Class	0.5	0.75	0.9	0.99
BRuleEngine	3, 2(4), 1(17)	19, 6, 3	22, 6	22, 6
BVDecompose	2(6), 1(26)	6, 3(6), 2(5), 1(4)	34, 2, 1(2)	34, 2, 1(2)
BVDecompose-SegCVSub	2(9), 1(29)	14, 9, 3(6), 2, 1(4)	43, 2, 1(2)	43, 2, 1(2)
CandidateURI	3(2), 2(11), 1(40)	32, 17, 3(2), 2(2), 1(9)	50, 3(2), 2(2), 1(8)	50, 3(2), 2(2), 1(8)
Command-Line	3(2), 2(2), 1(33)	10, 9, 5(2), 4(2), 3, 2, 1	37, 3, 2, 1	37, 3, 2, 1
Crawl-Controller	5, 3(3), 2(15), 1(118)	117, 8, 7, 5, 4, 3(2), 2(3), 1(9)	148, 7, 2, 1(5)	148, 7, 2, 1(5)
DatabaseUtils	5, 3(6), 2(2), 1(55)	35, 8, 7, 4, 3(3), 2, 1(17)	58, 8, 1(16)	58, 8, 1(16)
Experiment	3(3), 2(9), 1(35)	40, 5, 3(3), 2(2), 1(4)	60, 1(2)	60, 1(2)

Continued on next page

Table C.4 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
FreeColClient	5, 3(7), 2(15), 1(19)	27, 8, 6, 5, 3(4), 2(7), 1(3)	35, 8, 5, 3(4), 2(6), 1(3)	35, 8, 5, 3(4), 2(6), 1(3)
FreeColObject	4, 3(2), 2(4), 1(45)	32, 14, 4, 1(13)	32, 15, 4, 1(12)	32, 15, 4, 1(12)
FreeColServer	5, 3(3), 2(15), 1(31)	38, 9, 5, 3(2), 2(7), 1(3)	62, 3, 2(4), 1(2)	62, 3, 2(4), 1(2)
Heritrix	16, 3(7), 2(12), 1(79)	116, 8, 4, 2(4), 1(4)	135, 2, 1(3)	135, 2, 1(3)
ImageLibrary	3(4), 2(6), 1(46)	28, 10, 5, 3(2), 1(21)	46, 3, 1(21)	47, 3, 1(20)
LPBRule-Engine	4(2), 3(2), 2(3), 1(14)	8(2), 5, 4, 3(2), 2, 1	21, 4, 3(2), 2, 1	21, 4, 3(2), 2, 1
LPInterpreter	5, 3(3), 2(3), 1(15)	28, 6, 1	34, 1	34, 1
N3Jena-WriterCommon	3(2), 2(6), 1(51)	31, 9, 6, 4, 3, 2(3), 1(10)	48, 9, 2(2), 1(8)	48, 9, 2(2), 1(8)
Nearest-Neighbour-Search	3(2), 2(3), 1(16)	12, 3, 2, 1(11)	12, 3, 2, 1(11)	12, 3, 2, 1(11)
Node	4, 3, 2(6), 1(33)	10, 5, 4, 3, 2, 1(28)	18, 4, 2, 1(28)	18, 4, 2, 1(28)
Node	6, 4, 3(8), 2(3), 1(9)	17, 6, 4, 3(7), 1	17, 6, 4, 3(7), 1	17, 6, 4, 3(7), 1
ParserBase	4, 3(3), 2(4), 1(35)	27, 4, 2, 1(23)	27, 4, 2, 1(23)	27, 4, 2, 1(23)
RegOptimizer	8, 3(2), 2(6), 1(15)	18, 8, 5, 2, 1(8)	33, 1(8)	33, 1(8)
ResultMatrix	8, 4, 3(2), 2(15), 1(116)	91, 8, 7(2), 5, 4(2), 2(6), 1(26)	136, 2(2), 1(24)	136, 2(2), 1(24)
Rule	4(2), 3(3), 2(5), 1(17)	29, 9, 2, 1(4)	29, 9, 2, 1(4)	29, 9, 2, 1(4)

Continued on next page

Table C.4 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
Rule	5, 3(2), 2(2), 1(32)	31, 7, 4, 3, 1(2)	45, 1(2)	45, 1(2)
Script	3, 2(7), 1(20)	34, 1(3)	34, 1(3)	34, 1(3)
Settings-Handler	5, 4, 3, 2(4), 1(30)	18, 8, 6, 2(2), 1(14)	18, 9, 6, 2(2), 1(13)	18, 9, 6, 2(2), 1(13)
Specification	3(9), 2(20), 1(65)	43, 14, 7, 5, 4(2), 3(7), 2(10), 1(14)	122, 1(10)	122, 1(10)
TestInstances	7, 5, 3, 2(6), 1(70)	85, 7, 2, 1(3)	95, 1(2)	95, 1(2)
WorkQueue	4(2), 3(3), 2(6), 1(30)	39, 4(2), 3(2), 2, 1(4)	43, 4(2), 3, 2, 1(3)	43, 4(2), 3, 2, 1(3)
XMLDocument	3(4), 2(5), 1(35)	27, 7, 2, 1(21)	34, 2, 1(21)	34, 2, 1(21)

Nhood – average link

Table C.5 contains data about the clusters that existed at each of four distances (0.5, 0.75, 0.9, and 0.999) for average link agglomerative clustering using a Jaccard distance function and the `Nhood` property set.

Table C.5: Nhood – average link

Class	**0.5**	**0.75**	**0.9**	**0.99**
BRuleEngine	2(5), 1(18)	6, 4, 3(3), 2(4), 1	9, 6(2), 3, 2(2)	22, 6
BVDecompose	2(6), 1(26)	3(7), 2(6), 1(5)	7, 3(7), 2(4), 1(2)	34, 2, 1(2)
BVDecompose-SegCVSub	2(9), 1(29)	3(8), 2(9), 1(5)	6(2), 3(6), 2(7), 1(3)	43, 2, 1(2)
CandidateURI	3(2), 2(11), 1(40)	8, 4, 3(7), 2(12), 1(11)	15, 7, 6, 4(2), 3(6), 2(3), 1(8)	50, 3(2), 2(2), 1(8)
Command-Line	3, 2(3), 1(34)	5(2), 4(4), 3(4), 2(2), 1	6(2), 5(2), 4(3), 3(2), 2, 1	37, 3, 2, 1

Continued on next page

Table C.5 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
Crawl-Controller	5, 3(2), 2(16), 1(119)	7, 6(2), 5(3), 4(3), 3(11), 2(31), 1(21)	16(2), 10, 8, 7(6), 6, 5(2), 4(2), 3(5), 2(12), 1(7)	148, 7, 2, 1(5)
DatabaseUtils	3, 2(9), 1(61)	7, 4, 3(9), 2(13), 1(18)	9, 8, 7, 6(2), 5(2), 3(4), 2(4), 1(16)	58, 8, 1(16)
Experiment	2(12), 1(38)	5(3), 4, 3(5), 2(9), 1(10)	21, 6(2), 4(3), 3(3), 2(3), 1(2)	60, 1(2)
FreeColClient	3(7), 2(17), 1(20)	5, 4(3), 3(7), 2(16), 1(5)	9, 6, 5(2), 4(3), 3(6), 2(8), 1(4)	35, 8, 5, 3(4), 2(6), 1(3)
FreeColObject	2(7), 1(49)	14, 4(2), 3(4), 2(8), 1(13)	15, 7, 6, 4(5), 3, 1(12)	32, 15, 4, 1(12)
FreeColServer	3, 2(19), 1(34)	5, 3(9), 2(19), 1(5)	11, 8, 7, 4(3), 3(5), 2(10), 1(2)	62, 3, 2(4), 1(2)
Heritrix	16, 3(4), 2(15), 1(82)	16, 5(2), 4(4), 3(12), 2(26), 1(10)	26, 11, 9(2), 7(3), 6(2), 5(3), 4(4), 3(2), 2(6), 1(3)	135, 2, 1(3)
ImageLibrary	3, 2(9), 1(49)	6, 5, 3(8), 2(7), 1(21)	10(2), 5(2), 3(5), 2(2), 1(21)	47, 3, 1(20)
LPBRule-Engine	4, 3(2), 2(4), 1(16)	6, 5, 4, 3(2), 2(6), 1	8, 6, 5, 4, 3(2), 2(2), 1	21, 4, 3(2), 2, 1
LPInterpreter	3(3), 2(5), 1(16)	5(2), 4(3), 3, 2(4), 1(2)	21, 6, 4, 3, 1	34, 1
N3Jena-WriterCommon	3, 2(7), 1(52)	6, 4(3), 3(8), 2(7), 1(13)	9, 8, 6(2), 5, 4(3), 3(3), 2(3), 1(8)	48, 9, 2(2), 1(8)
Nearest-Neighbour-Search	3, 2(4), 1(17)	4, 3(3), 2(2), 1(11)	8, 4, 3, 2, 1(11)	12, 3, 2, 1(11)
Node	2(8), 1(36)	4(2), 3(2), 2(4), 1(30)	6, 5, 4, 3, 2(3), 1(28)	18, 4, 2, 1(28)
Node	3(8), 2(7), 1(11)	6, 4, 3(8), 2(7), 1	13, 6, 4, 3(7), 2(2), 1	17, 6, 4, 3(7), 1

Continued on next page

Table C.5 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
ParserBase	3(2), 2(7), 1(36)	5(2), 4(3), 3, 2(4), 1(23)	6, 5(3), 4(2), 2(2), 1(23)	27, 4, 2, 1(23)
RegOptimizer	5, 2(8), 1(20)	9, 5, 3(2), 2(6), 1(9)	18, 8, 5, 2, 1(8)	33, 1(8)
ResultMatrix	8, 3(2), 2(17), 1(116)	8(2), 7, 5(4), 4(4), 3(9), 2(26), 1(26)	29, 19, 14, 11, 7, 5(3), 4(6), 3(4), 2(4), 1(25)	136, 2(2), 1(24)
Rule	4, 3, 2(9), 1(19)	6, 4(3), 3(3), 2(6), 1(5)	7, 6, 4(5), 3, 2(2), 1(4)	29, 9, 2, 1(4)
Rule	5, 3, 2(3), 1(33)	11, 4(2), 3(7), 2(2), 1(3)	17, 4(2), 3(6), 2, 1(2)	45, 1(2)
Script	2(8), 1(21)	4, 3(5), 2(7), 1(4)	11, 6, 4(3), 3, 2, 1(3)	34, 1(3)
Settings-Handler	4(2), 2(5), 1(32)	8, 6, 4, 3(3), 2(4), 1(15)	9, 8, 7, 6, 3, 2(2), 1(13)	18, 9, 6, 2(2), 1(13)
Specification	3, 2(28), 1(73)	14, 6, 5, 4, 3(13), 2(23), 1(18)	15, 9, 7, 5(2), 4(3), 3(11), 2(18), 1(10)	122, 1(10)
TestInstances	5, 3, 2(8), 1(73)	17, 13, 5, 3(6), 2(13), 1(18)	39, 7(2), 6, 5(2), 4, 3(2), 2(4), 1(10)	95, 1(2)
WorkQueue	3(2), 2(9), 1(35)	4(4), 3(8), 2(6), 1(7)	11, 10, 6, 5(2), 4(2), 3(3), 2, 1(3)	43, 4(2), 3, 2, 1(3)
XML-Document	3(2), 2(7), 1(37)	7, 6, 5, 3(3), 2(4), 1(22)	8, 7(2), 6(2), 2, 1(21)	34, 2, 1(21)

Nhood – complete link

Table C.6 contains data about the clusters that existed at each of four distances (0.5, 0.75, 0.9, and 0.999) for complete link agglomerative clustering using a Jaccard distance function and the `Nhood` property set.

Table C.6: Nhood – complete link

Class	0.5	0.75	0.9	0.99
BRuleEngine	2(5), 1(18)	6, 5, 4, 3, 2(5)	6(2), 5, 3, 2(4)	6(2), 5, 3, 2(4)
BVDecompose	2(6), 1(26)	3, 2(12), 1(11)	7, 4, 3, 2(11), 1(2)	7, 4, 3, 2(11), 1(2)
BVDecompose-SegCVSub	2(9), 1(29)	2(17), 1(13)	7, 5, 4, 3, 2(13), 1(2)	7, 5, 4, 3, 2(13), 1(2)
CandidateURI	2(13), 1(42)	8, 3(7), 2(14), 1(11)	11, 3(8), 2(12), 1(9)	11, 3(8), 2(12), 1(9)
Command-Line	2(4), 1(35)	5(2), 4(4), 3(3), 2(3), 1(2)	5(2), 4(5), 3(2), 2(3), 1	5(2), 4(5), 3(2), 2(3), 1
Crawl-Controller	5, 3(2), 2(16), 1(119)	7, 6(2), 5(2), 4(2), 3(8), 2(38), 1(25)	9, 7, 6(4), 5(3), 4(7), 3(10), 2(21), 1(7)	9, 7(2), 6(4), 5(4), 4(8), 3(7), 2(18), 1(6)
DatabaseUtils	3, 2(9), 1(61)	7, 4, 3(9), 2(13), 1(18)	7, 5, 4(3), 3(8), 2(9), 1(16)	7, 5, 4(4), 3(8), 2(7), 1(16)
Experiment	2(12), 1(38)	5, 4(2), 3(4), 2(13), 1(11)	6(2), 5(2), 4, 3(6), 2(8), 1(2)	9, 8, 6, 5, 4, 3(4), 2(8), 1(2)
FreeColClient	3(3), 2(21), 1(24)	5, 3(9), 2(19), 1(5)	5(2), 4, 3(8), 2(16), 1(5)	5(3), 3(8), 2(16), 1(4)
FreeColObject	2(7), 1(49)	14, 4(2), 3(3), 2(9), 1(14)	15, 4(2), 3(4), 2(8), 1(12)	15, 4(2), 3(4), 2(8), 1(12)
FreeColServer	2(20), 1(35)	3(9), 2(21), 1(6)	6, 5(2), 4, 3(6), 2(17), 1(3)	6, 5(2), 4(2), 3(6), 2(15), 1(3)
Heritrix	16, 3(4), 2(15), 1(82)	16, 5(2), 4(4), 3(10), 2(28), 1(12)	22, 6(2), 5(3), 4(7), 3(7), 2(19), 1(4)	27, 6(4), 5(3), 4(6), 3(7), 2(13), 1(3)
ImageLibrary	2(10), 1(50)	8, 3(8), 2(8), 1(22)	8, 3(9), 2(7), 1(21)	9, 3(9), 2(7), 1(20)

Continued on next page

Table C.6 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
LPBRule-Engine	4, 2(6), 1(18)	6, 5, 4, 3(2), 2(6), 1	7, 6, 4(2), 3(2), 2(3), 1	7, 6, 4(2), 3(2), 2(3), 1
LPInterpreter	3(3), 2(5), 1(16)	5(2), 4, 3(2), 2(6), 1(3)	7, 6(2), 5, 4, 2(3), 1	13, 6, 5, 4, 2(3), 1
N3Jena-WriterCommon	3, 2(7), 1(52)	4(3), 3(9), 2(8), 1(14)	7, 6(4), 4(2), 3(4), 2(4), 1(10)	7, 6(4), 4(2), 3(4), 2(4), 1(10)
Nearest-Neighbour-Search	2(5), 1(18)	4, 3(2), 2(3), 1(12)	5, 3(2), 2(3), 1(11)	5, 3(2), 2(3), 1(11)
Node	2(8), 1(36)	4, 3(3), 2(5), 1(29)	4(2), 3(2), 2(5), 1(28)	4(2), 3(2), 2(5), 1(28)
Node	3, 2(14), 1(18)	6, 4, 3(8), 2(7), 1	6, 4, 3(8), 2(7), 1	6, 4, 3(8), 2(7), 1
ParserBase	3(2), 2(7), 1(36)	5, 4, 3(3), 2(7), 1(24)	5(2), 4(2), 3, 2(6), 1(23)	5(2), 4(2), 3, 2(6), 1(23)
RegOptimizer	4, 2(9), 1(19)	6, 4, 2(10), 1(11)	9, 5, 4, 3, 2(6), 1(8)	9, 5, 4, 3, 2(6), 1(8)
ResultMatrix	8, 3(2), 2(17), 1(116)	8, 7, 5(3), 4(6), 3(7), 2(30), 1(29)	19, 14, 8, 7, 5(2), 4(7), 3(6), 2(18), 1(24)	19, 17, 10, 7, 5(2), 4(8), 3(5), 2(15), 1(24)
Rule	3(2), 2(9), 1(20)	4(2), 3(6), 2(7), 1(4)	5, 4(2), 3(5), 2(6), 1(4)	5, 4(2), 3(5), 2(6), 1(4)
Rule	5, 3, 2(3), 1(33)	6(2), 3(8), 2(4), 1(3)	11, 6, 4(2), 3(6), 2, 1(2)	11, 6, 4(2), 3(6), 2, 1(2)
Script	2(8), 1(21)	4(2), 3(3), 2(7), 1(6)	5, 4(3), 3(2), 2(5), 1(4)	5, 4(3), 3(2), 2(5), 1(4)
Settings-Handler	3, 2(7), 1(33)	8, 6, 4, 3(3), 2(4), 1(15)	9, 6(2), 5, 3(2), 2(2), 1(14)	9, 6(2), 5, 3(2), 2(2), 1(14)
Specification	3, 2(28), 1(73)	14, 6, 4, 3(12), 2(26), 1(20)	15, 6, 5, 4(3), 3(12), 2(24), 1(10)	15, 6, 5, 4(4), 3(12), 2(22), 1(10)

Continued on next page

Table C.6 – *Continued from previous page*

Class	**0.50**	**0.75**	**0.90**	**0.99**
TestInstances	5, 3, 2(8), 1(73)	10, 7(2), 6, 3(4), 2(17), 1(21)	17, 8(2), 7, 5, 4(2), 3(5), 2(10), 1(9)	25, 8, 7, 5(2), 4(2), 3(4), 2(10), 1(7)
WorkQueue	3, 2(10), 1(36)	4(2), 3(9), 2(9), 1(6)	6, 5(2), 4(3), 3(6), 2(4), 1(5)	6(2), 5, 4(3), 3(6), 2(4), 1(4)
XML-Document	3(2), 2(7), 1(37)	7, 3(5), 2(6), 1(23)	7, 6, 5, 3(3), 2(4), 1(22)	7, 6, 5, 3(3), 2(4), 1(22)

C.3 Betweenness clustering

C.3.1 Cluster sizes

Table C.7 shows the sizes of the clusters produced by betweenness clustering when applied to the thirty large open source classes identified using the query described in Section 4.1. The numbers in the "Cluster sizes" column list the sizes of each cluster. If a class has more than one cluster of a given size, then the number of clusters of that size is given in parentheses. For example, `CrawlController` has five clusters – one of 147 members, one of 7 members, and three with a single member.

C.3.2 Non-cohesion metrics

Table C.8 contains the metric results for number of fields (NF), number of methods (NM), and weighted methods per class (WMC), where the WMC values are weighted using computational complexity [McC76]. The "Class" column contains entries for the *original* class (before it was refactored), the *modified* class (the larger class after refactoring), and the *extracted* class (the smaller class after refactoring). The Δ *modified* rows contain the improvement in the given measurement between the original and modified classes, and the Δ *extracted* rows contain the improvement in the given measurement between the original and extracted classes.

Table C.7: Betweenness cluster sizes

Class	# Clusters	Cluster sizes
BRuleEngine	3	19, 6, 3
BVDecompose	3	31, 2, 1
BVDecomposeSegCVSub	3	40, 2, 1
CandidateURI	11	33, 15, 3(2), 2(2), 1(5)
CommandLine	5	23, 14, 3, 2, 1
CrawlController	5	147, 7, 1(3)
DatabaseUtils	18	58, 8, 1(16)
Experiment	3	55, 3, 1
FreeColClient	16	35, 8, 5, 3(4), 2(6), 1(3)
FreeColObject	15	32, 15, 4, 1(12)
FreeColServer	9	55, 7, 3, 2(4), 1(2)
Heritrix	4	119, 9, 2, 1
ImageLibrary	23	42, 5, 3, 1(20)
LPBRuleEngine	7	13, 8, 4, 3(2), 2, 1
LPInterpreter	3	28, 6, 1
N3JenaWriterCommon	11	46, 10, 2, 1(8)
NearestNeighbourSearch	14	6, 3, 3, 2, 1(10)
Node	32	13, 5, 4, 2, 1(28)
Node	12	15, 6, 4, 3(7), 2, 1
ParserBase	27	15, 12, 4, 2, 1(23)
RegOptimizer	8	25, 7, 1(6)
ResultMatrix	28	127, 7, 2(2), 1(24)
Rule	7	29, 9, 2, 1(4)
Rule	4	42, 3, 1(2)
Script	4	27, 6, 1(2)
SettingsHandler	18	18, 9, 6, 2(2), 1(13)
Specification	12	76, 46, 1(10)
TestInstances	3	90, 3, 1
WorkQueue	5	30, 12, 4(2), 3
XMLDocument	24	27, 7, 2, 1(21)

Table C.8: Betweenness clustering – non-cohesion metric results

Class	NF	NM	WMC
CandidateURI (H)	16	56	84
modified	17	58	81
extracted	1	16	21
Δ modified	-1	-2	3
Δ extracted	15	40	63
CommandLine (J)	8	36	66
modified	8	36	59
extracted	1	15	22
Δ modified	0	0	7
Δ extracted	7	21	44
CrawlController (H)	63	100	247
modified	62	100	230
extracted	2	6	23
Δ modified	1	0	17
Δ extracted	61	94	224
DatabaseUtils (W)	36	47	194
modified	35	47	190
extracted	2	7	11
Δ modified	1	0	4
Δ extracted	34	40	183
FreeColClient (F)	28	48	94
modified	27	48	89
extracted	2	7	12
Δ modified	1	0	5
Δ extracted	26	41	82
FreeColObject (F)	10	53	99
modified	10	53	85
extracted	1	15	29
Δ modified	0	0	14
Δ extracted	9	38	70

Continued on next page

Table C.8 – *Continued from previous page*

Class	**NF**	**NM**	**WMC**
FreeColServer (F)	26	52	181
modified	24	52	180
extracted	3	8	9
Δ modified	2	0	1
Δ extracted	23	44	172
Heritrix (H)	47	97	293
modified	47	97	291
extracted	2	11	13
Δ modified	0	0	2
Δ extracted	45	86	280
LPBRuleEngine (J)	10	26	47
modified	10	27	43
extracted	2	9	14
Δ modified	0	-1	4
Δ extracted	8	17	33
N3JenaWriterCommon (J)	23	46	136
modified	22	47	120
extracted	3	11	28
Δ modified	1	-1	16
Δ extracted	20	35	108
ParserBase (J)	19	38	93
modified	17	37	88
extracted	4	10	14
Δ modified	2	1	5
Δ extracted	15	28	79
RegOptimizer (W)	21	22	48
modified	20	22	46
extracted	2	11	13
Δ modified	1	0	2
Δ extracted	19	11	35

Continued on next page

Table C.8 – *Continued from previous page*

Class	NF	NM	WMC
ResultMatrix (W)	41	127	305
modified	41	127	305
extracted	2	9	11
Δ modified	0	0	0
Δ extracted	39	118	294
Rule (J)	7	43	109
modified	7	34	82
extracted	0	9	27
Δ modified	0	9	27
Δ extracted	7	34	82
SettingsHandler (H)	22	29	60
modified	14	28	50
extracted	8	1	10
Δ modified	8	1	10
Δ extracted	14	28	50
Specification (F)	47	87	155
modified	41	87	121
extracted	7	44	78
Δ modified	6	0	34
Δ extracted	40	43	77
XMLDocument (W)	24	39	61
modified	24	40	61
extracted	2	8	9
Δ modified	0	-1	0
Δ extracted	22	31	52

C.3.3 Cohesion metrics

Table C.9 shows the values of six structural cohesion metrics – LCOM, LCOM*, TCC, DC_D, LCC, and DC_I, together with the C3V semantic cohesion metric. The "Class" column contains entries for the *original* class (before it was refactored), the *modified* class (the larger class after refactoring), and the *extracted* class (the smaller class after refactoring). The Δ *modified* rows contain the improvement in the given measurement between the original and modified classes, and the Δ *extracted* rows

contain the improvement in the given measurement between the original and extracted classes. Due to rounding, the improvement rows may sometimes appear to be off by 0.01.

Table C.9: Betweenness clustering – cohesion results

Class	**LCOM**	**LCOM***	**TCC**	**DCD**	**LCC**	**DCI**	**C3V**
CandidateURI (H)	1153	0.98	0.13	0.15	0.25	0.33	0.06
modified	1054	0.97	0.22	0.24	0.58	0.64	0.06
extracted	91	0.00	0.00	0.01	0.00	0.01	0.32
Δ modified	99	0.02	0.09	0.09	0.33	0.31	0.01
Δ extracted	1062	0.98	-0.13	-0.14	-0.25	-0.32	0.27
CommandLine (J)	507	0.93	0.22	0.22	0.64	0.64	0.17
modified	355	0.90	0.22	0.22	0.64	0.64	0.14
extracted	61	0.00	1.00	1.00	1.00	1.00	0.56
Δ modified	152	0.03	0.00	0.00	0.00	0.00	-0.03
Δ extracted	446	0.93	0.78	0.78	0.36	0.36	0.39
CrawlController (H)	3957	0.97	0.20	0.18	0.73	0.76	0.04
modified	3957	0.97	0.20	0.18	0.73	0.76	0.04
extracted	0	0.00	1.00	1.00	1.00	1.00	0.83
Δ modified	0	0.00	0.00	0.00	0.00	0.00	-0.00
Δ extracted	3957	0.97	0.80	0.82	0.28	0.25	0.79
DatabaseUtils (W)	641	0.98	0.26	0.26	0.55	0.53	0.07
modified	623	0.98	0.26	0.26	0.55	0.53	0.06
extracted	3	0.60	0.60	0.60	1.00	1.00	0.38
Δ modified	18	0.00	0.01	0.01	0.00	0.00	-0.01
Δ extracted	638	0.38	0.35	0.35	0.45	0.47	0.31
FreeColClient (F)	1011	0.98	0.05	0.05	0.11	0.11	0.04
modified	985	0.97	0.06	0.06	0.11	0.11	0.04
extracted	11	0.80	0.60	0.60	1.00	1.00	0.32
Δ modified	26	0.00	0.01	0.01	0.00	0.00	0.00
Δ extracted	1000	0.18	0.55	0.55	0.89	0.89	0.28

Continued on next page

Table C.9 – *Continued from previous page*

Class	LCOM	LCOM*	TCC	DCD	LCC	DCI	C3V
FreeColObject (F)	1134	0.97	0.11	0.20	0.11	0.20	0.13
modified	1134	0.97	0.11	0.20	0.11	0.20	0.13
extracted	0	0.00	1.00	1.00	1.00	1.00	0.50
Δ modified	0	0.00	0.00	0.00	0.00	0.00	0.00
Δ extracted	1134	0.97	0.89	0.81	0.89	0.81	0.37
FreeColServer (F)	1032	0.97	0.14	0.14	0.49	0.49	0.06
modified	1016	0.97	0.15	0.15	0.49	0.49	0.06
extracted	7	0.72	0.43	0.43	1.00	1.00	0.46
Δ modified	16	0.00	0.00	0.00	0.00	0.00	-0.00
Δ extracted	1025	0.25	0.29	0.29	0.51	0.51	0.41
Heritrix (H)	4044	0.99	0.20	0.29	0.55	0.77	0.06
modified	4006	0.99	0.20	0.28	0.55	0.77	0.06
extracted	0	0.56	1.00	1.00	1.00	1.00	0.41
Δ modified	38	0.00	-0.01	-0.01	0.00	0.00	0.00
Δ extracted	4044	0.43	0.80	0.71	0.45	0.23	0.35
LPBRuleEngine (J)	184	0.92	0.32	0.32	0.45	0.45	0.08
modified	182	0.92	0.21	0.21	0.47	0.47	0.09
extracted	0	0.57	0.89	0.89	1.00	1.00	0.36
Δ modified	2	0.01	-0.11	-0.11	0.02	0.02	0.00
Δ extracted	184	0.35	0.58	0.58	0.55	0.55	0.28
N3JenaWriterC... (J)	905	0.99	0.22	1.00	0.30	1.00	0.06
modified	867	0.98	0.17	0.60	0.19	0.60	0.08
extracted	41	0.93	0.62	1.00	0.62	1.00	0.19
Δ modified	38	0.01	-0.05	-0.40	-0.11	-0.40	0.02
Δ extracted	864	0.06	0.41	0.00	0.32	0.00	0.13
ParserBase (J)	624	1.00	0.06	0.00	0.08	0.00	0.08
modified	546	1.00	0.07	0.00	0.09	0.00	0.08
extracted	22	0.88	0.56	0.40	0.78	0.60	0.20
Δ modified	78	0.00	0.02	0.00	0.01	0.00	0.00
Δ extracted	602	0.13	0.50	0.40	0.70	0.60	0.12

Continued on next page

Table C.9 – *Continued from previous page*

Class	LCOM	LCOM*	TCC	DCD	LCC	DCI	C3V
RegOptimizer (W)	107	0.91	0.29	0.19	0.78	0.23	0.18
modified	31	0.89	0.40	0.31	0.89	0.31	0.14
extracted	33	0.78	0.60	0.60	0.80	0.80	0.21
Δ modified	76	0.02	0.11	0.12	0.11	0.08	-0.04
Δ extracted	74	0.13	0.31	0.41	0.02	0.57	0.03
ResultMatrix (W)	4750	0.98	0.13	0.15	0.60	0.68	0.10
modified	4750	0.98	0.13	0.15	0.60	0.68	0.10
extracted	0	0.43	0.71	0.71	1.00	1.00	0.59
Δ modified	0	0.00	0.00	0.00	0.00	0.00	-0.00
Δ extracted	4750	0.55	0.58	0.57	0.41	0.32	0.49
Rule (J)	550	0.93	0.17	0.19	0.42	0.44	0.07
modified	271	0.90	0.28	0.28	0.69	0.69	0.08
extracted	0	0.00	0.00	0.50	0.00	0.60	0.17
Δ modified	279	0.03	0.11	0.09	0.27	0.25	0.02
Δ extracted	550	0.93	-0.17	0.31	-0.42	0.16	0.11
SettingsHandler (H)	187	0.99	0.32	0.49	0.36	0.58	0.08
modified	165	0.99	0.35	0.49	0.40	0.58	0.11
extracted	0	0.00	1.00	1.00	1.00	1.00	0.01
Δ modified	22	0.00	0.03	0.00	0.04	0.00	0.03
Δ extracted	187	0.99	0.68	0.51	0.64	0.43	-0.07
Specification (F)	3364	0.99	0.14	0.13	0.98	0.70	0.15
modified	1802	0.98	0.24	0.23	0.98	0.70	0.15
extracted	837	0.94	0.49	0.49	0.95	1.00	0.14
Δ modified	1562	0.01	0.10	0.10	0.00	0.00	0.01
Δ extracted	2527	0.04	0.35	0.36	-0.02	0.30	-0.00
XMLDocument (W)	476	1.01	0.27	0.28	0.48	0.59	0.07
modified	479	1.00	0.16	0.18	0.30	0.40	0.08
extracted	19	0.92	1.00	1.00	1.00	1.00	0.38
Δ modified	-3	0.01	-0.11	-0.10	-0.18	-0.19	0.01
Δ extracted	457	0.09	0.73	0.72	0.52	0.41	0.30

C.4 Dual clustering

C.4.1 Preferences

These are the preferences (see Section 2.4.2) in effect when the metrics were collected:

- Constructors, inherited members, inner class members, object methods were filtered out.
- Static members, loggers and loggers were included.
- Only required methods were condensed.

C.4.2 Non-cohesion metrics

Table C.10 contains the metric results for number of fields (NF), number of methods (NM), and weighted methods per class (WMC), where the WMC values weighted using computational complexity. The "Class" column contains entries for the *original* class (before it was refactored), the *modified* class (the larger class after refactoring), and the *extracted* class (the smaller class after refactoring). The Δ *modified* rows contain the improvement in the given measurement between the original and modified classes, and the Δ *extracted* rows contain the improvement in the given measurement between the original and extracted classes.

Table C.10: Dual clustering – non-cohesion metric results

Class	NF	NM	WMC
BRuleEngine (J)	9	21	57
modified	8	21	57
extracted	2	10	10
Δ modified	1	0	0
Δ extracted	7	11	47
CandidateURI (H)	16	56	84
modified	17	58	81
extracted	1	18	23
Δ modified	-1	-2	3
Δ extracted	15	38	61
CommandLine (J)	8	36	66
modified	8	36	59
extracted	1	16	23
Δ modified	0	0	7
Δ extracted	7	20	43
CrawlController (H)	63	100	247
modified	62	100	230
extracted	2	7	24
Δ modified	1	0	17
Δ extracted	61	93	223
DatabaseUtils (W)	36	47	194
modified	35	47	190
extracted	2	7	11
Δ modified	1	0	4
Δ extracted	34	40	183
FreeColClient (F)	28	48	94
modified	19	48	89
extracted	10	19	24
Δ modified	9	0	5
Δ extracted	18	29	70

Continued on next page

Table C.10 – *Continued from previous page*

Class	**NF**	**NM**	**WMC**
FreeColObject (F)	10	53	99
modified	10	54	95
extracted	3	15	29
Δ modified	0	-1	4
Δ extracted	7	38	70
FreeColServer (F)	26	52	181
modified	21	52	166
extracted	6	12	27
Δ modified	5	0	15
Δ extracted	20	40	154
Heritrix (H)	47	97	293
modified	47	97	291
extracted	2	11	13
Δ modified	0	0	2
Δ extracted	45	86	280
ImageLibrary (F)	10	62	109
modified	11	61	93
extracted	1	20	35
Δ modified	-1	1	16
Δ extracted	9	42	74
LPBRuleEngine (J)	10	26	47
modified	8	27	43
extracted	4	13	18
Δ modified	2	-1	4
Δ extracted	6	13	29
N3JenaWriterCommon (J)	23	46	136
modified	19	48	119
extracted	6	16	35
Δ modified	4	-2	17
Δ extracted	17	30	101

Continued on next page

Table C.10 – *Continued from previous page*

Class	NF	NM	WMC
Node (J)	11	45	51
modified	10	39	44
extracted	3	21	22
Δ modified	1	6	7
Δ extracted	8	24	29
Node (W)	15	35	63
modified	14	30	39
extracted	3	14	33
Δ modified	1	5	24
Δ extracted	12	21	30
ParserBase (J)	19	38	93
modified	17	34	44
extracted	3	16	61
Δ modified	2	4	49
Δ extracted	16	22	32
RegOptimizer (W)	21	22	48
modified	6	24	36
extracted	17	28	42
Δ modified	15	-2	12
Δ extracted	4	-6	6
ResultMatrix (W)	41	127	305
modified	40	127	303
extracted	2	9	11
Δ modified	1	0	2
Δ extracted	39	118	294
Rule (J)	7	43	109
modified	7	32	76
extracted	0	11	33
Δ modified	0	11	33
Δ extracted	7	32	76

Continued on next page

Table C.10 – *Continued from previous page*

Class	**NF**	**NM**	**WMC**
Script (W)	8	33	72
modified	8	35	72
extracted	2	7	9
Δ modified	0	-2	0
Δ extracted	6	26	63
SettingsHandler (H)	22	29	60
modified	22	29	48
extracted	2	16	28
Δ modified	0	0	12
Δ extracted	20	13	32
Specification (F)	47	87	155
modified	45	87	139
extracted	8	40	73
Δ modified	2	0	16
Δ extracted	39	47	82
XMLDocument (W)	24	39	61
modified	24	37	52
extracted	2	11	18
Δ modified	0	2	9
Δ extracted	22	28	43

C.4.3 Cohesion metrics

Table C.11 shows the values of six structural cohesion metrics – LCOM, LCOM*, TCC, DC_D, LCC, and DC_I, and the C3V semantic cohesion metric. The "Class" column contains entries for the *original* class (before it was refactored), the *modified* class (the larger class after refactoring), and the *extracted* class (the smaller class after refactoring). The Δ *modified* rows contain the improvement in the given measurement between the original and modified classes, and the Δ *extracted* rows contain the improvement in the given measurement between the original and extracted classes. Due to rounding, the improvement rows may sometimes appear to be off by 0.01.

Table C.11: Dual clustering – cohesion results

Class	LCOM	LCOM*	DCD	DCI	LCC	TCC	C3V
BRuleEngine (J)	85	0.88	0.26	0.26	0.59	0.59	0.10
modified	55	0.85	0.35	0.35	1.00	1.00	0.12
extracted	0	0.56	0.50	0.50	0.50	0.50	0.34
Δ modified	30	0.02	0.09	0.09	0.41	0.41	0.01
Δ extracted	85	0.32	0.24	0.24	-0.09	-0.09	0.23
CandidateURI (H)	1,153	0.98	0.13	0.15	0.25	0.33	0.06
modified	998	0.97	0.25	0.27	0.64	0.71	0.07
extracted	118	0.00	0.01	0.02	0.01	0.02	0.31
Δ modified	155	0.02	0.12	0.12	0.40	0.38	0.01
Δ extracted	1,035	0.98	-0.12	-0.13	-0.24	-0.32	0.26
CommandLine (J)	507	0.93	0.22	0.22	0.64	0.64	0.17
modified	327	0.90	0.25	0.25	0.68	0.68	0.14
extracted	75	0.00	0.87	0.87	0.87	0.87	0.53
Δ modified	180	0.03	0.02	0.02	0.05	0.05	-0.02
Δ extracted	432	0.93	0.65	0.65	0.23	0.23	0.36
CrawlController (H)	3,957	0.97	0.20	0.18	0.73	0.76	0.04
modified	3,947	0.97	0.20	0.18	0.73	0.76	0.04
extracted	0	0.20	0.67	0.67	0.67	0.67	0.63
Δ modified	10	0.00	0.00	0.00	0.00	0.00	-0.00
Δ extracted	3,957	0.77	0.47	0.49	-0.06	-0.09	0.59
DatabaseUtils (W)	641	0.98	0.26	0.26	0.55	0.53	0.07
modified	623	0.98	0.26	0.26	0.55	0.53	0.06
extracted	3	0.60	0.60	0.60	1.00	1.00	0.38
Δ modified	18	0.00	0.01	0.01	0.00	0.00	-0.01
Δ extracted	638	0.38	0.35	0.35	0.45	0.47	0.31
FreeColClient (F)	1,011	0.98	0.05	0.05	0.11	0.11	0.04
modified	749	0.95	0.18	0.18	0.23	0.23	0.06
extracted	133	0.94	0.11	0.11	0.22	0.22	0.13
Δ modified	262	0.02	0.13	0.13	0.13	0.13	0.02
Δ extracted	878	0.04	0.06	0.06	0.11	0.11	0.09

Continued on next page

Table C.11 – *Continued from previous page*

Class	LCOM	LCOM*	TCC	DCD	LCC	DCI	C3V
FreeColObject (F)	1,134	0.97	0.11	0.20	0.11	0.20	0.13
modified	1,368	1.00	0.12	0.20	0.12	0.20	0.14
extracted	0	0.72	1.00	1.00	1.00	1.00	0.39
Δ modified	-234	-0.03	0.01	0.01	0.01	0.01	0.01
Δ extracted	1,134	0.25	0.89	0.81	0.89	0.81	0.26
FreeColServer (F)	1,032	0.97	0.14	0.14	0.49	0.49	0.06
modified	914	0.95	0.20	0.20	0.69	0.69	0.06
extracted	49	0.95	0.09	0.09	0.13	0.13	0.16
Δ modified	118	0.02	0.06	0.06	0.20	0.20	0.01
Δ extracted	983	0.02	-0.05	-0.05	-0.36	-0.36	0.11
Heritrix (H)	4,044	0.99	0.20	0.29	0.55	0.77	0.06
modified	4,006	0.99	0.20	0.28	0.55	0.77	0.06
extracted	0	0.56	1.00	1.00	1.00	1.00	0.41
Δ modified	38	0.00	-0.01	-0.01	0.00	0.00	0.00
Δ extracted	4,044	0.43	0.80	0.71	0.45	0.23	0.35
ImageLibrary (F)	1,390	0.98	0.22	0.23	0.24	0.33	0.24
modified	1,063	0.95	0.26	0.28	0.28	0.35	0.23
extracted	171	0.00	0.02	0.05	0.02	0.05	0.28
Δ modified	327	0.03	0.04	0.04	0.04	0.02	-0.01
Δ extracted	1,219	0.98	-0.20	-0.19	-0.22	-0.28	0.04
LPBRuleEngine (J)	184	0.92	0.32	0.32	0.45	0.45	0.08
modified	130	0.89	0.30	0.30	0.65	0.65	0.10
extracted	20	0.82	0.41	0.41	0.46	0.46	0.20
Δ modified	54	0.04	-0.02	-0.02	0.19	0.19	0.02
Δ extracted	164	0.10	0.09	0.09	0.00	0.00	0.12
N3JenaWriterC... (J)	905	0.99	0.22	1.00	0.30	1.00	0.06
modified	816	0.97	0.28	1.00	0.52	1.00	0.09
extracted	103	1.02	0.07	0.33	0.07	0.33	0.08
Δ modified	89	0.02	0.06	0.00	0.22	0.00	0.03
Δ extracted	802	-0.04	-0.15	-0.67	-0.24	-0.67	0.02

Continued on next page

Table C.11 – *Continued from previous page*

Class	LCOM	LCOM*	TCC	DCD	LCC	DCI	C3V
Node (J)	660	1.00	0.08	0.08	0.08	0.08	0.09
modified	303	0.96	0.21	0.21	0.21	0.21	0.11
extracted	78	0.83	0.30	0.31	0.30	0.31	0.20
Δ modified	357	0.04	0.14	0.14	0.14	0.14	0.03
Δ extracted	582	0.18	0.22	0.24	0.22	0.24	0.11
Node (W)	533	0.98	0.12	0.12	0.15	0.15	0.17
modified	340	0.96	0.07	0.06	0.07	0.06	0.16
extracted	72	0.92	0.04	0.04	0.05	0.07	0.25
Δ modified	193	0.02	-0.05	-0.06	-0.08	-0.09	-0.01
Δ extracted	461	0.06	-0.08	-0.07	-0.10	-0.08	0.07
ParserBase (J)	624	1.00	0.06	0.00	0.08	0.00	0.08
modified	402	0.99	0.17	0.33	0.28	0.33	0.11
extracted	79	0.86	0.17	0.29	0.27	0.68	0.12
Δ modified	222	0.01	0.11	0.33	0.20	0.33	0.02
Δ extracted	545	0.15	0.11	0.29	0.19	0.68	0.04
RegOptimizer (W)	107	0.91	0.29	0.19	0.78	0.23	0.18
modified	106	0.92	0.33	0.31	0.38	0.38	0.09
extracted	203	0.88	0.21	0.06	1.00	0.16	0.28
Δ modified	1	-0.02	0.04	0.12	-0.41	0.15	-0.08
Δ extracted	-96	0.02	-0.08	-0.13	0.22	-0.07	0.10
ResultMatrix (W)	4,750	0.98	0.13	0.15	0.60	0.68	0.10
modified	4,742	0.98	0.13	0.15	0.60	0.68	0.10
extracted	0	0.43	0.71	0.71	1.00	1.00	0.55
Δ modified	8	0.00	0.00	0.00	0.00	0.00	-0.01
Δ extracted	4,750	0.55	0.58	0.57	0.41	0.32	0.45
Rule (J)	550	0.93	0.17	0.19	0.42	0.44	0.07
modified	245	0.90	0.32	0.32	0.77	0.77	0.09
extracted	0	0.00	0.00	0.33	0.00	0.40	0.18
Δ modified	305	0.03	0.15	0.13	0.36	0.33	0.02
Δ extracted	550	0.93	-0.17	0.14	-0.42	-0.04	0.12

Continued on next page

Table C.11 – *Continued from previous page*

Class	**LCOM**	**LCOM***	**TCC**	**DCD**	**LCC**	**DCI**	**C3V**
Script (W)	186	0.85	0.30	0.35	0.63	0.90	0.13
modified	219	0.86	0.28	0.30	0.66	0.60	0.12
extracted	1	0.60	0.73	0.70	1.00	1.00	0.43
Δ modified	-33	-0.00	-0.03	-0.05	0.03	-0.30	-0.01
Δ extracted	185	0.25	0.43	0.35	0.37	0.11	0.30
SettingsHandler (H)	187	0.99	0.32	0.49	0.36	0.58	0.08
modified	61	0.98	0.37	0.49	0.42	0.58	0.08
extracted	33	0.64	0.87	1.00	0.87	1.00	0.40
Δ modified	126	0.01	0.05	0.00	0.06	0.00	0.01
Δ extracted	154	0.35	0.55	0.51	0.50	0.43	0.32
Specification (F)	3,364	0.99	0.14	0.13	0.98	0.70	0.15
modified	2,966	0.98	0.18	0.18	0.95	0.70	0.15
extracted	695	0.95	0.51	0.51	1.00	1.00	0.14
Δ modified	398	0.00	0.05	0.05	-0.02	0.00	0.00
Δ extracted	2,669	0.03	0.37	0.38	0.03	0.30	0.23
XMLDocument (W)	476	1.01	0.27	0.28	0.48	0.59	0.07
modified	383	1.00	0.20	0.22	0.36	0.49	0.08
extracted	43	0.94	0.47	0.42	0.47	0.42	0.24
Δ modified	93	0.01	-0.07	-0.06	-0.12	-0.10	0.01
Δ extracted	433	0.06	0.20	0.14	-0.01	-0.18	0.17

Appendix D

List of Cohesion Metrics

Table D.1 contains a non-exhaustive list of object-oriented cohesion metrics that have been described in conference proceedings, journals, or web sites. Where possible, the table provides a metric's acronym, the long version of its name, and a reference to a paper in which it is described.

Much of the early work on object-oriented cohesion metrics was derived from Chidamber and Kemerer's LCOM. Unfortunately, some of this subsequent research did not provide new metric names, but instead referred to their metrics as extensions of LCOM. This has led to much confusion, especially when researchers compared multiple variations of "LCOM". These researchers often appended a number to "LCOM", .e.g., "LCOM2", to refer to a particular LCOM variant, and they did so inconsistently. The column titled "Alias" provides some of the known aliases for the LCOM variants, together with references to the users of that alias.

Table D.1: Object-oriented cohesion metrics

Acronym	Name	Alias	Ref.
	Cohesion of a Module		[AKC01]
C	Cohesion Among Methods of class	Co ([CKB00])	[HM95]
C3	Conceptual Cohesion of Classes		[MP05]
CAM	Cohesion Among Methods of class	CAMC	[BD02]

Continued on next page

Table D.1 – *Continued from previous page*

Acronym	Name	Alias	Ref.
CBMC	Cohesion Based on Member Connectivity		[CKB00]
CC(X)	Class Cohesion		[BK06]
ClassCoh	Class Cohesion		[GS08]
CLCOM5	Conceptual Lack of Cohesion on Methods		[UFPG10]
COC	Cohesion of Class		[KD08]
CSM	Conceptual Similarity between Methods		[MP05]
DC	Degree of Cohesion		[BBF95]
DC_D	Degree of Cohesion (direct)		[BB04]
DC_{DE}	Degree of Cohesion (direct extension)		[BBG08]
DC_I	Degree of Cohesion (indirect)		[BB04]
DC_{IE}	Degree of Cohesion (indirect extension)		[BBG08]
DMC	Dependence Matrix-based Cohesion		[WZW+05]
DRC	Dependence Relationship based Cohesion		[ZLLX04]
ICBMC	Improved CBMC		[ZXZY02]
ICH	Information flow-based cohesion		[LLWW95]
HLD	High Level Design		[ADB10b]
LCC	Loose Class Cohesion		[BK95]
LCC_{DE}	Lack of Cohesion in Class (direct extension)		[BBG08]

Continued on next page

Table D.1 – *Continued from previous page*

Acronym	Name	Alias	Ref.
LCC_{IE}	Lack of Cohesion in Class (indirect extension)		[BBG08]
LCIC	Lack of Coherence in Clients		[ML09]
LCOM	Lack of Cohesion in Methods	LCOM1 ([AD10]) LCOM2 ([BT07])	[CK91]
LCOM	Lack of Cohesion in Methods	LCOM1 ([BT07]), LCOM2 ([BDW97, AD10]), LCOM-CK ([LLL08])	[CK94]
LCOM	Lack of Cohesion in Methods	LCOM3 ([CEJ06, AD10, BT07])	[LH93]
LCOM	Lack of Cohesion in Methods	LCOM2 ([WZW+05]), LCOM3 ([BDW97]), LCOM4 ([CEJ06, AD10, BT07, EGF+04])	[HM95]
LCOM	Lack of Cohesion in Methods	LCOM* ([BDW98, FPn06, BT07, SB10]), LCOM3 ([WZW+05]), LCOM5 ([BDW97, AD10, CEJ06, BBG08, UFPG10]), LCOM-HS ([LLL08])	[Hen96]
LCSM	Lack of Conceptual Similarity between Methods		[MP05]
LORM	LOgical Relatedness of Methods		[ED00]
LSCC	LLD Similarity-based Class Cohesion		[ADB10a]
MWE	Maximal Weighted Entropy		[LPF+09]

Continued on next page

Table D.1 – *Continued from previous page*

Acronym	Name	Alias	Ref.
NHD	Normalized Hamming Distance		[CSC06]
NewCoh			[BDW99]
NRCI	Neutral Ratio of Cohesive Interactions		[BDW97]
OL_n			[ZXZY02]
ORCI	Optimistic Ratio of Cohesive Interactions		[BDW97]
PRCI	Pessimistic Ratio of Cohesive Interactions		[BDW97]
RCI	Ratio of Cohesive Interactions		[BDW97]
SBFC	Similarity-Based Functional Cohesion		[AD09]
SCC	Similarity-based Class Cohesion		[ADB10b]
SCFD	Semantic Closeness From Disambiguity		[CEJ06]
SCFD2	Semantic Closeness From Disambiguity		[CEJ06]
SCOM	Sensitive Class Cohesion Metric		[FPn06]
SSM	Structural Similarity between Methods		[DLOV08]
TCC	Tight Class Cohesion		[BK95]

Bibliography

[ACP$^+$06] M. Angioni, D. Carboni, S. Pinna, R. Sanna, N. Serra, and A. Soro. Integrating XP project management in development environments. *Journal of Systems Architecture*, 52(11):619–626, November 2006.

[AD09] J. Al Dallal. Software similarity-based functional cohesion metric. *Software, IET*, 3(1):45 – 57, February 2009.

[AD10] J. Al Dallal. Measuring the discriminative power of object-oriented class cohesion metrics. *IEEE Transactions on Software Engineering*, PP(99):1–1, November 2010.

[ADB10a] J. Al Dallal and L. Briand. A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2010.

[ADB10b] Jehad Al Dallal and Lionel C. Briand. An object-oriented high-level design-based class cohesion metric. *Information and Software Technology*, 52(12):1346–1361, December 2010.

[AEF08] Ahari Abadi, Ran Ettinger, and Yishai Feldman. Re-approaching the refactoring Rubicon. In *Proceedings of the Second ACM Workshop on Refactoring Tools*, page 10, Nashville, Tennessee, USA, October 2008.

[AFL99] Nicolas Anquetil, Cédric Fourrier, and Timothy C. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, page 235. IEEE Computer Society, 1999.

[AKC01] Edward B. Allen, Taghi M. Khoshgoftaar, and Ye Chen. Measuring coupling and cohesion of software modules: an information-theory

approach. In *Proceedings of the 7th International Symposium on Software Metrics*, page 124. IEEE Computer Society, 2001.

[Arn89] Robert S. Arnold. Software restructuring. *Proceedings of the IEEE*, 77(4):607–617, 1989.

[ASKM07] K. K Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. Investigating effect of design metrics on fault proneness in object-oriented systems. *Journal of Object Technology*, 6(10):127–141, December 2007.

[BB04] Linda Badri and Mourad Badri. A proposal of a new class cohesion criterion: an empirical study. *Journal of Object Technology*, 3(4):145–159, 2004.

[BBD01] L.C. Briand, C. Bunse, and J.W. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Transactions on Software Engineering*, 27(6):513–530, 2001.

[BBF95] Linda Badri, Mourad Badri, and S. Ferdenache. Towards quality control metrics for object-oriented systems analysis. In *TOOLS (Technology of Object-Oriented Languages and Systems)*, Versailles, France, March 1995. Prentice-Hall.

[BBG08] Linda Badri, Mourad Badri, and Alioune Gueye. Revisiting class cohesion: an empirical investigation on several systems. *Journal Of Object Technology*, 7(6), August 2008.

[BCM+96] Kent Beck, Ron Crocker, Gerard Meszaros, John Vlissides, James O Coplien, Lutz Dominick, and Frances Paulisch. Industrial experience with design patterns. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, page 103–114, Washington, DC, USA, 1996. IEEE Computer Society.

[BD02] Jagdish Bansiya and Carl Davis. A hierarchical model for object-oriented design quality assessment. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28(1):4–17, 2002.

[BDLMO10a] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. A two-step technique for extract class refactoring. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 151–154, Antwerp, Belgium, 2010. ACM.

[BDLMO10b] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. A two-step technique for extract class refactoring. Technical report TR10_01, University of Salerno, Salerno, Italy, 2010.

[BDLO11] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, March 2011.

[BDW97] L.C. Briand, J.W. Daly, and J. Wust. A unified framework for cohesion measurement in object-oriented systems. In *Proceedings of the Fourth International Software Metrics Symposium*, pages 43–53, 1997.

[BDW98] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998. 10.1023/A:1009783721306.

[BDW99] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[BEDL99] J. Bansiya, L. Etzkorn, C. Davis, and W. Li. A class cohesion metric for object-oriented designs. *Journal of Object Oriented Programming*, 11:47–52, 1999.

[BEGR00] Saida Benlarbi, Khaled El Emam, Nishith Goel, and Shesh Rai. Thresholds for object-oriented measures. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, page 24. IEEE Computer Society, 2000.

[Ber02] Pavel Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002. Accessed via http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.3739 on 22 Nov 2011.

[BH06] Alexander Budanitsky and Graeme Hirst. Evaluating WordNet-based measures of lexical semantic relatedness. *Computational Linguistics*, 32(1):13–47, 2006.

[BK95] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *SIGSOFT Software Eng. Notes, Proceedings of the 1995 Symposium on Software Reusability*, volume 20 of *SSR '95*, pages 259–262, Seattle, Washington, United States, 1995. ACM.

[BK06] Challa Bonja and Eyob Kidanmariam. Metrics for class cohesion and similarity between methods. In *Proceedings of the 44th Annual Southeast Regional Conference*, pages 91–95, Melbourne, Florida, 2006. ACM.

[BLL09] H. Barkmann, R. Lincke, and W. Lowe. Quantitative evaluation of software quality metrics in open-source projects. In *Proceedings of The 2009 IEEE International Workshop on Quantitative Evaluation of Large-scale Systems and Technologies (QuEST09)*, Bedford, UK, May 2009.

[Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 2008.

[BM06] Fabian Bannwart and Peter Müller. Changing programs correctly: Refactoring with specifications. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085, pages 492–507. Springer, Berlin, Heidelberg, 2006.

[BME+07] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Jim Conallen, Kelli A. Houston, and Bobbi J. Young. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, April 2007.

[BPP08] Stanislav Busygin, Oleg Prokopyev, and Panos M. Pardalos. Biclustering in data mining. *Computers & Operations Research*, 35(9):2964–2987, September 2008.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[BS91] Alexander Borgida and John F. Sowa. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, 1991.

[BS06] Joachim Baumeister and Dietmar Seipel. Verification and refactoring of ontologies with rules. In Steffen Staab and Vojtěch Svátek, editors, *Managing Knowledge in a World of Networks*, volume 4248, pages 82–95. Springer, Berlin, Heidelberg, 2006.

[BT07] Richard Barker and Ewan Tempero. A large-scale empirical comparison of object-oriented cohesion metrics. In *14th Asia-Pacific Software Engineering Conference*, pages 414–421, 2007.

[BWZ02] Lionel C. Briand, Jürgen Wüst, and Marvin V. Zelkowitz. Empirical studies of quality models in object-oriented systems. In *Advances in Computers*, volume 56, pages 97–166. Elsevier, 2002.

[CAG11] Keith Cassell, Peter Andreae, and Lindsay Groves. A dual clustering approach to the extract class refactoring. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering*, pages 77–82, Miami, FL, July 2011.

[CAGA11] Keith Cassell, Craig Anslow, Lindsay Groves, and Peter Andreae. Visualizing the refactoring of classes via clustering. In *Proc. Australasian Computer Science Conference (ACSC 2011)*, volume 113 of *CRPIT*, pages 63–72, Perth, Australia, January 2011. ACS.

[CAGN09] Keith Cassell, Peter Andreae, Lindsay Groves, and James Noble. Towards automating class-splitting using betweenness clustering. In *24th IEEE/ACM International Conference on Automated Software Engineering*, pages 595–599, Auckland, NZ, November 2009.

[CAGN10] Keith Cassell, Peter Andreae, Lindsay Groves, and James Noble. An initial test suite for automated extract class refactorings. Technical Report ECSTR 10-21, Victoria University of Wellington, Dept. ECS, Wellington, NZ, September 2010.

[CC08] I.G. Czibula and G. Czibula. Clustering based automatic refactorings identification. In *Symbolic and Numeric Algorithms for Scientific Computing, 2008. SYNASC '08. 10th International Symposium on*, pages 253–256, 2008.

[CCS10] Márcio Cornélio, Ana Cavalcanti, and Augusto Sampaio. Sound refactorings. *Science of Computer Programming*, 75(3):106–133, March 2010.

[CDK98] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer. Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639, 1998.

[CDPV07] Shi-Kuo Chang, Vincenzo Deufemia, Giuseppe Polese, and Mario Vacca. A logic framework to support database refactoring. In Roland Wagner, Norman Revell, and Günther Pernul, editors, *Database and Expert Systems Applications*, volume 4653, pages 509–518. Springer, Berlin, Heidelberg, 2007.

[CEJ06] Glenn W. Cox, Letha H. Etzkorn, and William E. Hughes Jr. Cohesion metric for object-oriented systems based on semantic closeness from disambiguity. *Applied Artificial Intelligence*, 20(5):419–436, 2006.

[Chi10] Chire. ClusterAnalysis mouse.svg. `http://upload.wikimedia.org/wikipedia/commons/thumb/0/09/ClusterAnalysis_Mouse.svg/2000px-ClusterAnalysis_Mouse.svg.png`, October 2010. Accessed 17 November 2011.

[CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *SIGPLAN Not.*, 26(11):197–211, 1991.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CKB00] Heung Seok Chae, Yong Rae Kwon, and Doo-Hwan Bae. A cohesion measure for object-oriented classes. *Software Practice and Experience*, 30(12):1405–1431, 2000.

[CLM06] Y. Crespo, C. Lopez, and R. Marticorena. Relative thresholds: Case study to incorporate metrics in the detection of bad smells. In *Proceedings of 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 109–118, Lugano, Switzerland, 2006.

[Cor03] J. R Cordy. Comprehending reality-practical barriers to industrial adoption of software maintenance automation. In *11th IEEE International Workshop on Program Comprehension*, page 196–205, 2003.

[cRGA08] Ilie Şavga, Michael Rudolf, Sebastian Götz, and Uwe Aßmann. Practical refactoring-based framework upgrade. In *Proceedings of the 7th international conference on generative programming and component engineering*, GPCE '08, page 171–180, New York, NY, USA, 2008. ACM.

[CS06] István-Gergely Czibula and Gabriela Serban. Improving systems design using a clustering approach. *IJCSNS International Journal of Computer Science and Network Security*, 6(12):40–49, 2006.

[CSC06] Steve Counsell, Stephen Swift, and Jason Crampton. The interpretation and utility of three cohesion metrics for object-oriented design. *ACM Transactions on Software Engineering and Methodology*, 15(2):123–149, 2006.

[CV07] Rudi Cilibrasi and Paul M. B Vitanyi. The Google similarity distance. *IEEE Trans. Knowledge and Data Engineering*, 19(3):370–383, March 2007.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Prentice Hall, 1st edition, June 1991.

[DBDV04] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring – improving coupling and cohesion of existing code. In *Proceedings 11th Working Conference on Reverse Engineering*, pages 144–151, 2004.

[DDF$^+$90] S. Deerwester, S. T Dumais, G. W Furnas, T. K Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[DJ03] Melis Dagpinar and Jens H. Jahnke. Predicting maintainability with object-oriented metrics - an empirical comparison. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 155. IEEE Computer Society, 2003.

[DJ06] Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, March 2006.

[DLOV08] A. De Lucia, R. Oliveto, and L. Vorraro. Using structural and semantic metrics to improve class cohesion. In *IEEE International Conference on Software Maintenance*, pages 27 – 36, Beijing, September 2008.

[DYM$^+$08] Jens Dietrich, Vyacheslav Yakovlev, Catherine McCartin, Graham Jenson, and Manfred Duchrow. Cluster analysis of Java dependency graphs. In *Proceedings of the 4th ACM Symposium on Software Visualization*, pages 91–94, Ammersee, Germany, 2008. ACM.

[EBGR01] Kalhed El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[ED00] Letha Etzkorn and Harry Delugach. Towards a semantic metrics suite for object-oriented design. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, page 71. IEEE Computer Society, 2000.

[EGF$^+$04] Letha H. Etzkorn, Sampson E. Gholston, Julie L. Fortune, Cara E. Stein, Dawn Utley, Phillip A. Farrington, and Glenn W. Cox. A comparison of cohesion metrics for object-oriented systems. *Information and Software Technology*, 46(10):677–687, August 2004.

[EKS94] Johann Eder, Gerti Kappel, and Michael Schrefl. Coupling and cohesion in object-oriented systems. Technical report, Institut fur Informatik, Univ. of Klagenfurt, 1994.

[Etz06] L. H. Etzkorn. Semantic metrics, conceptual metrics, and ontology metrics: An analysis of software quality using IR-based systems, potential applications and collaborations. In *Proc. Int. Conf. Software Maintenance*, 2006.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999.

[FF09] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. In Ira Gessel and Gian-Carlo Rota, editors, *Classic Papers in Combinatorics*, pages 243–248. Birkhäuser Boston, Boston, MA, 2009.

[FLE00] Fredrik Farnstrom, James Lewis, and Charles Elkan. Scalability for clustering algorithms revisited. *ACM SIGKDD Explorations Newsletter*, 2(1):51–57, June 2000.

[Fok10] Marios Fokaefs. *Identification and Application of Extract Class Refactorings in Object-Oriented Systems*. Master's thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, 2010.

[For10] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, February 2010.

[Fow02] Martin Fowler. Public versus published interfaces. *IEEE Software*, 19(2):18–19, 2002.

[FPn06] Luis Fernández and Rosalía Peña. A sensitive metric of class cohesion. `http://sci-gems.math.bas.bg:8080/jspui/handle/10525/730`, 2006. Accessed 2011-11-21.

[FR91] Thomas M. J Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, November 1991.

[fS01] ISO International Organization for Standardization. *ISO/IEC 9126-1:2001 - Software engineering – Product quality – Part 1: Quality model*. IEEE, Geneva, Switzerland, 2001.

[FTCS09] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiou, and Jorg Sander. Decomposing object-oriented class modules using an agglomerative clustering technique. In *IEEE International Conference on Software Maintenance*, pages 93–101, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[FTSC11] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. JDeodorant: identification and application of extract class refactorings. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, page 1037–1039, New York, NY, USA, 2011. ACM.

[FWE03] B.C.M Fung, K. Wang, and M. Ester. Hierarchical document clustering using frequent itemsets. In *Proceedings of the Third SIAM International Conference on Data Mining*, volume 3, page 59, 2003.

[GE07] Erich Gamma and Thomas Eggenschwiler. JHotDraw start page. `http://www.jhotdraw.org/`, 2007. Accessed 2011-10-03.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, November 1994.

[GN02] Michelle Girvan and M.E. Newman. Community structure in social and biological networks. *Proc Natl Acad Sci U S A*, 99(12), June 2002.

[GR69] J. C. Gower and G. J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 18(1):54–64, January 1969.

[Gro10] Qualitas Research Group. Qualitas Corpus version 20101126. qualitascorpus.com, November 2010.

[GS08] G. Gui and P. D Scott. New coupling and cohesion metrics for evaluation of software component reusability. In *The 9th*

International Conference for Young Computer Scientists, 2008. ICYCS 2008., page 1181–1186, Hunan, 2008.

[GTA10] Tony Gorschek, Ewan Tempero, and Lefteris Angelis. A large-scale empirical study of practitioners' use of object-oriented concepts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1, pages 115–124, Cape Town, South Africa, 2010. ACM.

[Han93] Mary Hanna. Maintenance burden begging for remedy. *Software Magazine*, 13(6):53–53, April 1993.

[Har75] John A. Hartigan. *Clustering Algorithms*. Wiley, 1975.

[HCXY07] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, August 2007.

[Hen96] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, Inc., 1996.

[HFH+09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[HM95] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proc. Int. Symposium on Applied Corporate Computing*, Monterrey, Mexico, 1995.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, page 1–12, New York, NY, USA, May 2000. ACM.

[HW79] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, January 1979.

[ISO06] IEEE ISO. *International Standard - ISO/IEC 14764 IEEE Std 14764-2006 - Software Engineering - Software Life Cycle Processes - Maintenance*. IEEE, 2 edition, September 2006.

[Jem08] Dmitry Jemerov. Implementing refactorings in IntelliJ IDEA. In *Proceedings of the Second ACM Workshop on Refactoring Tools*, pages 13:1–13:2, Nashville, Tennessee, USA, October 2008.

[JMF99] A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[JS10] David Jurgens and Keith Stevens. The S-Space package: An open source package for word space models. In *System Papers of the Association of Computational Linguistics*, pages 30–35, Uppsala, Sweden, July 2010.

[KD08] O. Kurubus and N. Duru. A novel approach about cohesion measurement for classes. In *23rd International Symposium on Computer and Information Sciences*, pages 1–6, Istanbul, Turkey, October 2008.

[KE00] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings 8th International Workshop on Program Comprehension*, pages 201–210, 2000.

[Ker05] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005.

[KKZ09] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 3:1:1–1:58, March 2009.

[Kle99] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[KPF95] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton. Towards a framework for software measurement validation. *IEEE Trans. Softw. Eng.*, 21(12):929–944, 1995.

[KS08] Hannes Kegel and Friedrich Steimann. Systematically refactoring inheritance to delegation in Java. In *Proceedings of the 30th International Conference on Software Engineering*, pages 431–440, Leipzig, Germany, 2008. ACM.

[Lak97] Arun Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, 36(3):211–231, March 1997.

[Lan11] LangPop.com. Programming language popularity. `http://www.langpop.com/`, February 2011. Accessed February 2011.

[Len95] Douglas B. Lenat. CYC: a large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, November 1995.

[LH93] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122, 1993.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall object-oriented series. PTR Prentice Hall, Englewood Cliffs, NJ, 1994.

[LLL08] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 131–142, Seattle, Washington, USA, 2008.

[LLWW95] Y.-S. Lee, B.-S. Liang, S.-F. Wu, and F.-J. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proc. International Conference of Software Quality*, Maribor, Slovenia, 1995.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., 2006.

[LPF+09] Yixun Liu, Denys Poshyvanyk, Rudolf Ferenc, Tibor Gyimothy, and Nikos Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *IEEE International Conference on Software Maintenance*, pages 233–242, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[Mar04] Andrian Marcus. Semantic driven program analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 469–473. IEEE Computer Society, 2004.

[Mar06] Cristina Marinescu. Identification of design roles for the assessment of design quality in enterprise applications. In *Proc. IEEE International Conference on Program Comprehension*, pages 169–180, 2006.

[Mar08] Robert Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2008.

[MB08a] Emerson Murphy-Hill and Andrew P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, 2008.

[MB08b] Emerson Murphy-Hill and Andrew P. Black. Seven habits of a highly effective smell detector. In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, pages 36–40, Atlanta, Georgia, 2008. ACM.

[McC76] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[MD85] Sanjay Mittal and Clive Dym. Knowledge acquisition from multiple experts. *AI Magazine*, 6(2):32–36, June 1985.

[MDM07] Marius Marin, Arie Van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37, 2007.

[MHVG08] Naouel Moha, Amine Hacene, Petko Valtchev, and Yann-Gaël Guéhéneuc. Refactorings of design defects using relational concept analysis. In *Formal Concept Analysis*, volume 4933/2008 of *Lecture Notes in Computer Science*, pages 289–304. Springer, Berlin / Heidelberg, 2008.

[Mil95] George A. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[ML06] Sami Mäkelä and Ville Leppänen. Observations on lack of cohesion metrics. In *International Conference on Computer Systems and Technologies*, University of Veliko Tarnovo, Bulgaria, June 2006.

[ML07] Sami Mäkelä and Ville Leppänen. External views on class cohesion. In *Proceedings of the 2007 International Conference on Computer Systems and Technologies*, pages 1–6, Bulgaria, 2007. ACM.

[ML09] Sami Mäkelä and Ville Leppänen. Client-based cohesion metrics for Java programs. *Sci. Comput. Program.*, 74(5-6):355–378, 2009.

[MM01] Brian S. Mitchell and Spiros Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *IEEE International Conference on Software Maintenance*, page 744, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[MM06] B.S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[MP05] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 133–142. IEEE Computer Society, 2005.

[MPF08] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 355–382. Springer-Verlag London, UK, 1998.

[MT04] T Mens and T Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):139, 126, 2004.

[MU90] Hausi A Müller and James S Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Proceedings of the Conference on Software Maintenance, 1990*, pages 12—19, San Diego, CA , USA, 1990.

[Mun05] M.J. Munro. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In *Proceedings of the 11th IEEE International Symposium on Software Metrics*, page 15, 2005.

[MW03] James Moody and Douglas R White. Structural cohesion and embeddedness: A hierarchical concept of social groups. *American Sociological Review*, 68:103—127, 2003.

[New01] M.E. Newman. Scientific collaboration networks. II. shortest paths, weighted networks, and centrality. *Phys Rev E Stat Nonlin Soft Matter Phys*, 64(1 Pt 2), July 2001.

[New10] M.E.J. Newman. *Networks: An Introduction*. Oxford University Press, USA, 1st edition, May 2010.

[OCS10] S. M Olbrich, D. S Cruzes, and D. I. K. Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *2010 IEEE International Conference on Software Maintenance (ICSM)*, page 1–10, 2010.

[OFWB03] Joshua O'Madadhain, Danyel Fisher, Scott White, and Yan-Biao Boey. The JUNG (Java Universal Network/Graph) framework. Technical Report UCI-ICS 03-17, School of Information and Computer Science, University of California, Irvine, 2003.

[OJ90] William Opdyke and Ralph Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, pages 145–160, 1990.

[oM08] University of Manchester. TONES ontology repository. `http://owl.cs.manchester.ac.uk/repository/browser`, 2008. Accessed 2011-12-15.

[Opd92] William Opdyke. *Refactoring: a program restructuring aid in designing object-oriented application frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, May 1992.

[PC09] Javier Pérez and Yania Crespo. Perspectives on automated correction of bad smells. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 99–108, Amsterdam, The Netherlands, 2009. ACM.

[PGN08] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Proceedings of the 4th ACM Symposium on Software Visuallization*, pages 77–86, Ammersee, Germany, 2008. ACM.

[PHL04] Lance Parsons, Ehtesham Haque, and Huan Liu. Subspace clustering for high dimensional data: a review. *SIGKDD Explor. Newsl.*, 6(1):90–105, 2004.

[PLM+09] Weifeng Pan, Bing Li, Yutao Ma, Jing Liu, and Yeyi Qin. Class structure refactoring of object-oriented softwares using community detection in dependency networks. *Frontiers of Computer Science in China*, 3(3):396–404, August 2009.

[PM06] Denys Poshyvanyk and Adrian Marcus. The conceptual coupling metrics for object-oriented systems. In *22nd IEEE International Conference on Software Maintenance, 2006. ICSM'06*, pages 469–478, 2006.

[PNA10] Robert Patrick, Gregory Nyberg, and Philip Aston. *Professional Oracle WebLogic Server*. John Wiley and Sons, December 2010.

[Pre97] Roger S Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, 4 edition, 1997.

[Pri57] R. C Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[RB07] Martin Rosvall and Carl T. Bergstrom. An information-theoretic framework for resolving community structure in complex networks. *Proceedings of the National Academy of Sciences*, 104(18):7327–7331, May 2007.

[RBL+90] James R. Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design*. Prentice-Hall, 1st edition, October 1990.

[Rum96] James Rumbaugh. *OMT Insights*. Cambridge University Press, 1996.

[SAR+07] Barry Smith, Michael Ashburner, Cornelius Rosse, Jonathan Bard, William Bug, Werner Ceusters, Louis J Goldberg, Karen Eilbeck, Amelia Ireland, Christopher J Mungall, Neocles Leontis, Philippe Rocca-Serra, Alan Ruttenberg, Susanna-Assunta Sansone, Richard H Scheuermann, Nigam Shah, Patricia L Whetzel, and Suzanna Lewis. The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. *Nature Biotechnology*, 25(11):1251–1255, November 2007.

[SB10] Frank Sauer and Guillaume Boissier. Eclipse metrics plugin continued. `http://metrics2.sourceforge.net/`, 2010. Accessed 2010-04-22.

[SC07] G. Serban and l.-G. Czibula. Restructuring software systems using clustering. In *Proceedings of The 22th International Symposium on Computer and Information Sciences*, pages 1–6, Ankara, Turkey, November 2007.

[SC08] Gabriela Serban and István-Gergely Czibula. Object-oriented software systems restructuring through clustering. In *Artificial Intelligence and Soft Computing - ICAISC 2008*, pages 693–704. Springer-Verlag, Berlin / Heidelberg, 2008.

[Sch07] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, August 2007.

[Sch08] Stephen Schaub. Eclipse corner article: Creating database web applications with Eclipse. `http://www.eclipse.org/articles/article.php?file=Article-EclipseDbWebapps/index.html`, January 2008. Accessed 2010-04-28.

[SDF+03] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java(TM) Developer's Guide to Eclipse*. Addison-Wesley Professional, May 2003.

[SDGP10] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk. TopicXP: exploring topics in source code using Latent Dirichlet Allocation. In *Proc. of 26th IEEE International Conference on Software Maintenance*, Timişoara, Romania, September 2010.

[SEM08] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge proposal: verification of refactorings. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, pages 67–72, Savannah, GA, USA, 2008. ACM.

[SFB06] Stephen Saunders, Duane K. Fields, and Eugene Belayev. *IntelliJ IDEA in Action*. Manning Publications, March 2006.

[SGM00] Alexander Strehl, Joydeep Ghosh, and Raymond Mooney. Impact of similarity measures on web-page clustering. In *Workshop on Artificial Intelligence for Web Search (AAAI 2000)*, pages 58—64. AAAI, 2000.

[Sht10] Mark Shtern. *Methods for evaluating, selecting and improving software clustering algorithms*. PhD thesis, York University, Toronto, Ontario, 2010.

[SK03] R. Subramanyam and M.S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.

[SM07] Friedrich Steimann and Philip Mayer. Type access analysis: Towards informed interface design. *Journal of Object Technology*, 6(9):147–164, October 2007.

[SMC79] Wayne Stevens, Glenford Myers, and Larry Constantine. *Structured Design*. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[Som96] Ian Sommerville. *Software Engineering*. International computer science series. Addison-Wesley Pub. Co, Wokingham, England, 5th edition, 1996.

[SS04] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with KABA. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 315–330, Vancouver, BC, Canada, 2004. ACM.

[SSB06] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1909–1916, Seattle, Washington, USA, 2006. ACM.

[SSL00] F. Simon, F. Steinbrückner, and C. Lewerentz. 3d-spring embedder for complete graphs. Technical Report 11/00, BTU, Inst. of Computer Science, September 2000.

[SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, page 30. IEEE Computer Society, 2001.

[ST98] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 99–110, Lake Buena Vista, Florida, United States, 1998. ACM.

[ST09] Mark Shtern and Vassilios Tzerpos. Refining clustering evaluation using structure indicators. In *IEEE International Conference on Software Maintenance*, pages 297–305, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[ST11] Mark Shtern and Vassilios Tzerpos. Evaluating software clustering using multiple simulated authoritative decompositions. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 353–361. IEEE, September 2011.

[Ste07] Friedrich Steimann. The infer type refactoring and its use for interface-based programming. *Journal Of Object Technology*, 6(2):67–89, 2007.

[Ste11] Friedrich Steimann. Constraint-based model refactoring. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, page 440–454. Springer, 2011.

[Sut95] Jeff Sutherland. Business objects in corporate information systems. *ACM Computing Surveys*, 27(2):274–276, 1995.

[SWY75] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, November 1975.

[TAD+10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas Corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, Sydney, NSW, December 2010.

[TC09] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, January 2009.

[TH99] V. Tzerpos and R.C. Holt. MoJo: a distance metric for software clusterings. In *Proceedings Sixth Working Conference on Reverse Engineering, 1999*, pages 187–193, 1999.

[TM05] Adrian Trifu and Radu Marinescu. Diagnosing design problems in object oriented systems. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005)*, pages 155–164, Pittsburgh, Pennsylvania, November 2005.

[TP10] Peter Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research*, 37(1):141–188, 2010.

[UFPG10] B. Újházi, R. Ferenc, D. Poshyvanyk, and T. Gyimóthy. New conceptual coupling and cohesion metrics for object-oriented systems. In *Proc. of 10th IEEE International Working Conference on Source Code Analysis and Manipulation*, Timisoara, Romania, September 2010.

[WFH11] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier Science & Technology, January 2011.

[Wid06] Tobias Widmer. Unleashing the power of refactoring. *Eclipse Magazine*, July 2006.

[Wig97] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 33. IEEE Computer Society, 1997.

[WL08] Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM Symposium on Software Visualization*, pages 155–164, Ammersee, Germany, 2008. ACM.

[WT04] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 194–203. IEEE Computer Society, 2004.

[WZW+05] Jianmin Wang, Yuming Zhou, Lijie Wen, Yujian Chen, Hongmin Lu, and Baowen Xu. DMC: a more precise cohesion measure for classes. *Information and Software Technology*, 47(3):167–180, March 2005.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1979.

[YL94] S.W.L. Yip and T. Lam. A software maintenance survey. In *Proceedings First Asia-Pacific Software Engineering Conference*, pages 70–79, 1994.

[ZGC08] Danfeng Zhang, Yao Guo, and Xiangqun Chen. Automated aspect recommendation through clustering-based fan-in analysis. In *23rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2008*, pages 278–287, 2008.

[ZHR+06] Sharon Zakhour, Scott Hommel, Jacob Royal, Isaac Rabinovitch, Tom Risser, and Mark Hoeber. *The Java Tutorial: A Short Course on the Basics, 4th Edition*. Prentice Hall, 4th edition, October 2006.

[ZLLX04] Yuming Zhou, Jiangtao Lu, Hongmin Lu, and Baowen Xu. A comparative study of graph theory-based class cohesion measures. *SIGSOFT Softw. Eng. Notes*, 29(2):13–13, 2004.

[ZXZY02] Yuming Zhou, Baowen Xu, Jianjun Zhao, and Hongji Yang. ICBMC: an improved cohesion measure for classes. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 44–53. IEEE Computer Society, 2002.