Visualizing the Refactoring of Classes via Clustering

Keith Cassell **Craig Anslow** **Lindsay Groves** **Peter Andreae**

School of Engineering and Computer Science
Victoria University of Wellington,
PO Box 600, Wellington 6140, New Zealand
Email: {kcassell,craig,lindsay,pondy}@ecs.vuw.ac.nz

Abstract

When developing object-oriented classes, it is difficult to determine how to best reallocate the members of large, complex classes to create smaller, more cohesive ones. Clustering techniques can provide guidance on how to solve this allocation problem; however, inappropriate use of clustering can result in a class structure that is less maintainable than the original. The ExtC Visualizer helps the programmer understand the class structure by visually emphasizing important features of the class's members and their interrelationships. More importantly, it helps users see how various clustering algorithms group the class's members. These insights help a programmer choose appropriate techniques for refactoring large classes.

Keywords: Software visualization, clustering, refactoring, graph, maintainability

1 Introduction

Code maintenance is expensive. Some studies (Yip & Lam 1994) indicate that over 65% of the cost of software is maintenance. We address a common maintenance problem in object-oriented systems - the presence of large, complex classes with many methods and attributes (a.k.a. *members*). This paper describes our research in visualizing how the members of large classes can be re-organized using clustering techniques. Using the outputs of the clustering process, programmers can refactor their large classes and improve their software.

Fowler (Fowler et al. 1999) defines *refactoring* as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior", and he identifies a large class as being one of the "bad smells" in software that indicate likely problems. Fowler recommends using the *Extract Class* refactoring to distribute the methods and attributes from the large class into appropriate new classes.

Our research is primarily concerned with determining how methods and attributes can be reallocated, so that class-oriented refactorings like Extract Class can be applied. As part of this, we want to help the programmer see the important characteristics of these class members and their inter-relationships, and based on these, how they can be recombined to form more smaller, more cohesive classes.

Many tools, including our ExtC[1] tool (*Ext*ract *C*lass), help programmers see potentially important characteristics of an object-oriented class through the use of color, shape, and size, as well as through the relationships between the members as depicted in graphs. While such displays are helpful, large classes tend to produce crowded displays that obscure the underlying structure. Moreover, some intraclass relationships are complex and involve many methods and attributes. What is needed is a display that emphasizes the most important relationships within the class, including complex ones.

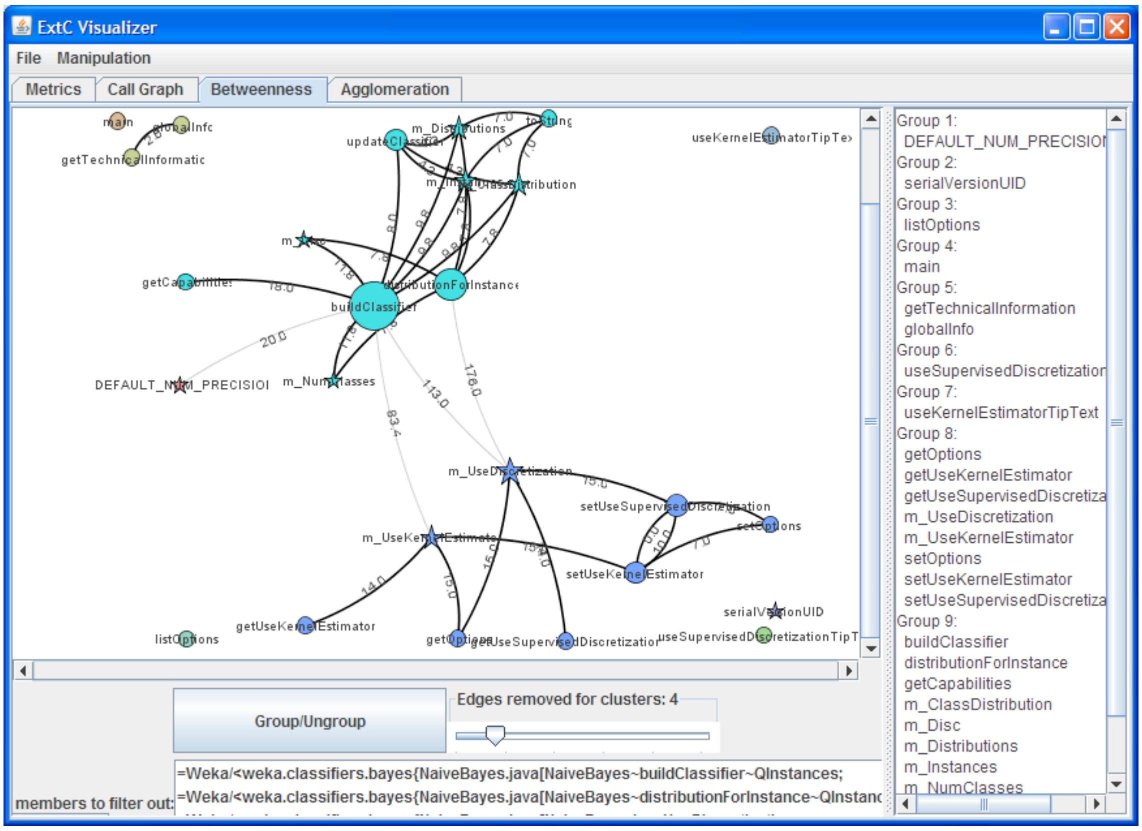
Clustering algorithms can help deal with the complexity. Clustering algorithms provide the ability for grouping things based on their characteristics, so we can use them to discover the underlying class structure that is critical for refactoring. However, there are many potentially useful clustering algorithms, and these can produce widely varying results depending on the algorithm chosen, how the algorithms are parameterized, and on the characteristics of the underlying data upon which the algorithms operate.

Some programmers may be content to see the outcomes of the clusterings and then use these as the bases of new classes via an Extract Class refactoring. Other programmers will prefer to see the clustering algorithm in action, either to see how closely the algorithm's functioning matches their intuition, or to see whether some intermediate results might suit them better as the basis for new classes.

Our ExtC tool can show particular incremental clustering algorithms in action (Figure 1). The user can control when these algorithms combine (or separate) members. This provides the ability to see the groups produced by the clustering algorithm. Moreover, because the user sees when members are combined (and to a limited extent, why they are combined), he can determine how much this matches his intuition.

The remainder of this paper is structured as follows. Section 2 provides some background. Section 3 covers the visualization features of the ExtC tool, while Section 4 discusses observations we made while using ExtC to analyze a number of open source software projects and the insights we gathered based on the visualizations. Section 5 discusses related work. The final section contains our conclusions and discusses potential future work.

[1] available from http://code.google.com/p/ext-c/



```
Group 1:
 DEFAULT_NUM_PRECISION
Group 2:
 serialVersionUID
Group 3:
 listOptions
Group 4:
 main
Group 5:
 getTechnicalInformation
 globalInfo
Group 6:
 useSupervisedDiscretization
Group 7:
 useKernelEstimatorTipText
Group 8:
 getOptions
 getUseKernelEstimator
 getUseSupervisedDiscretiza
 m_UseDiscretization
 m_UseKernelEstimator
 setOptions
 setUseKernelEstimator
 setUseSupervisedDiscretiza
Group 9:
 buildClassifier
 distributionForInstance
 getCapabilities
 m_ClassDistribution
 m_Disc
 m_Distributions
 m_Instances
 m_NumClasses
```

```
=Weka/<weka.classifiers.bayes{NaiveBayes.java[NaiveBayes~buildClassifier~QInstances;
=Weka/<weka.classifiers.bayes{NaiveBayes.java[NaiveBayes~distributionForInstance~QInstanc
```

Figure 1: The ExtC GUI

2 Background

This section gives a brief background of some software metrics that are relevant to clustering. It then discusses clustering and how it is relevant to refactoring. We discuss related work in visualization in Section 5.

2.1 Metrics

Software metrics are used to measure various aspects of software. We are particularly interested in measuring cohesion (how well software elements fit together), which has proven useful for identifying problematic software (Lanza & Marinescu 2006, Simon et al. 2001, Wettel & Lanza 2008, Tsantalis & Chatzigeorgiou 2009, Cassell et al. 2009) and for evaluating a modified software system relative to the original one. Furthermore, because one of our goals is improved cohesion for the new classes, an understanding of how these metrics work can help us devise algorithms to improve cohesion.

There are many such metrics described in the literature (Briand et al. 1997, Chae et al. 2000, Zhou et al. 2002, 2004). While many just consider the number of interactions between methods and attributes, some also consider the pattern of method and attribute interactions within a class. For example, the Cohesion Based on Member Connectivity (CBMC) metric (Chae et al. 2000) represents the access patterns between methods and attributes as a graph, and calculates the cohesiveness based on the number of nodes that need to be removed to fragment the graph. Closely related to CBMC is ICBMC (Improved CBMC) (Zhou et al. 2002), which fragments the graph by removing edges rather than nodes. Metrics such as these offer insights into the underlying structure of classes which can be exploited for determining how to extract new classes via clustering.

2.2 Clustering

Unsupervised clustering (Jain et al. 1999, Witten & Eibe 2001) is useful for identifying subsets of data that may represent coherent concepts. In the context of correcting poorly designed classes, the data to be clustered consists of the classes' members. The clustering algorithms use information about the affinity of the classes' members to group them. These groups can be used as the basis for forming cohesive classes, either in a large-scale class reorganization (Anquetil et al. 1999, Mitchell & Mancoridis 2006, Serban & Czibula 2008) or on a smaller scale, using the Extract Class refactoring (Cassell et al. 2009).

There are many different kinds of clustering techniques described in the literature. Berkhin (Berkhin 2002), 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 often produce different results with the same data set. (In fact, some algorithms are heuristic-based, so they may produce different results on different runs over the same data.) This section briefly reviews two categories of algorithms that we are investigating for use in refactoring - agglomerative and divisive clustering. Agglomerative clustering is a "bottom up" approach to clustering, while divisive clustering is "top down".

2.2.1 Agglomerative Clustering

Agglomerative clustering starts with seed entities and adds closely related entities to them until some stopping criterion is reached. The algorithms typically determine what constitutes a closely related entity using a distance function (or similarity function). Entities that are closest (most similar) are combined, distances are recalculated, and the process repeats.

The effective use of agglomerative clustering depends on important choices regarding the parameters to these algorithms:

1. Feature set - the characteristics of the entities to be evaluated

2. Distance function - a function that measures the distance between the entities based on their feature set

Part of this parameterization involves the representation of the clusters, i.e., how one defines the feature set of the cluster and how the distance function takes those into account when computing the distance between groups or between groups and individual entities.

For the purpose of restructuring classes, the entities to be clustered are generally attributes and methods, which are themselves dissimilar. This raises the issue of how one calculates the distance between an attribute and a method or between groups that are combinations of attributes and methods.

Several researchers have used a Jaccard similarity metric (Simon et al. 2001, Serban & Czibula 2008) for this. A Jaccard similarity metric calculates similarity by dividing the number of features two entities have in common by the number of featues total. As an example, one can assign slightly different feature sets to methods and attributes. The feature set for an attribute might include the attribute itself and the methods that access it, whereas the feature set for a method might include the method itself and the attributes it accesses. When a cluster is formed, its feature set becomes the merged features of its components.

We discuss how some other researchers have used this approach in Section 5, while Section 3.2.2 discusses our visualization of agglomerative clustering.

2.2.2 Divisive Clustering

Divisive clusterers work by splitting large groups into smaller ones. There are many divisive clustering techniques; this paper discusses *betweenness clustering*, which is a graph-based technique that has been applied to many domains (Girvan & Newman 2002), including object-oriented software (Dietrich et al. 2008, Cassell et al. 2009). One major difference between agglomerative and betweenness clustering is that the latter does not rely on similarity or distance functions that operate on the data.

Instead, betweenness clustering separates a connected graph into disconnected subgraphs by removing edges based on mathematical characteristics of the original graph. The subgraphs produced constitute the clusters. In a previous paper (Cassell et al. 2009), we discuss how we used betweenness clustering on the intraclass dependency graphs of some open source projects to recommend refactorings. We discuss betweenness clustering in the context of our visualizations in Section 3.2.2.

It is worth noting that betweenness clustering is similar in spirit to the ICBM technique (Zhou et al. 2002) for measuring cohesion, which relies on determining the cut sets for a graph. In fact, the creators of ICBMC mention that it could be used as a basis for class restructuring.

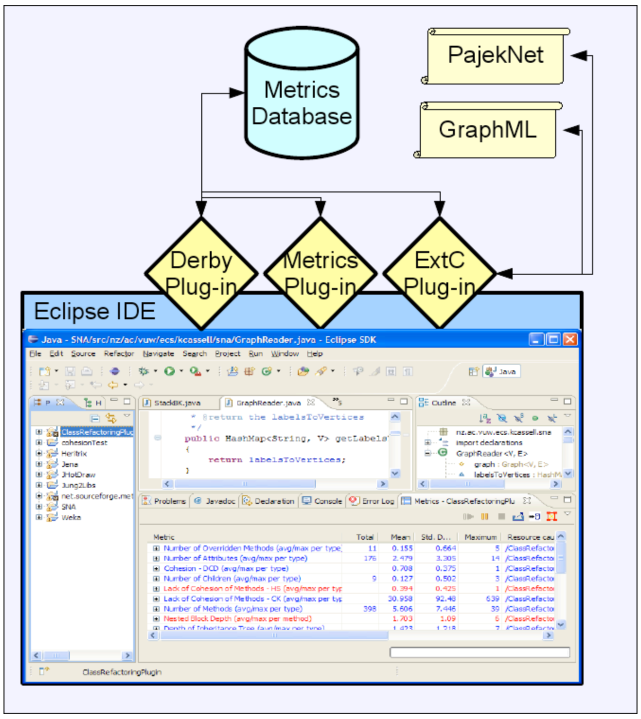


Figure 2: ExtC Architecture

3 ExtC Visualizer

3.1 Architecture

ExtC is designed to work as a plug-in in the Eclipse development environment (Shavor et al. 2003) as shown in Figure 2. In addition to providing an overall architecture via its plug-in oriented development framework, Eclipse provides extensive capabilities for code navigation and for programmatically processing Java code.

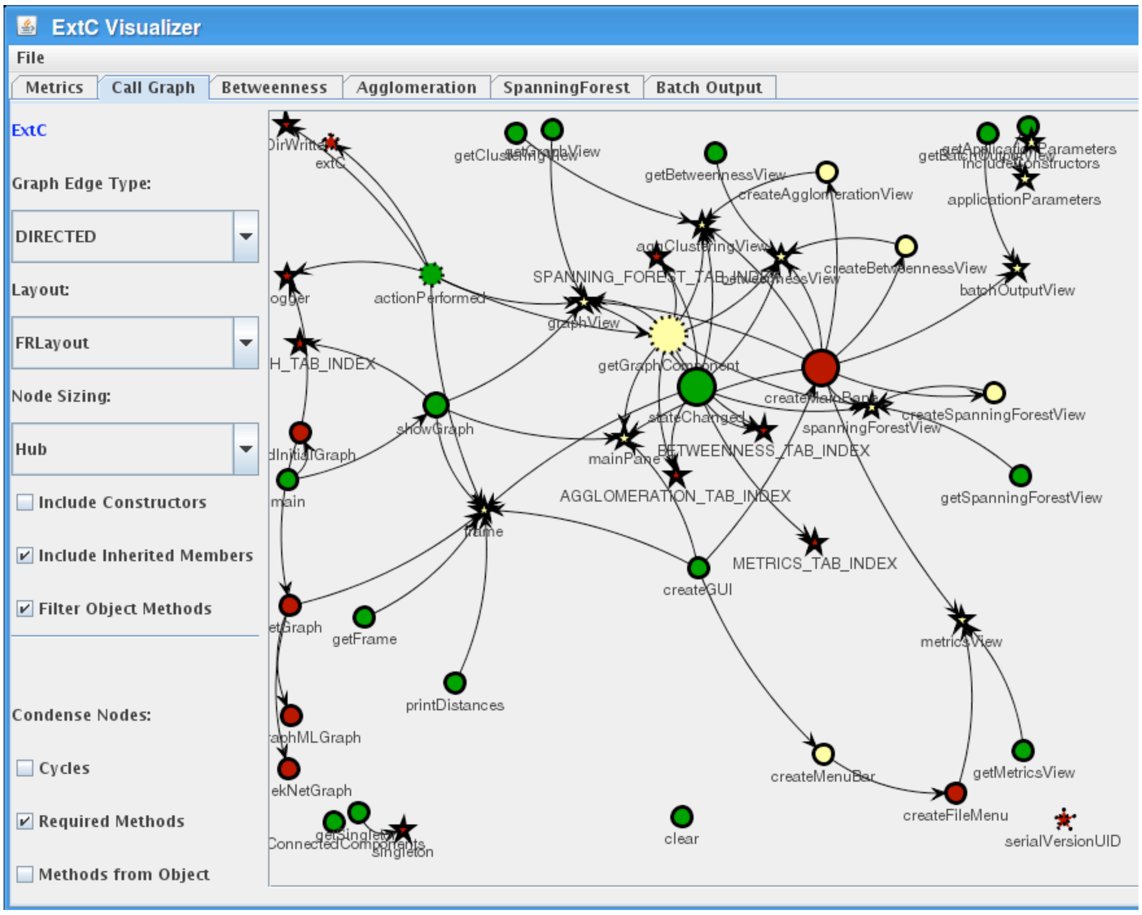
The Eclipse plug-in framework allows us to make use of third-party plug-ins. For example, we have enhanced the open-source *Eclipse Metrics2* plug-in (Sauer & Boissier 2010) to gather additional metrics and to store those metrics in a database. The Derby plug-in (Schaub 2008) provides ExtC access to that database.

ExtC uses the JUNG graph framework (O'Madadhain et al. 2003) for many graph-related tasks, including graph processing algorithms, layout, and manipulation. It further provides the capability of reading and writing graphs in either PajekNet or GraphML (Brandes & Pich 2005) format, so they can be read by other graphing packages.

Classes selected in the ExtC user interface have their intraclass dependency graphs shown in an ExtC graph display (Figure 3) while the corresponding code is loaded into Eclipse. In addition, the output of the clustering can be used as an input to an Extract Class refactoring tool, although this is currently a manual step.

3.2 Graphical User Interface

The ExtC GUI provides several views of the classes. The metrics view provides a tabular display of metric data pertaining to the classes of interest, while the dependency graph view (Figure 4) allows one to explore the relationships between a class's members. The agglomerative clustering view (Figure 5) and the betweenness clustering view (Figure 6) help the user



ExtC

Graph Edge Type: DIRECTED

Layout: FRLayout

Node Sizing: Hub

☐ Include Constructors

☑ Include Inherited Members

☑ Filter Object Methods

Condense Nodes:

☐ Cycles

☑ Required Methods

☐ Methods from Object

Figure 3: The ExtC Graph View

see how those clustering algorithms work on the underlying member data.

Its interactive graph display makes use of 2D graphics and color where the user can perform various manipulations (panning, scrolling, resizing, node movement, etc.). The clustering views have much the same capabilities, but also provide animations showing how the clustering algorithms work on the class's members. The animation capabilities will be discussed more thoroughly in Section 3.2.2.

3.2.1 Graph Display

The ExtC graph display (Figure 3) helps the user visualize the intraclass relationships between methods and attributes based on a static analysis of the code in a Java file. Each node represents either a method (circle), an attribute (star), or a group of members (triangle). Edges between nodes indicate either a method directly calling a method, or a method directly accessing an attribute. Colors indicate the allowable access to a member. For this figure, green indicates public, yellow indicates protected, and red indicates private; these colors are user configurable. The borders of the nodes indicate where the member is defined. A heavy border indicates that the member was defined in the class being examined, a lighter border indicates an inherited member, and a dashed border indicates a member defined in an inner class.

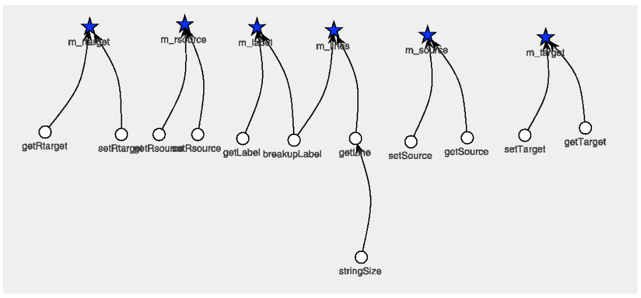
The user can alter the display of the graph as a whole by choosing any of several graph layout algorithms. Different layouts highlight different aspects of the graph structure. For example, Figure 4(a) shows how a shallow structure displayed within a directed acyclic graph (DAG) layout makes it easy to identify "data classes" that provide access to attributes but that have little logic. After the initial layout, the user can move nodes using the mouse or choose another layout to redisplay.

Our tool provides an interface where the user can choose to size nodes by various criteria, for example, by their out degree. (This can be useful in helping to identify large "brain methods" (Lanza & Marinescu 2006) that contain too much functionality.) Current bases for sizing include in-degree, out-degree, and hub and authority scores (Kleinberg 1999).

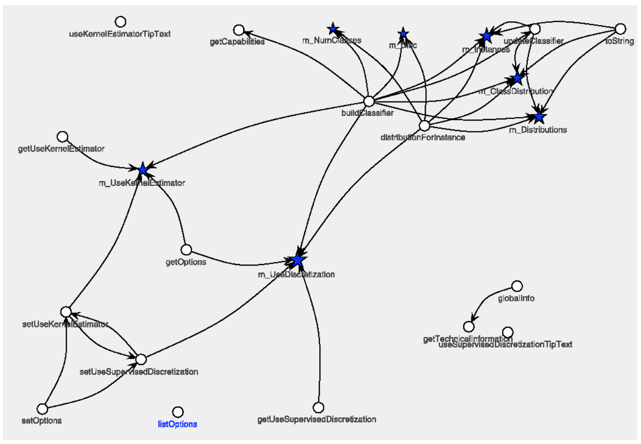
In addition to the options on how to display the nodes representing the class's members, the user also has options regarding whether to display certain members. For example, users may decide that methods inherited from Java's *Object* class (*toString*, *equals*, etc.) are not relevant to their analysis of the behavior of the class and may choose to exclude them. This reduces "noise" in the graph and makes it easier to see the relationships between the class members that provide the main functionality.

ExtC also provides an interface where the user can choose to "condense" nodes that should be considered as a group. Currently, there are two supported condensations (1) nodes representing methods involved in recursive cycles, and (2) nodes representing methods required by interfaces or superclasses.

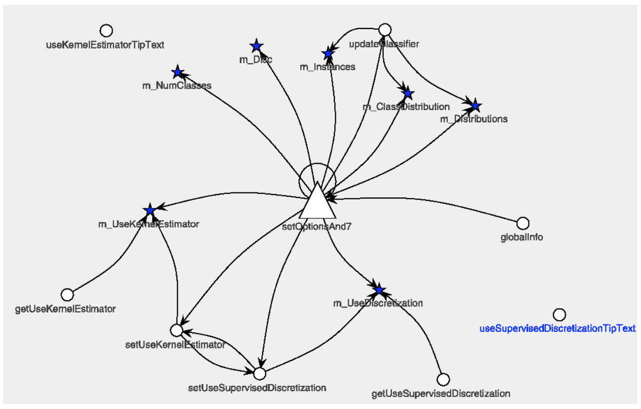
Interfaces and superclasses impose constraints on clustering, e.g. a single class must implement all of the methods specified in an interface. Consider Figure 4(b). This dependency graph looks like it could be split into two meaningful groups by eliminating three edges near the middle. However, if one condenses all



(a) DAG View



(b) Default View - No Condensation



(c) Default View - With Condensation

Figure 4: Dependency Graph Displays

methods imposed by interfaces and superclasses into a single triangular node, one gets the graph in Figure 4(c). This "hub and spoke" arrangement is less amenable to splitting via edge removal.

3.2.2 Clustering

The clustering views (Figures 5 and 6) have many of the same features as the graph view, including showing the dependency graph of a class. The primary difference between the graph view and the clustering views is the capability for animation. By manipulating a slider at the bottom of the screen, the user indicates the number of iterations of the clustering algorithm that he wants to execute. This may cause clusters of nodes to form on the screen. The exact graphical effect of moving the slider will depend on the clustering algorithm being used.

The visualizations for agglomerative and betweenness clustering are independent, but complementary. Because the algorithms operate in a different manner and can produce different results, seeing the algorithms in action helps the programmer choose those clustering results that best match his intuition and use these as the basis for forming new classes.

Agglomerative Clustering

Agglomerative clustering starts with individuals and merges them together into groups. Many clustering systems (Jain et al. 1999) show the results of agglomeration as dendrograms, which are tree-based structures. Each level in the tree indicates the merger of existing clusters of one or more elements. ExtC includes such a tree-based display. It also provides a display that shows the agglomeration algorithm acting on the software dependency graph. The graph is customized for our software task in that it shows the underlying dependencies between the class's members, but only shows the distances between linked nodes, rather than for all node pairs.

The display for the agglomerative clustering is similar to that of the graph view, with the following differences. Circular nodes represent unclustered members and are labeled with the member name. Clustered nodes are polygons. Clusters of only two members are represented as triangles 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 names of one of their members followed by one or more additonal special characters. The edges in the graph are labeled with the distances between the nodes.

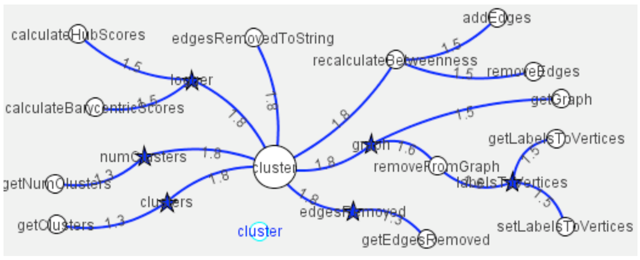
The user controls the clustering via a slider. Moving the slider to the right causes more more clusters to form, and the distance values on the edges to change.

Figure 5 shows agglomerative clustering in action. For this example, we have created a distance function that produces a small distance for the nodes that have few links except to each other. 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, a separate pane is simultaneously displaying a dendrogram-like tree that shows the hierarchical structure of the clusters formed thus far.

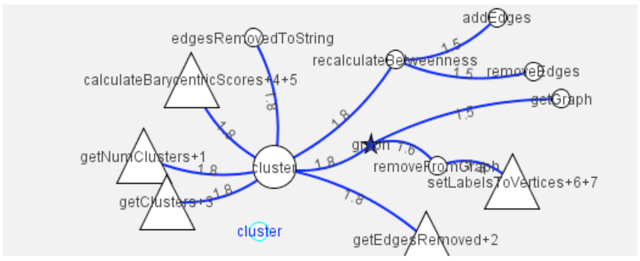
Figure 5(a) shows the dependency graph before the first aggregation step. The first seven iterations of aggregation are not shown. They are relatively uninteresting as the nodes on the outskirts of the graph are being clustered with their neighbors, and no cluster has more than three members. Figure 5(b) shows the graph after the seventh iteration. Figure 5(c) shows the graph after eight clustering iterations, when the first group of four is formed. Figure 5(d) shows the formation of a new group of two, and 5(e) shows the merger of that group with the group of four.

Betweenness Clustering

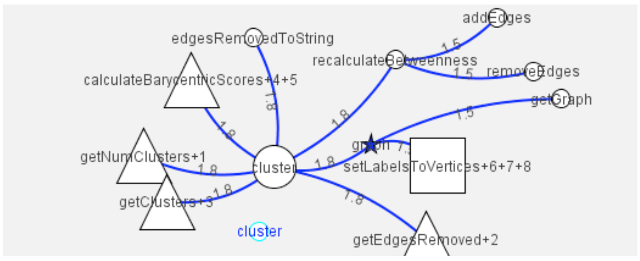
Betweenness clustering starts with a graph and removes edges to break the graph into disconnected parts, which are the clusters. The edges removed are those with the highest betweenness, where the betweenness value is the number of shortest paths between pairs of nodes that pass through that edge. If one considers a graph to represent information flow, where information passes through the edges, the high betweenness edges indicate where a graph can be cut to maximally disrupt information flow (or equiva-



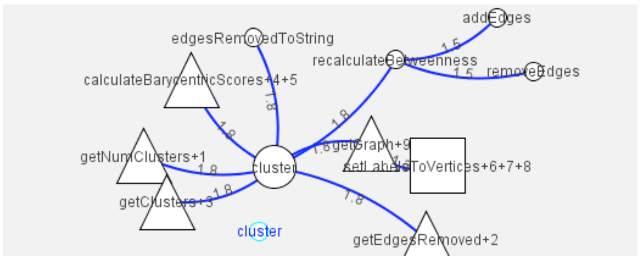
(a) Iteration 0



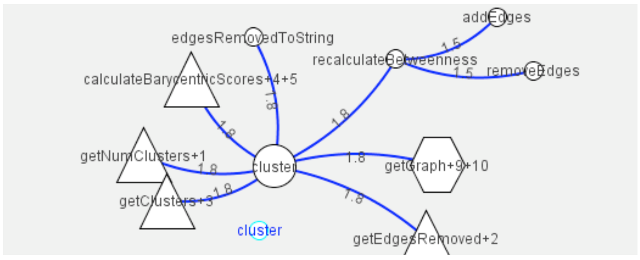
(b) Iteration 7



(c) Iteration 8

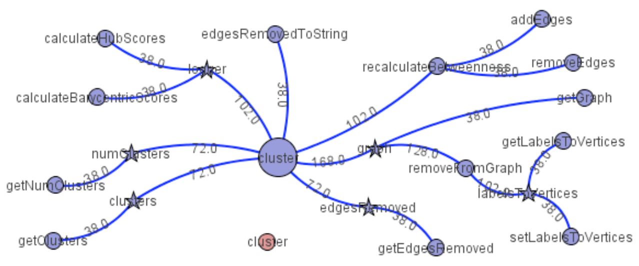


(d) Iteration 9

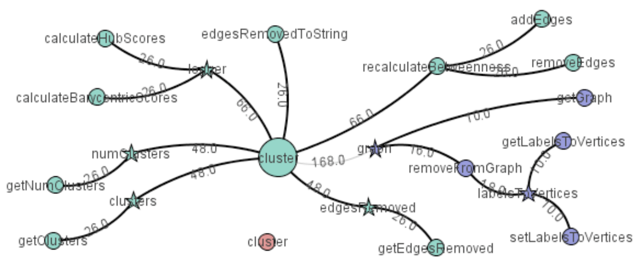


(e) Iteration 10

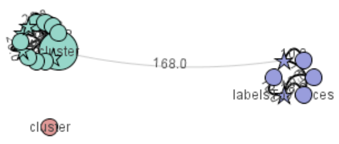
Figure 5: Agglomerative Clustering



(a) Unclustered View - Iteration 0



(b) Unclustered View - Iteration 1



(c) Clustered View - Iteration 1

Figure 6: Betweenness Clustering

lently, group together those nodes with highly shared information).

The display for the betweenness clustering is also similar to that of the graph view, and is based on a demo in JUNG (O'Madadhain et al. 2003). Circular nodes represent a class's members and are labeled with the member name. Clusters are represented by multiple nodes having the same color. Edges are labeled with their betweenness values.

The user controls edge removal via a slider. Moving the slider to the right causes more edges to be removed from the graph, more clusters to form, and the betweenness values on the edges to change.

Figure 6(a) shows a dependency graph before the first edge has been removed by betweenness clustering for the same class as in the prior example. In this example, the removal of a single edge (Figure 6(b)) is sufficient to form a new cluster of five nodes towards the bottom right of the graph. Each cluster has a different color.

The basic graph display can be fairly cluttered for large classes. However, by toggling the grouping feature, the user can cause each cluster of nodes to be put close together (Figure 6(c)). Each cluster still has distinct colors, but now its members are arranged in a tight circular layout. This focuses the user's attention on the groups and their interactions.

The most visible links now convey a different meaning. The links between the nodes of a group are generally obscured due to the tight packing, so the most visible links are the grey links between clusters. These indicate edges that have been removed to separate the groups and indicate locations where the classes to be created will be coupled.

4 Discussion

We developed ExtC primarily to help us better understand the characteristics of large Java classes and how they might be refactored into smaller, more cohesive classes. The classes we examined came primarily from four open source projects collected within the Qualitas Corpus (Heritrix, Jena, JHotDraw, and Weka) (Qualitas Research Group 2008), as well as our own ExtC project. Over 100 classes with at least 20 methods were examined using ExtC.

This section discusses observations we made while using the visualizations to analyze these classes. Based on these observations, we reached conclusions encompassing several broad areas:

1. Domain knowledge that should be considered by clustering algorithms to improve their ability to aid in refactoring large classes.
2. Insights into how the clustering algorithms worked
3. Possible improvements to the visualizations

The following subsections discuss these observations and conclusions, organized according to the view in which the observations were made. The subsection on the graph view incorporates most of the observations and conclusions about (1), while the subsections on agglomerative and betweenness clustering mostly cover (2) and (3). In general, for each observation we will name a single example of a class that showed the discussed behavior, although the behavior was typically shown by multiple classes.

4.1 Graph View

4.1.1 Cohesion and Clustering

Observation: In a DAG layout, many of the "leaves" of the graph were the expected star-shaped attributes; however, there were also fairly many circle-shaped methods. Upon investigating the corresponding code, some of these methods were shown to be "no-ops" or simple descriptors required by interfaces, while others made heavy use of other classes. Example: weka.gui.visualize.SVGTextAreaFigure (Weka)

Conclusion: Because most of the popular cohesion metrics concentrate on relationships between methods and attributes within a class, these leaf methods, and the associated calling methods, will cause misleadingly low cohesion scores. This may cause suboptimal results when clustering using cohesion measurements as part of the distance function or stopping criteria.

4.1.2 Data Classes

Observation: Some large, noncohesive classes are largely "data classes" composed of attributes and their accessors. Example: nz.ac.vuw.ecs.kcassell.callgraph.CallGraphNode (ExtC)

Conclusion: Because we are primarily interested in splitting up a class's logic, data classes should be not be considered. These should be detectable programmatically.

4.1.3 "Special" Members

Observation: Some of the nodes with the most links corresponded to methods that were not really part of the "business logic" of the class. These include nodes that represented "informational" methods (calls to loggers, *toString*, ...) and nodes that represented "generic" methods that consider most if not all of the fields (e.g. clone, equals, initialization methods), and probably cross-cutting concerns. Example: net.n3.nanoxml.XMLElement (JHotDraw)

Conclusion: To help clarify the fundamental logic of the class to the user, it is highly beneficial to remove nodes that are not part of the "business logic", but are connected to many other nodes. Such nodes may not have a large effect on the ultimate results, but they do decrease efficiency and introduce noise.

It would be nice to be able to automatically dispose of informational nodes, but this is difficult. For example, one could automatically eliminate *clone* from consideration. However, some of the code we have analyzed does most of its cloning work in a related method that clone calls. To adjust for the vagaries of highly connected nodes, we added a filter to the UI that enables the user to indicate nodes that should not be displayed.

Furthermore, many of these special members should not be moved exclusively to one of the split classes, but should themselves be split between the new classes. For example, each new class will likely want its own logging class, *toString* methods, etc.

4.1.4 Method Chains and Clustering Criteria

Observation: Many call graphs have "chains" of method calls, where each method in the chain calls only one other method from the class, terminating with a method acessing a single variable. Example: com.hp.hpl.jena.graph.impl.GraphBase (Jena)

Conclusion: If there is a member that is only connected to a single other member, those members should be clustered. An agglomerative clusterer's distance function should capture this idea.

4.2 Agglomerative Clustering

Based on our tentative conclusion that chains of methods should be clustered, we decided to run some agglomerative clustering experiments where the distance function was primarily based on the number of edges on the shortest path between nodes, with a fractional secondary distance being added based on the number of edges each node had. (The secondary distance was to ensure that nodes with many connections would be joined after those that were more exclusively linked.)

4.2.1 "Fast-forwarding" Needed

Observation: Many of the initial groupings are somewhat obvious, as the nodes with only one connection are combined with their neighbors. For large classes, this makes the beginning of the animation fairly uninteresting. Example: com.hp.hpl.jena.rdf.model.impl.StatementBase (Jena)

Conclusion: We should provide a user option for "fast-forwarding" past specified kinds of agglomerations. It may be useful to provide options for fast forwarding, e.g. until a group of a specified size is reached.

4.2.2 Distance Functions

Observation: Squares (indicating three clustering steps) were appearing on the graph while there were still singly connected circles (indicating an unclustered member). This indicates that when there are multiple chains of nodes in the original graph, a single chain might be involved in multiple clusterings before another chain is involved in any. Example: weka.gui.beans.CostBenefitAnalysis (Weka)

Conclusion: A distance function that only looks at the current state of the graph and ignores the original state may give counterintuitive results.

4.2.3 Handling of Deprecated Methods

Observation: Some of the first nodes to be agglomerated involved deprecated methods.

Conclusion: Deprecated methods need to be a special case for class refactoring. Presumably, they were deprecated, because they could not be safely removed from the class. Visually, these should be grouped with the other condensed nodes. Example: org.apache.commons.httpclient.HttpConnection (Heritrix)

4.2.4 Variable Results

Observation: Running the same algorithm on the same data multiple times can give varying results. This occurs when multiple edges each have the same (smallest) distance. Example: weka.classifiers.functions.RBFNetwork (Weka)

Conclusion: Nondeterminism is troubling. This could be eliminated in many cases by having a more precise distance function. On the other hand, when two distances are the same, or nearly so, the user might prefer seeing the alternative clusterings. This warrants further study.

4.3 Betweenness Clustering

4.3.1 Graph Density

Observation: It is difficult to see the relative densities of different areas of graphs that represent classes with hundreds of members. Example: com.hp.hpl.jena.rdf.model.impl ModelCom (Jena)

Conclusion: Betweenness clustering helps by highlighting the less dense areas of a graph when the high betweenness edges are removed and the edge colors are changed.

4.3.2 Edge Weight Recalculation

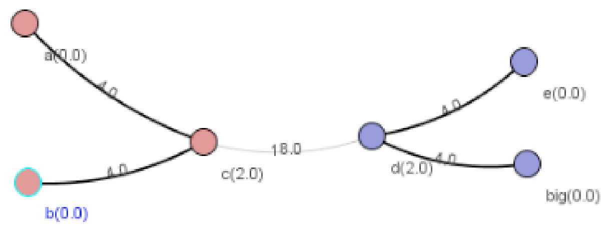
Observation: When one removes the edge with the highest betweenness value, there can be a drastic shift in edge weights when they are recalculated. This occurs when the removal of the edge disconnects two subgraphs. The nodes that were connected by the high betweenness edge tended to be central to the pre-split graph. After the graph is diconnected, they tend to be peripheral to the new subgraphs. Example: org.archive.crawler.datamodel.CrawlURI (Heritrix)

Conclusion: Attempts to be efficient by cutting down on betweenness recalculation may give faulty results.

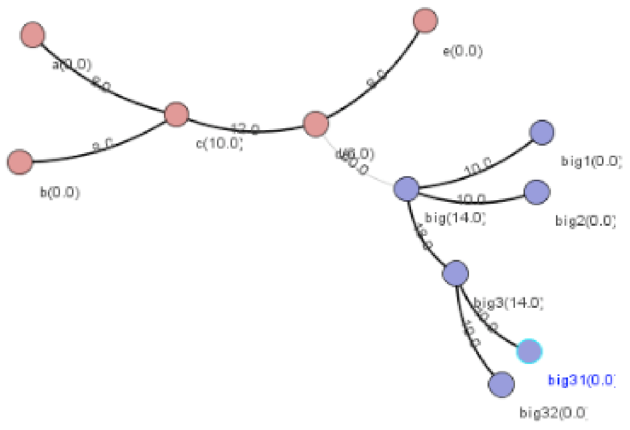
4.3.3 Node Weighting Based on Method Size

Observation: The betweenness clustering algorithm is sensitive to long chains of links. Consider the call graph shown in Figure 7(a). Suppose the method in the lower right is a large method that should be broken up into smaller, more maintainable pieces. After performing several Extract Method refactorings, one has a functionally equivalent class with the call graph in Figure 7(b). However, betweenness clustering now produces different groups.

Conclusion: This weakness might be addressed by node weighting, e.g. giving nodes representing large methods more weight than nodes representing small methods.



(a) Before Method Extraction



(b) After Method Extraction

Figure 7: Betweenness Clustering and Method Extraction

4.3.4 "Short cut" Edges

Observation: Some high betweenness edges are just "short cuts" between some nodes that may also be linked indirectly, therefore betweenness clustering may remove edges that were not part of the cut set. Example: org.jhotdraw.samples.svg.io.SVGOutputFormat (JHotDraw)

Conclusion: We had specified several a priori conditions meant to ensure the quality of the proposed new class structure. One of the heuristic criteria specified for being an acceptable group was that the number of nodes in the new group should be at least four times the number of edges removed. This criterion was intended to limit the amount of coupling introduced by extracting a new class. Unfortunately, it can result in false negatives when edges are removed from the graph that are not part of the cut set, because these edges do not indicate coupling between the newly created classes.

Our original criteria should be modified as follows. Instead of a simple 4x multiplier based on edges removed, the density of the proposed clusters should be compared with that of the subgraph from which it came. Alternatively, only the edges removed that were part of the cut set should be considered in the 4x calculation.

4.3.5 Betweenness and Directed Graphs

Observation: When the graph is directed, the highly weighted edges tend to indicate those method calls that have the largest call tree. Removing edges just serves to isolate the busiest methods. Example: com.hp.hpl.jena.n3.N3JenaWriterCommon (Jena)

Conclusion: Betweenness clustering on directed call graphs is not helpful for determining how to extract classes.

5 Related Work

There has been significant work regarding the identification and visualization of object-oriented software that needs refactoring (Lanza & Marinescu 2006, Wettel & Lanza 2008), but relatively little work on visualizing how these problematic classes could be improved. Contributions have come from several areas. In addition to work in visualization, there have been contributions from the fields of refactoring, machine learning and clustering, graph theory, metrics, and network analysis.

The work on visualizing how object-oriented software can be improved by refactoring has shown a steady progression over the years. A Virtual Reality Modelling Language (VRML) visualization (Simon et al. 2001) can be used to present distance information between class members in three dimensions, so programmers might see opportunities to move methods or attributes between classes or to perform Extract Class or Inline Class refactorings. This visualization shows the proximity of the classes' members based on a Jaccard similarity metric, but does not explicitly show the relationships between them.

Churcher and his colleagues (Churcher et al. 2003) capture those relationships. Their 3D graphs of classes include relationship information as links. They point out how various graph shapes show different degrees of cohesion. They further note the relationship between graphs indicating low cohesion and the possibilities for splitting a class, but they do not provide further guidance about how that split might be accomplished.

Noack's thesis (Noack 2007) does describe how a split could be done. While his primary emphasis is on determining how to lay out call graphs to emphasize the structure of software using clustering techniques, he also provides several techniques for splitting the graph to separate dense areas of the graph (similar to betweenness clustering).

In an approach somewhat similar to ours, a group of researchers at Massey University (Dietrich et al. 2008) use a variation of betweenness clustering to help identify opportunities for reorganizing software modules. Based on a specification of how many edges are to be removed, their tool suggests groupings of software components.

None of the previously mentioned tools showed clustering in action; however, there has been some activity in the network analysis community. For example, the JUNG graph framework (O'Madadhain et al. 2003) provides a clustering demo graphically illustrating how betweenness clustering works on social network data, and we used this as the basis of our betweenness clustering visualization of object-oriented software.

Most of the work above has involved some kind of graph-based visualization; however, it is worth mentioning some clustering work for the purposes of refactoring classes that is not graph-based and does not have a significant visual component. Several researchers have applied clustering techniques to the problem of remodularizing software systems (Anquetil et al. 1999, Mitchell & Mancoridis 2006), but only recently have these techniques been applied to refactoring classes. A Romanian team (Serban & Czibula 2008) created a system to enable experimentation with various ways of recombining attributes and methods into classes. In most of their experiments with agglomerative clustering, they used a Jaccard similarity metric. Unfortunately, it can be difficult to understand how these work without corresponding visualizations.

6 Conclusions

Our goal is to make object-oriented software easier to maintain by breaking large, noncohesive classes into smaller, more cohesive ones. Because programmers will make the final judgment about how classes will be organized, it is important that they see how a particular recommendation came about. ExtC helps by showing how clustering algorithms group (reallocate) methods and attributes for the formation of new classes.

We have provided visualizations for two major categories of clustering algorithms - agglomerative and divisive. Our experiences using ExtC thus far have inclined us to favor divisive clustering techniques, like betweenness clustering, over agglomerative techniques for two main reasons. First, divisive clustering matches up better with our mental model of the task of splitting classes. When refactoring, one wants to maintain the existing interface, so to be conservative, one generally wants to extract one new class at a time. This is consistent with making a single division of the graph. Secondly, divisive clustering generally requires fewer steps than agglomerative clustering for extracting classes. For a large class, watching the many steps required to reach two clusters can be extremely tedious.

ExtC has provided us some useful insights into our refactoring tasks, so we intend to enhance it. Our main effort over the next months is to expand the scope of ExtC beyond working on the dependency graph of a single class. Some of the information that can be useful for clustering involves how the studied class is used by other classes, and how it uses other classes. Our tool should display these relationships and distinguish them visually from the intraclass relationships.

We also want to expand our tool to be applicable for other class refactorings besides Extract Class. *Move Method*, *Extract Subclass*, and other refactorings have similar requirements to Extract Class in that they look for tight relationships (clusters) between certain members that are not well captured in the current class structure.

In addition to expanding the scope of what ExtC can do, we would like to make it more flexible. For example, it would be nice if a user could add some domain knowledge to help the clustering algorithms generate better results. Right now, the clustering algorithms work with little or no domain knowledge, i.e. the clustering doesn't know anything about software and how it should be structured. As software engineers we have clandestinely added some knowledge. For instance, the call graphs themselves represent certain relationships within the software. We have also added knowledge via our hard-coded distance functions; however, an arbitrarily complex user-defined (and domain aware) distance function could be used to give more precise results. The challenge is to determine how to let the user enter such knowledge for a distance function. Perhaps a software domain specific rule language can be constructed.

While there are enhancements to make, we feel that ExtC already provides capabilities that are useful to object-oriented programmers. There has been little prior work on applying clustering algorithms to refactoring large classes and, as far as we know, ExtC is the first tool that suggests how to refactor software by showing clustering in action.

7 Acknowledgments

We thank James Noble for his helpful review comments and Stuart Marshall for his assistance.

References

Anquetil, N., Fourrier, C. & Lethbridge, T. C. (1999), Experiments with clustering as a software remodularization method, *in* 'Proceedings of the Sixth Working Conference on Reverse Engineering', IEEE Computer Society, p. 235.

Berkhin, P. (2002), Survey of clustering data mining techniques, Technical report, Accrue Software.

Brandes, U. & Pich, C. (2005), GraphML transformation, *in* 'Graph Drawing', pp. 89–99.

Briand, L., Daly, J. & Wust, J. (1997), A unified framework for cohesion measurement in object-oriented systems, *in* 'Proceedings of the Fourth International Software Metrics Symposium, 1997.', pp. 43–53.

Cassell, K., Andreae, P., Groves, L. & Noble, J. (2009), Towards automating class-splitting using betweenness clustering, *in* '24th IEEE/ACM International Conference on Automated Software Engineering', Auckland, New Zealand, pp. 595–599.

Chae, H. S., Kwon, Y. R. & Bae, D.-H. (2000), 'A cohesion measure for object-oriented classes', *Software Practice and Experience* **30**(12), 1405–1431.

Churcher, N., Irwin, W. & Kriz, R. (2003), Visualising class cohesion with virtual worlds, *in* 'Proceedings of the Asia-Pacific Symposium on Information Visualisation (APVIS)', Australian Computer Society, Inc, pp. 89–97.

Dietrich, J., Yakovlev, V., McCartin, C., Jenson, G. & Duchrow, M. (2008), Cluster analysis of java dependency graphs, *in* 'Proceedings of the 4th ACM Symposium on Software Visualization', ACM, Ammersee, Germany, pp. 91–94.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, D. (1999), *Refactoring : Improving the Design of Existing Code*, Addison-Wesley, Boston.

Girvan, M. & Newman, M. (2002), 'Community structure in social and biological networks.', *Proc Natl Acad Sci U S A* **99**(12), 7826, 7821.

Jain, A., Murty, M. & Flynn, P. (1999), 'Data clustering: a review', *ACM Computing Surveys* **31**(3), 264–323.

Kleinberg, J. M. (1999), 'Authoritative sources in a hyperlinked environment', *J. ACM* **46**(5), 604–632.

Lanza, M. & Marinescu, R. (2006), *Object-Oriented Metrics in Practice*, Springer-Verlag New York, Inc.

Mitchell, B. & Mancoridis, S. (2006), 'On the automatic modularization of software systems using the bunch tool', *IEEE Transactions on Software Engineering* **32**(3), 193–208.

Noack, A. (2007), Unified quality measures for clusterings, layouts, and orderings of graphs, and their application as software design criteria, PhD thesis, Brandenburg University of Technology, Cottbus, Germany.

O'Madadhain, J., Fisher, D., White, S. & Boey, Y. (2003), The JUNG (Java universal Network/Graph) framework, Technical Report UCI-ICS 03-17, School of Information and Computer Science, University of California, Irvine.

Qualitas Research Group (2008), 'Qualitas corpus version 20080208'.
URL: *http://www.cs.auckland.ac.nz/ ewan/corpus*

Sauer, F. & Boissier, G. (2010), 'Eclipse metrics plugin continued', http://metrics2.sourceforge.net/.

Schaub, S. (2008), 'Eclipse corner article: Creating database web applications with eclipse', http://www.eclipse.org/articles/article.php?file= Article-EclipseDbWebapps/index.html.

Serban, G. & Czibula, I. (2008), Object-Oriented software systems restructuring through clustering, *in* 'Artificial Intelligence and Soft Computing - ICAISC 2008', Springer-Verlag, Berlin / Heidelberg, pp. 693–704.

Shavor, S., D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J. & McCarthy, P. (2003), *The Java(TM) Developer's Guide to Eclipse*, Addison-Wesley Professional.

Simon, F., Steinbrckner, F. & Lewerentz, C. (2001), Metrics based refactoring, *in* 'Proceedings of the Fifth European Conference on Software Maintenance and Reengineering', IEEE Computer Society, p. 30.

Tsantalis, N. & Chatzigeorgiou, A. (2009), 'Identification of move method refactoring opportunities', *IEEE Transactions on Software Engineering* **35**(3), 347–367.

Wettel, R. & Lanza, M. (2008), Visually localizing design problems with disharmony maps, *in* 'Proceedings of the 4th ACM Symposium on Software Visualization', ACM, Ammersee, Germany, pp. 155–164.

Witten, I. H. & Eibe, F. (2001), *Data Mining.*, Hanser Fachbuch.

Yip, S. & Lam, T. (1994), A software maintenance survey, *in* 'Proceedings First Asia-Pacific Software Engineering Conference', pp. 70–79.

Zhou, Y., Lu, J. & Xu, H. L. B. (2004), 'A comparative study of graph theory-based class cohesion measures', *SIGSOFT Softw. Eng. Notes* **29**(2), 13–13.

Zhou, Y., Xu, B., Zhao, J. & Yang, H. (2002), ICBMC: an improved cohesion measure for classes, *in* 'Proceedings of the International Conference on Software Maintenance (ICSM'02)', IEEE Computer Society, pp. 44–53.