Cluster Analysis of Java Dependency Graphs

Jens Dietrich[*], Vyacheslav Yakovlev[†], Catherine McCartin[‡], Graham Jenson[§]
School of Engineering and Advanced Technology, Massey University
Palmerston North, New Zealand

Manfred Duchrow[¶]
Consulting and Software
Laichingen, Germany

Abstract

We present a novel approach to the analysis of dependency graphs of object-oriented programs. We propose to use the Girvan-Newman clustering algorithm to compute the modular structure of programs. This is useful in assisting software engineers to redraw component boundaries in software, in order to improve the level of reuse and maintainability. The results of this analysis can be used as a starting point for refactoring the software. We present BARRIO, an Eclipse plugin that can detect and visualise clusters in dependency graphs extracted from Java programs by means of source code and byte code analysis. These clusters are then compared with the modular structure of the analysed programs defined by package and container specifications. Two metrics are introduced to measure the degree of overlap between the defined and the computed modular structure. Some empirical results obtained from analysing non-trivial software packages are presented.

CR Categories: D.2.8 [Metrics]: Product metrics—; D.2.7 [Distribution, Maintenance, and Enhancement]: Restructuring, reverse engineering, and reengineering—;

Keywords: dependency analysis, anti-pattern detection, refactoring, cluster analysis

1 Introduction

The analysis of the dependency graph of an object-oriented program is useful in assessing the quality of the design. The dependency graph can be extracted from a program using various methods, including source code and byte code analysis. The nodes in the dependency graph are types, while the edges represent relationships between those types. Once the graph has been extracted, it can be analysed in order to quantify certain characteristics of the respective program. Examples include the detection of circular dependencies, and measurement of the responsibility and independence of artefacts based on their relationships. Tools like JDepend[1] implementing these principles have become very popular in recent years.

In object-oriented design, special container artefacts are used to group types together. In Java, packages and jar files are used for this purpose. These structures can be used to built independent components that can be deployed separately. OSGi[2] and similar component technologies are based on a component model that consists of jar containers combined with private class loaders, private packages and extended meta data. The question arises of how existing, monolithic programs can be refactored into these component models.

We propose to analyse the dependency graph for clusters. The aim of this project is to build a tool that can produce a list of refactorings that can be used to transform programs into a more modular structure, one that is easier to customise and to maintain.

The paper is organized as follows: we introduce the algorithm used for clustering and the tool used to extract the dependency graph in Section 2. In Section 3 we discuss how clustering is performed. Visualisation of clusters is described in Section 4. We then introduce two metrics to measure the modularity of programs and present some empirical results in Section 6. Finally, we discuss related work and our plans for future work.

As part of the project presented here, we have implemented a tool called BARRIO that is available under the Apache Open Source License 2.0. The project site is http://code.google.com/p/barrio/. The tool is implemented as an Eclipse plugin, and can be installed directly from within Eclipse using the update mechanism.

2 Background

2.1 The Clustering Algorithm

The notion of a *cluster* or *community structure* in a graph was first proposed in the context of the social sciences. Given a graph $G = (V, E)$, a cluster is defined as a subgraph $G' = (V', E')$ whose nodes are tightly connected, i.e. cohesive. There are several ways in which the cohesion of a group of nodes can be quantified, we define clusters as groups of nodes within which connections are dense and between which connections are sparser. Girvan and Newman have proposed an iterative method for revealing clusters of this type in a graph, based on finding and removing progressively edges with highest *betweenness* [Girvan and Newman 2002], until the graph breaks up into components.

Betweenness is a centrality measure for edges in a graph. The betweenness of an edge is defined as the number of shortest paths between all pairs of nodes in the graph passing through that edge. Since the edges that lie between clusters are expected to be those with highest betweenness, we can find a good separation of the graph into clusters by removing them recursively. The general structure of the Girvan-Newman algorithm is as follows:

1. Calculate the betweenness value for each of the edges.
2. Remove the edge(s) having the highest value.
3. Repeat the analysis on the resulting graph until a suitable separation of the graph into clusters has been achieved.

2.2 Building the Dependency Graph

There are (at least) two different approaches to harvest the dependency graph from Java programs: byte code and source code analysis. Byte code analysis supports the analysis of programs for

[*]j.b.dietrich@massey.ac.nz
[†]slava.nz@gmail.com
[‡]C.M.McCartin@massey.ac.nz
[§]GrahamJenson@maori.geek.nz
[¶]manfred.duchrow@md-cs.de

[1]http://clarkware.com/software/JDepend.html
[2]Open Service Gateway Initiative - http://osgi.org

SOFTVIS 2008, Herrsching am Ammersee, Germany, September 16–17, 2008.
© 2008 ACM 978-1-60558-112-5/08/0009 $5.00

which source code is not available, such as third party commercial libraries. Unfortunately, byte code analysis cannot discover all relationships. In particular, the Java compiler erases type parameters information and resolved references to final static fields of primitive types (constants). Source code analysis obviously requires the availability of source code. There is information missing in source code as well - how the compiled code will be arranged in runtime containers (jar files). This information is usually defined in scripts executed at build time.

To some extent, the different dependency graphs that can be extracted with byte code and source code analysis reflect different aspects of dependency analysis related to design time and runtime modularity of software. Design time and runtime modules do often overlap, but not always. Therefore, dependency analysis from source code or byte code sometimes serves different purposes: modular organisation of source code to optimise development, and modular organisation of runtime artefacts to facilitate deployment and maintenance.

To decouple dependency analysis from a particular technique or tool used to extract the dependency graph we use the Object Dependency Exploration Model (ODEM)[3], a tool and platform independent XML vocabulary. BARRIO reads ODEM instances from streams in order to import dependency graphs for analysis. The Class Dependency Analyzer (CDA)[4] tool is used to extract ODEM instances from Java programs using byte code analysis. BARRIO has a built-in ODEM compliant source code analyser that is based on the Eclipse AST (Abstract Syntax Tree) API.

The dependency graph encoded in ODEM contains nodes representing classes. These nodes have annotations defining their classification (class, interface, annotation, etc), visibility, abstractness, and whether they are final. Every node also contains a list of one way relationships (dependencies) with the full name of the class (packageName.className) referenced and a dependency classification annotation (uses, extends or implements).

3 Clustering the Dependency Graph

For the clustering phase, we use the JUNG [O'Madadhain et al. 2003] implementation of the Girvan-Newman algorithm. Our tool gives the user the option of setting a separation level. Given a separation level S, the algorithm iterates through S cycles of betweenness value calculation and edge removal. At the start of each cycle betweenness values are calculated for the current state of the graph. All edges that have the maximum betweenness value are then removed from the graph.

In some cases, as edges are removed from the graph and new betweenness values are calculated, the maximum value can increase. This happens when there are alternative longer paths between pairs of nodes that are connected by shorter paths in earlier versions of the graph. The betweenness value of an edge is represented as a double rather than an integer. In cases where there are multiple shortest paths between a pair of nodes the count for that pair is distributed as a fraction over the edges in each of the paths, thus, the betweenness value for an edge is not necessarily an integer.

The annotations attached to nodes and edges can be used to define filters. Filters can be applied to edges and nodes. For instance,we can apply filters that allow us to reveal clusters in the subgraph whose edge set consists of only *uses* edges, or the subgraph whose node set consists of only *abstract class* nodes.

[3]http://pfsw.org/ODEM/schema/dtd/odem-1.1.dtd

[4]http://www.dependency-analyzer.org/

4 Visualisation

The Prefuse visualisation toolkit [Heer et al. 2005] is used to visualise the graph. Prefuse includes support for layouts and visual encoding techniques, interaction and animation. The nodes in the generated graph represent Java types. The nodes are displayed with labels that contain the name of the class, the namespace, the name of the container that the class belongs to, and an icon that reflects class properties. The relationships between classes are represented as directed edges with labels describing the type of relationship (*extends*, *implements* or *uses*).

Force directed layout is used to position visual elements of the graph [Fruchterman and Reingold 1991]. Force directed algorithms view the graph as a virtual physical system, where the nodes and edges of the graph are bodies of the system. The nodes have anti-gravitational or negative magnetic forces acting between them, edges act as springs between nodes. Unfortunately, the force directed layout and its animation become ineffective or unusable for large graphs (more than 1500 nodes). To overcome this problem force directed layout is replaced by a static[5] radial tree layout for graphs that contain over 1200 nodes.

To display groups of nodes, the visualisation uses elements called aggregates. An aggregate draws a border around a group of nodes that contain the same value for a particular property. Graph visualisation fills aggregates with colour with transparency. For example, Figure 1 shows the dependency graph built for the Apache Commons collections[6] library version 3.2. The aggregates show packages and clusters. In this example these structures do not overlap, indicating the presence of refactoring opportunities. In particular, there are some clusters on the left side of the figure that are part of the same package. Classes in these clusters could be safely moved into new packages and deployed in different jar files (runtime modules).

The graph visualisation provides the user with interactive controls such as pan, zoom and visual element drag. The aim of user interaction is to improve the layout of visual elements so that the display of the analysed system has a similar layout to the model of the system itself. On user request any visual node can be moved to any position on the screen. In cases where the user wishes to move a group of nodes the aggregate that represents that group must be visible. The visualisation of the graph allows the user to reposition elements of the graph in 2D space and to zoom in or out on parts of the graph. Highlighting occurs upon user action. Highlighting paints an edge and its end nodes with a darker colour and brings end nodes to the front.

These interaction controls allow the user to investigate the dependencies of the part of the system analysed, e.g. the package or dependency cluster. The graph visualisation offers smooth animated zoom and pan which helps the user to preserve a sense of position and context. This complies with Shneiderman's mantra regarding interaction techniques [Shneiderman 1996].

5 Measuring modularity

For the user looking for opportunities to refactor software, the following scenarios are of interest: clusters containing multiple packages (containers) may indicate opportunities to merge packages (containers), while packages (containers) containing multiple clusters may indicate that these packages (containers) could be split.

These opportunities can be measured by introducing two simple

[5]does not contain animation

[6]http://commons.apache.org/collections/

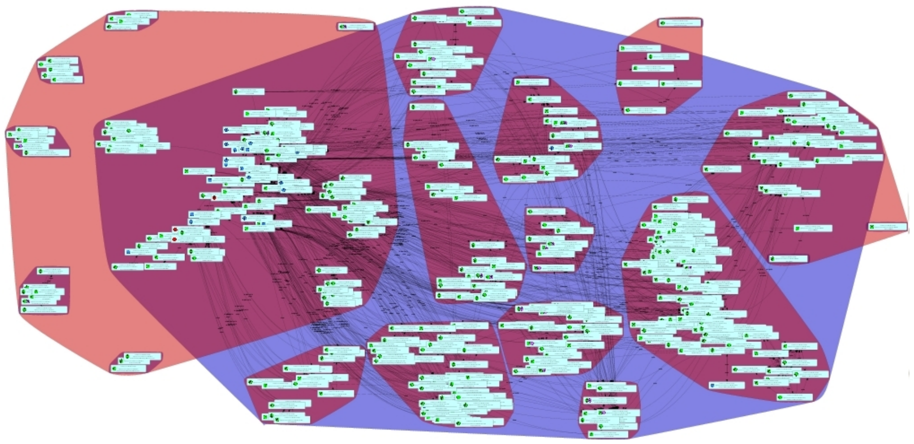

Figure 1: *Aggregates of packages and dependency clusters*

metrics: the average number of packages per cluster (APC) and the average number of clusters per package (ACP). Similar metrics can be introduced to measure the relationship between clusters and containers.

6 Empirical Results

To measure scaleability of the tool produced we have analysed a number of programs, including some popular open source programs such as Xerces, Xalan, Commons-collections, and the MySQL ConnectorJ JDBC driver. The largest program analysed was a product supplied by Kiwiplan, a New Zealand company providing software solutions for the packaging industry. The dependency graph for this project consists of 37 containers (jar files), 237 packages, 3116 classes and 13905 relationships. Running on a PC with a Intel Core 2 6600 @ 2.4 GHz processor and 2 GB of memory, calculating the connected components of the initial graph took only 25s. Cluster analysis with the separation level set to ten, i.e the clustering algorithm cycled through ten iterations, took 3min 10s.

In none of the programs analysed did increasing the separation level have a big impact. We interpret this as an indication that these programs already have a well defined modular structure. We have also analysed the dependencies of APC and ACP metrics on the separation level. Figure 2 shows the relationship between the separation level and ACP. Note that many of the singleton clusters appear even in the initial graph. These singleton clusters represent classes defining constants, that is, static, final fields. Due to the inline compilation for references to constants these field references are not visible in byte code.

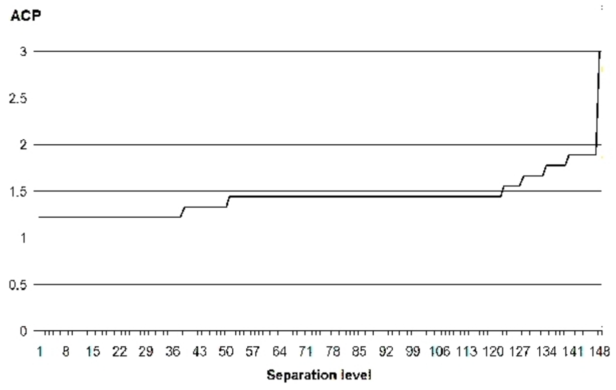

Figure 2: *ACP dependency on separation level*

Analysis of the chart (Figure 2) shows that a step occurs on the chart when the increased separation level causes cluster separation. A horizontal line indicates that while edges are removed the clusters remain connected. At separation level 148 all edges are removed from the graph, every package contains as many clusters as there are classes in that package.

7 Related work

A good overview of clustering techniques is given by B.S. Mitchell in his PhD thesis [Mitchell 2002]. Schwanke's Arch project [Schwanke 1991] is an approach to maximising the cohesion of procedures within the same module, while minimising the coupling between procedures in different modules. He also introduces maverick analysis - misplaced procedures are located and moved into new modules. In our approach we also locate relationships that reduce the modularity of programs so that the users can remove them using techniques like proxying or dependency injection. Müller's [Müller et al. 1993] and Anquetil and Lethbridge [Anquetil and Lethbridge 1999] use module names to detect clusters. Choi and Scacchi's [Choi and Scacchi 1990] approach is similar to our approach, they use articulation points to divide the dependency graph. Again, the difference is that our approach focuses on locating and breaking relationships. Mancoridis and Mitchell's tool Bunch [Mitchell and Mancoridis 2006] treats the software clustering as a search problem. Clusters are generated using formal concept analysis and genetic algorithms. Lattix [Sangal et al. 2005] is a tool for the analysis of dependencies in OO programs. The dependencies are presented using a dependency structure matrix (DSM). A variety of algorithms are available to help organise the matrix in a form that reflects the architecture and highlights patterns and problematic dependencies. A hierarchical structure, obtained in part by such algorithms and in part by input from the user, then becomes the basis for *design rules* that capture the architect's intent about which dependencies are acceptable. The design rules are applied repeatedly as the system evolves, to identify violations, and to keep the code and its architecture consistent.

8 Future work

While we were positively surprised about the scalability of our tool, we feel that more can be done to improve it, in particular by taking advantage of multi-threading and the availability of multi-core processors. Once the graph has been separated, clustering of the new components can be done in parallel.

We have already implemented several filters but have not yet combined filtering and clustering. For instance, edge filters could be used in order to separate abstraction layers, and we could look for clusters within these layers only. This could reveal modular structures within the specification or implementation layers which are not visible in the original graph.

The output of our tool is an XML report that can be further processed by applying stylesheets. In particular, by transforming the report into HTML it can be used by software engineers to plan refactoring. We plan to automate refactoring to some extent by making recommendations about dependency-breaking patterns that could be used in order to remove edges from the graph. This includes the use of dynamic linkage, dependency injection and proxies.

Future work is required to improve the visualisation of the graph in cases where large numbers of nodes are displayed on the screen at once. To make the visual graph more compact, super nodes [Auber et al. 2003] could be introduced. The concept of a super node is that one node contains many child nodes. For example, a node could represent a package, on user request it could be opened to display class nodes that the package contains. This technique will improve the visualisation, making the appearance of the graph less complicated. To improve the graph layout, force directed layout could be applied between aggregates if they are visible (the visibility of aggregates is an optional selection by the user).

Acknowledgements

The authors would like to thank Gareth Cronin from Kiwiplan for inspiring this project and providing us with test data. This work has been supported in part by Technology Fellowship Funding from the New Zealand Foundation for Research, Science and Technology (FRST), and through the Software Product and Project Improvement project also funded by FRST.

References

ANQUETIL, N., AND LETHBRIDGE, T. C. 1999. Recovering software architecture from the names of source files. *Journal of Software Maintenance 11*, 3, 201–221.

AUBER, D., CHIRICOTA, Y., JOURDAN, F., AND MELANCON, G. 2003. Multiscale visualization of small world networks. In *IEEE Symposition on Information Visualisation*, IEEE Computer Society, 75–81.

CHOI, S. C., AND SCACCHI, W. 1990. Extracting and restructuring the design of large systems. *IEEE Software 07*, 1, 66–71.

FRUCHTERMAN, T. M. J., AND REINGOLD, E. M. 1991. Graph drawing by force-directed placement. *Software: Practice and Experience 21*, 11, 1129–1164.

GIRVAN, M., AND NEWMAN, M. E. 2002. Community structure in social and biological networks. *Proc Natl Acad Sci U S A 99*, 12 (June), 7821–7826.

HEER, J., CARD, S. K., AND LANDAY, J. A. 2005. Prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceeding of the SIGCHI conference on Human factors in computing systems*, ACM Press, New York, NY, USA, 421–430.

MITCHELL, B. S., AND MANCORIDIS, S. 2006. On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng. 32*, 3, 193–208.

MITCHELL, B. S. 2002. *A heuristic search approach to solving the software clustering problem*. PhD thesis, Philadelphia, PA, USA. Adviser-Spiros Mancoridis.

MÜLLER, H., ORGUN, M., TILLEY, S., AND UHL, J. 1993. A reverse-engineering approach to subsystem structure identication. *Journal of Software Maintenance: Research and Practice 5*, 181–204.

O'MADADHAIN, J., FISHER, D., WHITE, S., AND BOEY, Y.-B. 2003. The jung (java universal network/graph) framework. Tech. Rep. UCI-ICS 03-17, University of California, Irvine.

SANGAL, N., JORDAN, E., SINHA, V., AND JACKSON, D. 2005. Using dependency models to manage software architecture. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM, New York, NY, USA, 164–165.

SCHWANKE, R. W. 1991. An intelligent tool for re-engineering software modularity. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, 83–92.

SHNEIDERMAN, B. 1996. The eyes have it: A task by data type taxonomy for information visualizations. In *VL '96: Proceedings of the 1996 IEEE Symposium on Visual Languages*, IEEE Computer Society, Washington, DC, USA, 336.