Cluster Analysis of Object-Oriented Programs

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Science

In

Computer Science

at Massey University, Palmerston North,

New Zealand.

Vyacheslav Yakovlev

2009

Abstract

In this thesis we present a novel approach to the analysis of dependency graphs of object-oriented programs, and we describe a tool that has been implemented for this purpose. A graph-theoretical clustering algorithm is used in order to compute the modular structure of programs. This can be used to assist software engineers to redraw component boundaries in software in order to improve the level of reuse and maintainability.

The analysis of the dependency graph of an object-oriented program is useful for assessing the quality of software design. The dependency graph can be extracted from a program using various different methods, including source code, byte code, and dynamic (behavioral) analysis. The nodes in the dependency graph are classes, members, packages and other artifacts, while the edges represent uses and extends relationships between those artifacts. Once the dependency 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 measurements of the responsibility or independence of units based on their relationships. Tools like JDepend[1] implementing these principles have become very popular in recent years.

Our work includes grouping types in dependency graphs using different clustering methods:

- Grouping into namespaces

[1] http://clarkware.com/software/JDepend.html

- Grouping into clusters using graph clustering algorithms
- Grouping into clusters using rules

The detected mismatches are candidates for refactoring.

We have developed a tool for processing dependency graphs clustering and producing results where users can outline possible design violations.

Acknowledgements

The very first thanks must be to the project supervisors Jens Dietrich and Catherine McCartin. Their input and guidance were essential for the success of this project.

We thank Kiwiplan Ltd for their involvement in the project. Kiwiplan is an innovative software development company specialising in providing a wide range of software solutions for the corrugating and packaging industry. Participation of Kiwiplan's developers brought commercial software development expertise and mentoring into the research work.

Special thanks belong to Manfred Duchrow from "Consulting & Software" in Laichingen, Germany. His Class Dependency Analyser (CDA) tool was particularly helpful for our software analysis. We thank Manfred Duchrow for presenting our research work in the software visualisation conference in September 2008.

We acknowledge Technology Fellowship (TIF) New Zealand along with Kiwiplan Ltd for sponsorship of our project and providing funding support that made this work happen.

We would like to thank Ewan Tempero from Auckland University for collecting, managing and providing to us a collection of open source Java software applications. The name of this collection is "Qualitas Corpus". This allowed us to have solid base for our analysis and evaluation of the results.

Contents

List of Figures

List of Tables

Chapter 1

Motivation

Component based software is concerned with implementation of software components which are assembled into larger pieces of software. Underlying this process is the notion that software components are developed in such a way that they provide functions common to many different systems. The idea was inspired by hardware components and was first proposed at the NATO Software engineering conference in 1968 by Douglas Mcllroy [25] and followed up by Clemens Alden Szyperski [42]. The goal of component based development is reusing software, making it cheaper and improving software quality by allowing parts (components) of a software system to be replaced by newer, functionally equivalent, components.

The purposes of component based development [33] could be described by two statements:

- A component based development should make it easier to fulfill the needs of many different applications
- A component based development should make it easier to adapt a given application to changing requirements

If any of these statements are present in the system implementation requirements to a sufficient degree, it may be worth developing a framework, or

investing in the use and possible adaptation of an existing framework.

A software component is described as a unit of composition. The purpose of a component is to be assembled with other components. A software component can be deployed independently and is subject to composition by third parties [42]. Assembly of the set of the collaborating components represent a component based software system. In order for components to collaborate, each component must provide or follow a contract. The way of describing contract between components varies between implementations of the frameworks. For example Eclipse Plugin Framework (EPF) introduces extension points that provide a contract between the component and its environment and defines which services the component provides and therefore defines its responsibility. The extension points and extensions act very similar to a electrical outlet, where the extension points act as power sockets and extensions act as power plugs.

Benefits of component driven development:

- **Reuse.** A component developed for reuse forms part of a framework of components that are intended to be used together. Modular architecture is made of components that can be combined in many ways to suit different needs.

- **Easier update.** Swapping a component for a newer version can be as easy as replacing a file in the folder. The system may remain running (software variation of the hot swap of the components).

- **Easier maintenance.** Similar to update, the existing modules can be maintained while the system is running.

- **System customisation.** Allows to supply only parts of the system to the customer to fill the requirements. Unnecessary functionality is not sold, but could be provided in the future if needed.

Risks [33] that should be considered:

- Developers and the company have to spend some time and effort learning a framework before the benefits can be realized.
- High costs of developing new frameworks
- Component-oriented development commits to long term goals. Some smaller scale projects with strict deadlines and short-term goals might conflict with the ideas of component engineering.

These points are considered as risks by many software engineering practices which discourage component-oriented development. Such practices focus on the individual application rather than viewing it as part of a much broader software process. As a result, many monolithic systems have been installed. Some of these are older systems that have been distributed before component based development gained its popularity, others are the result of inexperienced developers and/or unwillingness of companies to implement component based architectures. This motivates us to research the possibility of an automated process of restructuring complete software systems into components without changing a system's functionality. Much work has been done in this field using various methods, techniques and algorithms. We propose dependency graph clustering and rule based clustering to identify parts of a system that are potential candidates as components.

To achieve separation of a system into sensible components, we analyse software architecture at three different layers, or levels of granularity.

At the first level, we look at software architecture that represent grouping classes into namespaces and containers.

At the second level, we represent the software system as a graph, with nodes representing classes and edges representing relationships between the classes. Classes are grouped into clusters using an algorithm originating from the study of social networks. This algorithm attempts to form clusters by detecting the most tightly connected subgraphs in the original graph.

At the third level, we propose to group classes into clusters by applying a set of rules. Rules are created by the user and based on class dependencies. This is an assignment of the stereotypes[1] to the classes. The stereotypes of the class depend on the dependant types. For example class references any class in the package "javax.swing" receives "user interface" stereotype.

Our overall aim is to detect differences between "corresponding" clusters in each of the different layers. This information can then be used to guide the restructuring of the software architecture into components that relate to clusters that sensibly line up across all three layers.

[1]stereotype is a created additional category for the class, based on some class properties

Chapter 2

Anti-patterns, smells and refactoring

2.1 Introduction

Every software system lifecycle contain a stage of construction, and in the later stages it is maintained and updated. The design quality of the system varies depending on the skill set of the development team. The system's design quality is relative to how many **anti-patterns** the system contains.

In software engineering, an **anti-pattern** is a working solution to a problem that has negative consequences [16]. In other words some repeated pattern of action, process or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results.

The other term for possible bad practice in software development is a **smell**. A code smell is a surface indication that usually corresponds to a deeper problem in the system [9]. When a smell is detected in the software code or structure it is commonly said that the system smells. Smells do not always indicate problems, for example long methods are smells, but some long methods are fine [9].

The most common technique for removing anti-patterns is **refactoring**.

Refactoring involves changing a system's architecture while the functionality of the system itself remains unchanged [9]. Refactoring does not change functionality as it does not add or remove features of the software system, but it alters non-functional features like maintainability.

2.2 Anti-patterns

Knowledge of the anti-patterns is crucial to a system's success. Once anti-patterns are recognised they can be eliminated from the system. Frequently occurring anti-patterns should be well documented. Documentation of anti-patterns helps in their recognition and reduces their repetition. This way, the community of software developers can learn how others have eliminated bad designs. Some commonly occurring anti-patterns are described in [36].

Anti-patterns may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context. Some anti-patterns appear from the processes inherent in the development of information technology [18].

Depending on the area where anti-patterns cause their negative impact they can be sorted by type to reflect this, for example:

- **Software Development Anti-patterns.** A key goal of development anti-patterns is to describe useful forms of software refactoring. Software refactoring (section 2.4 has a description of the process) is a form of code modification, used to improve the software structure in support of subsequent extension and long-term maintenance. In most cases, the goal is to transform code without impacting correctness.

- **Software Architecture anti-patterns.** Architecture anti-patterns focus on the system-level and enterprise-level structure of applications and components. Although the engineering discipline of software architecture

is relatively immature, what has been determined repeatedly by software research and experience is the overarching importance of architecture in software development.

- **And many others** Software Project Management anti-patterns, Organizational anti-patterns, Testing anti-patterns, Quality assurance anti-patterns.

Documented anti-patterns do help software engineers to gain experience in recognising commonly reoccurring problems in the software industry. They outline common problems in software development and so to help establish techniques to recognise these problems and determine their underlying causes. Documentation of anti-patterns assists developers in creating a plan for reversing these underlying causes and implementing more productive solutions.

Enterprise applications usually contain a number of common design mistakes. They consistently occur and cause undesirable results. In some cases the same mistakes can often be found across various parts of the application. A large number of well known and commonly occurring problems have been documented as software anti-patterns [5] [35]. Similar to software design patterns [11], which describe better practices in software development, anti-patterns describe not only bad practices but also contain the corresponding solution to the problem in their description.

Trevor Parsons [35] concentrates on the run-time anti-patterns where the system's design can be defined as an instantiation (or instantiations) of a systems design decisions which have been made during its development A design model captures structural and behavioural information from an executing system. It contains information on the components that are executed, as well as the dependencies and patterns of communication between components that occur at run-time. Using advanced analysis techniques he summarises the run-time data and identifies relationships and patterns that might suggest potential anti-patterns in the system. The information extracted from the monitoring data can be represented in a run-time design model of the system. This model

is loaded into a rule engine or knowledge base which, (using predefined rules) can identify potential (well known) performance anti-patterns that exist in the system. Any detected anti-patterns are subsequently presented to the user along with specific data on the anti-pattern instance. This approach takes the burden away from developers having to sift through large volumes of data, in order to understand issues in their systems, and instead automates this process.

Trevor Parsons work [35] might appear to be somewhat removed from our topic, as his analysis of the program happens at run-time and is based on the object relationships, while we analyse programs using source code and bytecode and base our analysis on class dependencies. However, his methods of detection of the anti-patterns are similar to the research presented in this thesis. Detection can be achieved through the use of a rule engine. Rules are defined to match a number of class properties. If a rule matches it fires. When the rule fires, some defined tasks can be performed.

Naouel Moha and Yann-Gael Gueheneuc [30] define Software Architectural Defects (SAD) as anti-patterns and present tools and techniques for detection of SAD objects:

- OptimalAdvisor[1] is a static code analysis and refactoring tool. OptimalAdvisor can analyze source files, class files or a combination of the two. OptimalAdvisor's automated expertise can be requested about code fragments (methods, classes) or complete projects. Core features include:
  - Package and code analysis - visualization of the application
  - Code Analysis Rules validation
  - Refactoring: Dependency Inversion
- IBM Structural Analysis tool for Java (SA4J)[2] analyses structural dependencies of Java applications in order to measure their stability. It

[1]http://frontline.compuware.com/javacentral/tools/26222.asp
[2]http://www.alphaworks.ibm.com/tech/sa4j

detects structural anti-patterns (suspicious design elements) and provides dependency web browsing for detailed exploration of anti-patterns in the dependency web. SA4J also enables "what if" analysis in order to assess the impact of change on the functionality of the application; and it offers guidelines for package re-factoring.

Another interesting tool called SmallLint[3] automatically checks for over 60 common types of bugs at code level, such as identification of long methods.

Since Java supported multithreading it has become a popular programming language for creating applications that run multiple threads. But multithreading in Java is quite difficult not only for beginners but also for experienced programmers. S. Boroday's article [3] presents techniques for detecting anti-patterns in Java code that contain running multiple threads.

2.3 Smells

Some architectural smells can often be identified via the dependency graphs [13] of programs. Most common smells are singleton classes that are not referenced by any other class and do not reference any other classes. Not only a singleton class but also a whole isolated dependency cluster of several classes may be a smell.

Tree like structures in dependency graphs are also considered as smells, one class used by only one other class. This indicates that functional decomposition of the system and software reuse does not happen.

Dependency cycles in the graph are commonly occurring architecture smells. Cycles between namespaces will have negative effect, especially if one of the namespaces contain user interface classes.

Smells related to problems with namespaces are also common. Common smells are unused namespaces, dependency cycles, namespaces that are too big or too small, badly named namespaces.

[3] http://st-www.cs.uiuc.edu/users/brant/Refactory/Lint.html

2.4 Refactoring

Refactoring [9] is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.

A simple refactoring example is shown the figure 2.1. The system contains two packages with several classes in each of them. By moving "Class F" from one package to another a better modular structure is achieved and circular dependencies between packages are eliminated. The system's functionality is not altered in any way. Rebecca Wirfs-Brock [49] in her article presents refactoring techniques.

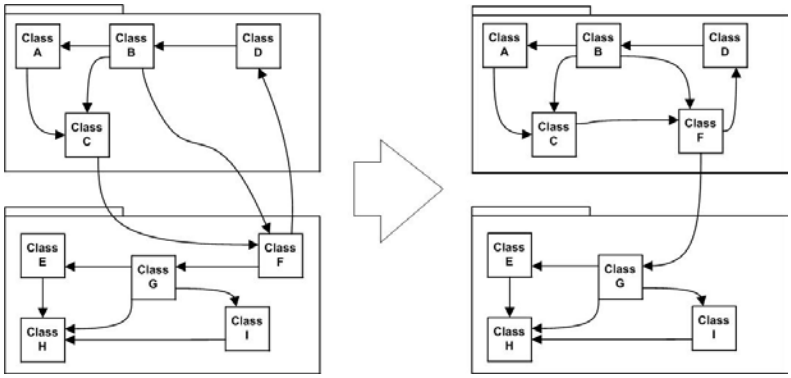

Figure 2.1: Refactoring example

Martin Fowler [9] highlights two reasons why should we refactor:

- Refactoring Improves the Design of Software.
- Refactoring Makes Software Easier to Understand.

Refactoring makes it easier to understand program's architecture by looking the code. Refactoring won't make the system run any faster, but it will, however, make a big difference in modification of the code.

Automated refactoring usualy included into IDE, for example:

- Eclipse[4] and NetBeans[5] for Java
- Visual Studio 2008 for .NET
- And many others for different languages

In some cases refactoring can be and should be avoided [9] and full rewrite is in order instead. When existing code is such a mess that it would be easier to start from the beginning.

The other time the refactoring should be avoided is when a deadline is close. At that point the productivity gain from refactoring would appear after the deadline and thus be too late.

[4]http://www.eclipse.org/
[5]http://www.netbeans.org/

Chapter 3

Clustering

3.1 Introduction

This chapter outlines our research into converting Java programs[1] into dependency graphs and applying graph clustering algorithms.

The idea involves analysing object-oriented programs to discover dependencies between classes. Using this information, we build directed dependency graphs:

- Nodes in a graph represent types, e.g. classes, interfaces
- Directed edges between the nodes represent relationships between the types

The nodes in a graph are annotated with data about the classes they represent, such as class modifiers, level of abstractness, visibility, etc. The edges contain type of relationship they represent (e.g. uses, implements, extends).

Once the graph is built the next step of processing it is filtering. This step is optional, but it allows us to exclude some parts of the system from further analysis. For example we might want to analyse only the abstract layer of the

[1] At this stage the research is applicable to Java programs

program or maybe we want to exclude the abstract layer and analyse only the concrete part of the system.

The third step is clustering, by applying Girvan-Newman clustering algorithm [12] [32], to reveal structures in the graph which are potential candidates to be components of the modular system.

More detailed information about the clustering process is contained in the section 3.5.

3.2 Background

The topic of clustering elements of object-oriented systems into subsystems has attracted the interest of many researchers over time. Many techniques, methods, metrics and tools has been developed to achieve automated clustering of monolithic software systems into components.

One interesting metric [44] named MoJo gives a numeric value for the groups of clusters. The higher the value the better the clustering layout. MoJo can not be used to compare different systems. It is used to compare different clustering layouts within the same analysed systems. A tool named "Bunch" [21], [27], [28] has been developed, that takes all the classes from an analysed object-oriented program and creates all possible cluster combinations, the one chosen should have highest MoJo value.

Many other processes and algorithms have been introduced [48], [22], [45], [8], [1] for the purpose of detecting potential modules inside monolithic software systems and restructuring those systems into module-based systems. As well, there are tools [40], [39] that are built to detect better modular structure of the software architecture and to perform automatic or semi-automatic restructuring. To back up the results of the clustering achieved there are evaluation procedures, some of these are presented in [26], [24].

Andreas Noak presents in his PhD thesis [34] quality measures that can be applied to clustering and layout of a graph. He presents measures of software

design quality based on dependencies between subsystems.

One way of clustering the elements of a set into subsets is similarity clustering [23], [17], [47]. This way, elements are grouped by some similar property they have. For example the set of classes in an object-oriented program divided into subsets of classes that belong to the same package.

There has been a study done on tracking not only class dependencies but also object interaction during run-time [37], [19]. Even though our research concentrates on the architectural level of object-oriented systems we are still able to use similar techniques in a system's analysis.

We propose novel approach for recognizing opportunities for refactoring object-oriented programs into modular structures. This method is based on detecting dependencies between classes in object-oriented programs, building dependency graphs where classes are nodes and relationships are directed edges and applying clustering algorithm in order to identify clusters of closely connected classes[12], [2], [32]. The research is dedicated to establish effectiveness of such graph clustering algorithm to detect refactoring opportunities. More details of the process are presented in section 3.5.

3.3 Reverse engineering in system's analysis

Reverse engineering [31] is a process of examining an implemented software system with the purpose of extracting its design information. This is an examining process only, the analysed software is not modified.

We can to analyse software systems using two different distributions:

- Compiled bytecode of the system (usual distribution)
- Source code of the system (open source software)

To analyse Java bytecode files we use a tool called CDA[2], kindly provided by Manfred Duchrow. The CDA tool takes .jar file as input and after processing

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

exports an XML description of the analysed .jar file. The XML file is structured with reference to the ODEM.DTD[3] type definition file presented in Appendix B.

Appendix C contains a snippet of the XML file generated as the result of analysis of JUnit library.

The important information starts with the element "container" (note: can be multiple). The container element represents a Java container .jar file in the example and it is a parent to a "namespace" element (can be multiple too). For the Java example namespace elements describe Java packages.

The namespace element is a parent of "type" elements which represent classes. The attributes of the type element hold information about class modifiers, abstractness, visibility...

The element "depends-on" contains attributes that state the relationship type and the names of the destination classes.

Our analysis of the systems (section 3.4) revealed the appearance of multiple singleton[4] clusters. The reason for the existence of singleton clusters usually comes, for example, from lost references to "final" fields in Java programming. A simple example is displayed on the figure 3.1.

The reference that is in the consumer class is replaced by the Java compiler by the value that was referenced in the provider class. At the byte-code level the classes still exist but the relationship between them is lost.

To overcome this problem and to recover lost relationships we propose source code analysis of object-oriented programs. For this purpose a Java library has been supplied by Jens Dietrich. This API scans for classes and relationships between them in the Eclipse project, and produces output XML file with reference to ODEM.DTD.

However source code analysis has limitations too:

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

[4]a single class not related to any other class in the system

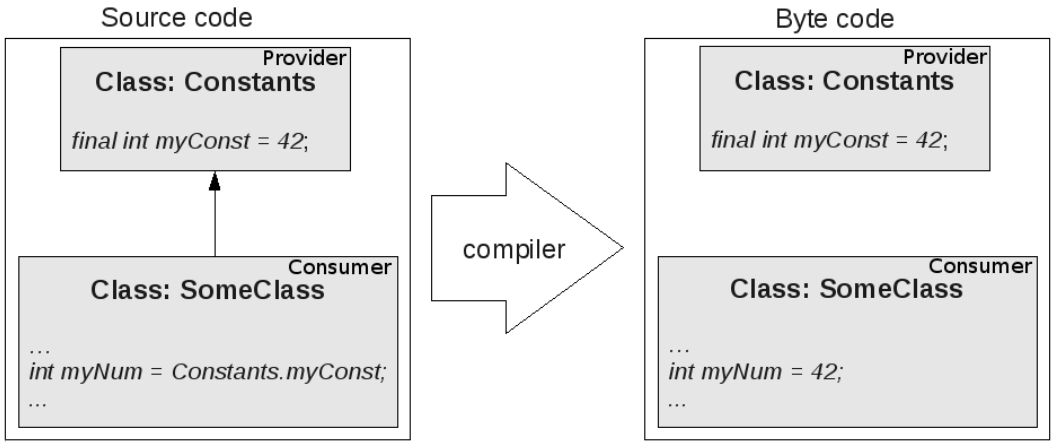

Figure 3.1: Lost reference at byte code

- Grouping into containers is not available.
- Analysis only of a single project at a time possible.
- Source code of the system often not provided.

3.4 Graph building and filtering

A graph $G = (V, E)$ is a pair of 2 sets:

- set of vertices V
- set of edges E

The elements of the set of edges E are ordered pair of the elements of the set of vertices V, for example $e_i = (v_n, v_m)$, where v_n is a source vertex, and v_m destination vertex. Having the pair of vertices ordered makes the graph

directed. If $e_j = (v_m, v_n)$ then $e_i \neq e_j$. Every element of the graph (a node and an edge) contains a collection of labels to outline properties of this element.

The dependency graph is built by parsing the XML file that was produced by CDA or by our source code analyser. For every "type" element a new vertex is added to the graph to represent it. Several properties are assigned to the vertex for the purpose of identifying the class it represents and to apply clustering algorithms. Such properties are:

- class container[5] (jar file)
- namespace[6] name
- class[7] name
- isAbstract[8] (true or false)
- isInterface[9] (true or false)
- isException[10] (true or false)
- access[11] (public, private, protected or default)

For every "depends-on" XML element a new directed edge is added to the graph to represent a relationship between classes. Similar to the vertices, edges contain some assigned properties: to identify and to apply algorithms. The edge properties are:

[5] A container is a distribution of programs, in Java - .jar files.

[6] A namespace, or a package in Java, is a mechanism for grouping classes (similar to a directory).

[7] A Java class is a group of Java methods and variables.

[8] An abstract Java type is a type that contains members that are also members of a subtype(s).

[9] An interface in Java is an abstract type that specifies what methods and variables classes must implement.

[10] An exception in Java is an event, that occurs at run-time of programs, that interrupts the normal execution of programs.

[11] Access defines rules whether the type could be accessed by other types in the program.

- edge source (graph vertex)
- edge destination (graph vertex)
- type of relationship (uses, extends, implement)

After the XML file is parsed and the dependency graph is built we introduce optional filters. The combination of graph element filters allows us to analyse not only complete systems but to analyse subsystems as well. For example, if we want to omit the abstract layer of the system and to analyse only concrete classes or vice versa. The choice of filters is dictated by the graph element's properties. For the vertices the options are:

- Interfaces
- Abstract classes
- Non-abstract classes
- Non public types
- Exceptions

For the edges available filters are:

- Extends
- Implements

3.5 Graph clustering

Graph clustering is a process of arranging elements of the graph, vertices in our case, to form groups. The question of what group a given element belongs to is decided by the clustering algorithms.

One of the clustering algorithms we use in our research is based on similarity clustering [23], [17], [47]. We know that vertices in the dependency graph have

certain properties. The vertices can be arranged into groups by container name property, and/or arranged by namespace name property. This gives us a virtual representation of the architecture of the analysed software system.

The other clustering algorithm used in our project is the Girvan-Newman algorithm [12], [2], [32]. This algorithm detects groups of the vertices in the graph that are most closely connected. An example is displayed in figure 3.2. The edges that join tightly connected groups are detected and removed from the graph, leaving us a set of component sub-graphs.

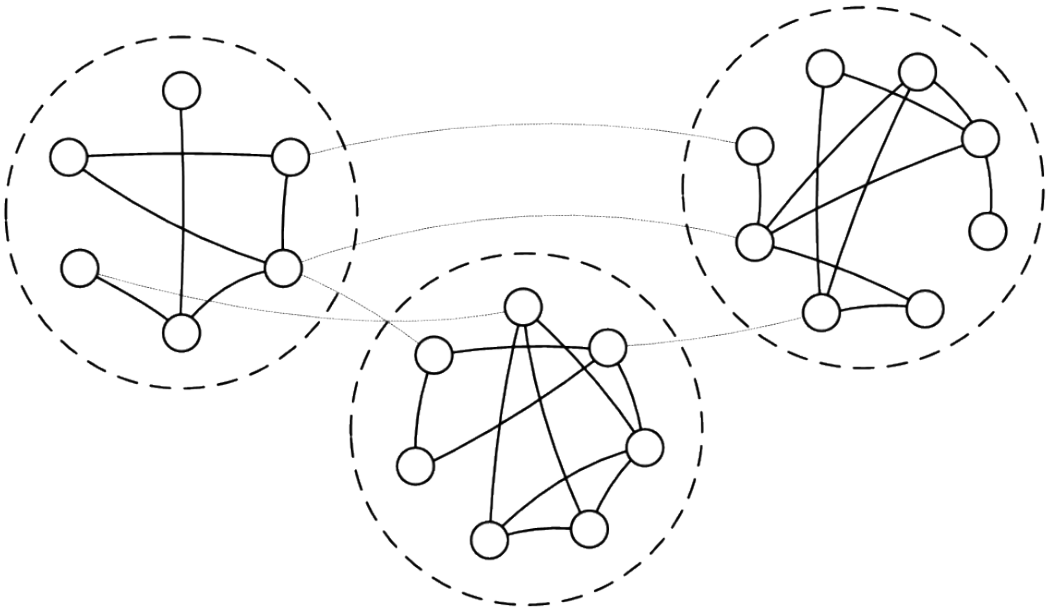

Figure 3.2: Girvan Newman algorithm

The detection of the edges that join tightly connected groups of vertices is based on the calculation of the betweenness value for every edge in the directed dependency graph. The edge or edges with the highest value are the edges that connect clusters. The betweenness value [12] of an edge is the number of all shortest paths, between all pairs of nodes, that are passing through that edge. To calculate betweenness for an edge it is necessary to count all of shortest paths in the graph between all pairs of nodes that use that edge. A simple

example, illustrated in figure 3.3, displays the calculation of betweenness for edges in the graph.

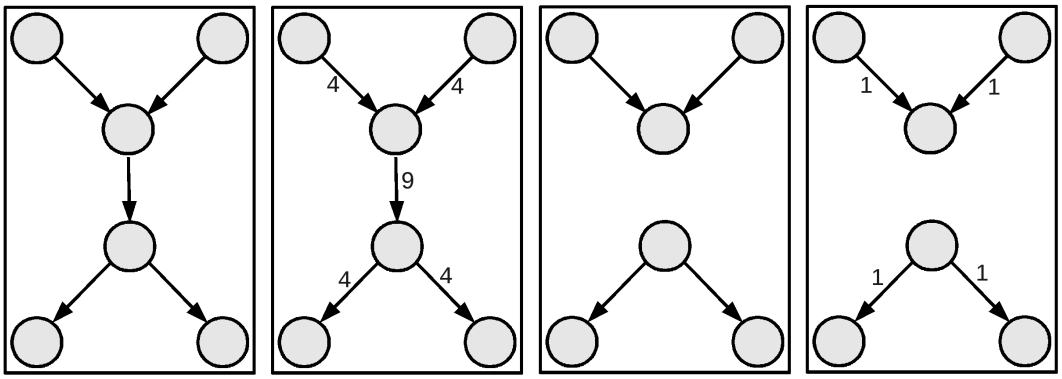

Figure 3.3: Calculating betweenness

The process of removing edges is iterative, and the betweenness values are recalculated every iteration. When the edge with the highest value of betweenness (edge with betweenness = 9 in the example in figure 3.3) is removed from the graph the dependency graph is separated into two dependency clusters. The values of betweenness for the remaining edges calculated previously are not valid for new graph structure and have to be calculated again.

Betweenness values can be confusing for people who have not dealt with them in the past. In some cases the betweenness value rises when edges are removed. Figure 3.4 displays such an example.

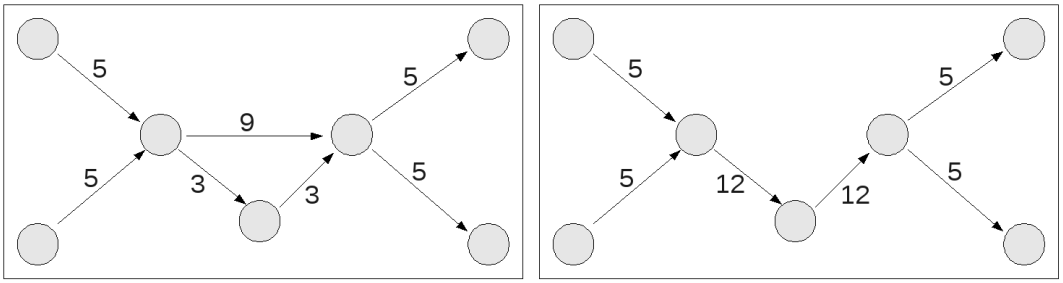

Figure 3.4: Raising betweenness

Very often when clustering large graphs (thousands of vertices) cases occur where there are multiple shortest paths between pairs of nodes, then the betweenness values become fractional. An example is shown in figure 3.5. Starting from a simple graph with 3 nodes and 2 edges the betweenness values for all edges is 2. In the graph with two alternative shortest paths between a pair of nodes the betweenness for the edges is $1\frac{1}{2}$. For the graph with three alternative shortest paths the edge betweenness is $1\frac{1}{3}$, and so on.

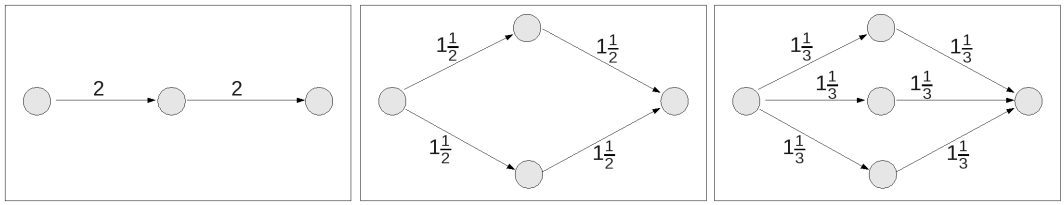

Figure 3.5: Betweenness value is a double

As mentioned before, clustering is an iterative process. The question arises: when do we stop iterations? If iterations are not stopped, eventually all edges are removed from the graph and we will get one singleton cluster for every vertex in the graph.

In our project the user makes decision as to whether to continue removing edges from the graph to achieve the desired clustering. The target the user should be trying to aim for is to have one dependency cluster (social group) of vertices per similarity cluster, in the other words - one dependency cluster per namespace, or one dependency cluster per container.

Chapter 4

Visualisation

4.1 Background

In object-oriented programming, using Java in our case, when classes are created it is common practice to group related classes into packages. The completed library of packages is distributed as a Java archive (jar) file which is based on the Zip format. Jar files can represent a complete application and/or can be used as building blocks for other applications as multiple libraries. Jar files contain an optional manifest.mf file and potentially signature files. Classes are usually created with properties which contain class descriptions:

- Modifiers: class, interface or exception
- Whether or not the class is abstract
- The class access level: public, private[1] or protected

We use this knowledge to build dependency graphs to represent software systems. Classes are nodes in the graph and relationships between them are edges. We present two different ways of displaying dependency graphs. The first approach displays a dependency graph as a table where graph elements

[1] only inner classes can be private

are placed in the rows and dependency clusters in the columns. A mark on the intersection of row and column means that the element belongs to the corresponding cluster. The second approach displaying a dependency graph as an interactive graph view. Section 4.2 presents more detailed description.

A related approach to manage and visualise the architecture of large software systems is presented by [38]. Dependencies are extracted from the code by a conventional static analysis, and shown in a tabular form known as the "Dependency Structure Matrix" (DSM). A variety of algorithms are available to help organize 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 keep the code and its architecture in conformance with one another.

A variety of tools have been implemented over time to help software developers to reverse engineer programs in order to understand a system's architecture.

Adam Buchsbaum and others [6] present a very similar approach of reverse engineering Java applets, C and C++ programs and drawing graph as the result of reverse engineering. The graph itself contain system parts as nodes and relationships between them are directed edges.

The Mondrian tool [20] tracks run-time objects and dependencies between them, it builds graphs based on captured data and provides interactive visualisation of a system at run-time.

In [15] Reid Holmes and Robert Walker present a tool called Gilligan that utilises source code analysis in order to produce structural dependency information.

4.2 Visualisations

4.2.1 Table view

One way to visualise a Java program structure is a tree view of its components as shown in the table 4.1. This was inspired by [38].

Table 4.1: Tree view of the program structure

commons-collections-3.2.jar-(458)
. . . org.apache.commons.collections-(138)
. . . org.apache.commons.collections.functors-(51)
. . . org.apache.commons.collections.list-(29)
. . . . . FixedSizeList-(pub)
. . . . . TreeList$AVLNode-(dfl)
. . . . . AbstractLinkedList$Node-(pub)
. . . . . PredicatedList-(pub)
. . . . . CursorableLinkedList$SubCursor-(pub)
. . . . . TransformedList$TransformedListIterator-(pub)
. . . . . AbstractLinkedList$LinkedListIterator-(pub)
. . . . . LazyList-(pub)
. . . . . TreeList-(pub)
. . . . . AbstractListDecorator-(pub, Abs)
. . .

The tree starts from the root element which represents a container (there can be multiple root elements). Every container element has one or more children which represent namespaces. Children of the namespace elements represent classes.

The number in the brackets next to the container and namespace element names shows the number of classes within that element. The abbreviation next to the class name displays class properties like pub - public, Abs - abstract, etc.

The other part of the table (table 4.2) contains information about the clusters. Every column represents a single cluster of classes connected by

relationships. The header displays cluster names (e.g. cluster-0). The number in the brackets next to the cluster name shows the number of classes in the corresponding cluster.

Project	cluster-0 - (427)	cluster-1 - (5)	cluster-2 - (1)	cluster-3 - (7)	cluster-4 - (1)	cluster-5 - (4)
commons-collections-3.2.jar-(458)	427	5	1	7	1	4
org.apache.commons.collections-(138)	111	5		7		4
org.apache.commons.collections.functors-(51)	51					
org.apache.commons.collections.list-(29)	29					
FixedSizeList-(pub)	x					
TreeList$AVLNode-(dfl)	x					
AbstractLinkedList$Node-(pub)	x					
PredicatedList-(pub)	x					
CursorableLinkedList$SubCursor-(pub)	x					
TransformedList$TransformedListIterator-(pub)	x					
AbstractLinkedList$LinkedListIterator-(pub)	x					
LazyList-(pub)	x					
TreeList-(pub)	x					
AbstractListDecorator-(pub, Abs)	x					

Table 4.2: Table view of the program structure.

The numbers in the rows of container and namespace elements shows the count of class elements in the intersection of the container or namespace element with the corresponding cluster element:

$$|element \cap cluster|$$

To display what class belongs to what cluster the “x” marks the appropriate cluster column in the class element row, (table 4.2).

This view type is scalable and compact for displaying the architecture of larger systems. The table view is useful for displaying the behavior of the system: how the namespaces and clusters are laid out in the architecture. The downside is that the type of the relationship between classes is lost (by type we mean: implements, extends, uses). As well, there is no indication what classes use other classes.

4.2.2 Visual graph

Java applications can consist of single or multiple .jar files. In cases like this one jar file may use one or more other jar files (jar level dependency is presented in figure 4.1).

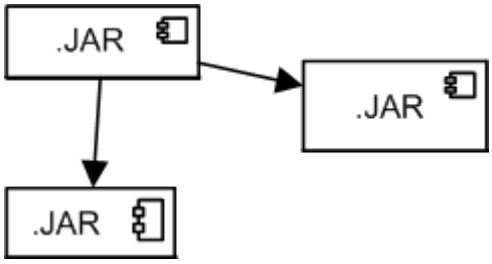

Figure 4.1: Container level dependencies

Packages have dependencies with packages belonging to the same .jar file and with packages belonging to different .jar files (package level dependencies are displayed in figure 4.2).

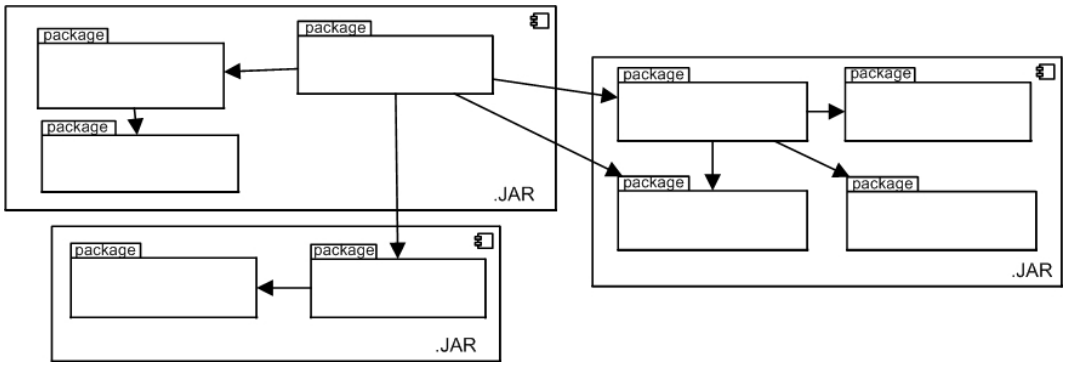

Figure 4.2: Namespace level dependencies

In this project we are interested in class level dependencies (figure 4.3). Here, dependency reflects whether the class contains variables instantiating the other class, implements an interface, extends a superclass or has a reference to a constant (final variable).

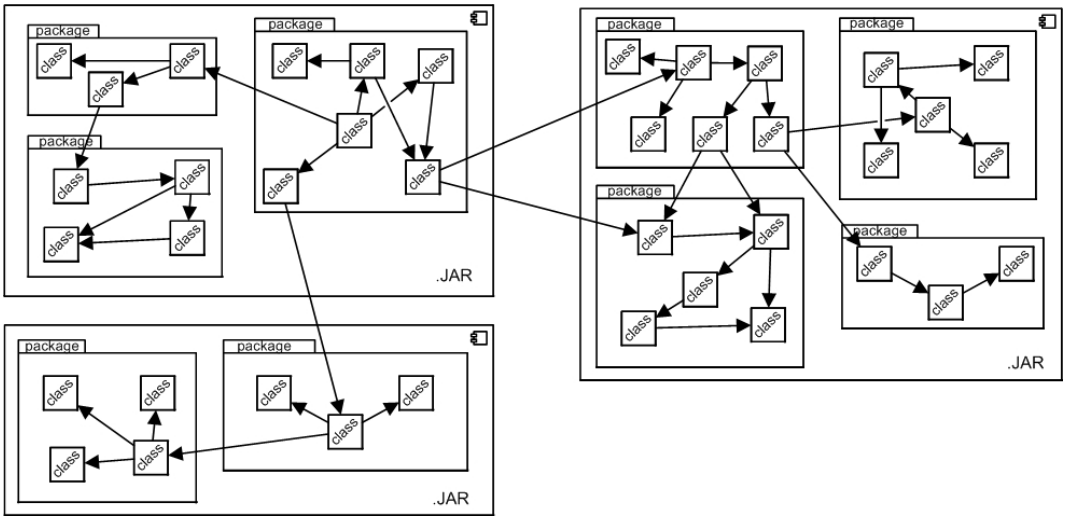

Figure 4.3: Class level dependencies

Our aim is to analyse dependencies of the object-oriented programs at class level. For this reason the nodes in the graph represent classes. The nodes are displayed with labels that contain the name of the class, the namespace name and the container name the class belongs to, and an icon that reflects class properties such as class modifier (class, interface, exception), access (public, private), and abstractness of the class (figure 4.4).

The relationships between classes are represented as directed edges in the visual graph (figure 4.5). Every edge displays a label describing the type of the relationship. The "uses" label means that the source class has at least one instance of the destination class as a variable or has a reference to a static field. The labels "extends" and "implements" mean that the source class extends a superclass or/and implements an interface.

The choice of the algorithms for laying out visual graphs comes from force directed layouts and tree layouts. There are multiple variations of them available. The comparison of these layout algorithms is presented in table 4.3.

The force directed layout [10] is used to position visual elements of the graph. Force directed algorithms view the graph as a virtual physical system,

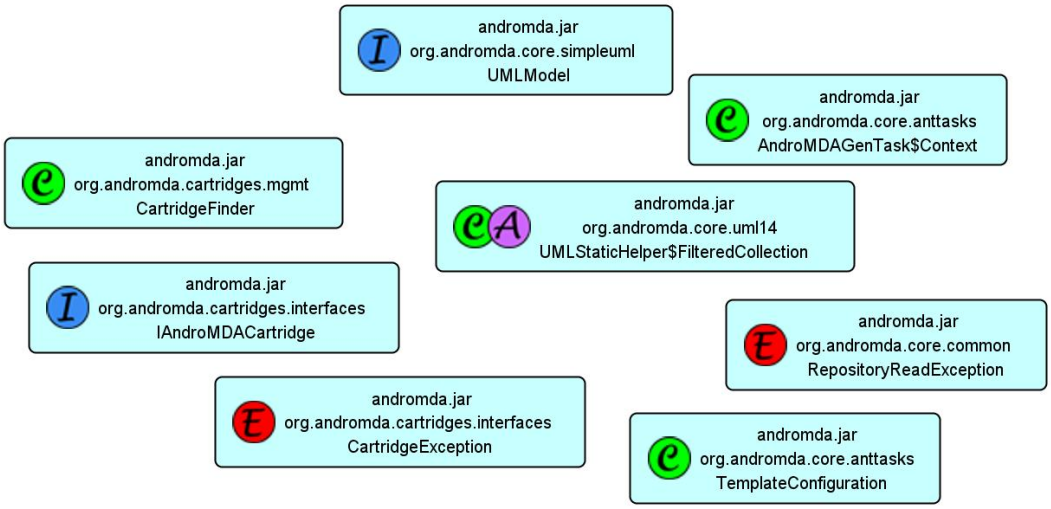

Figure 4.4: Visual nodes on the graph

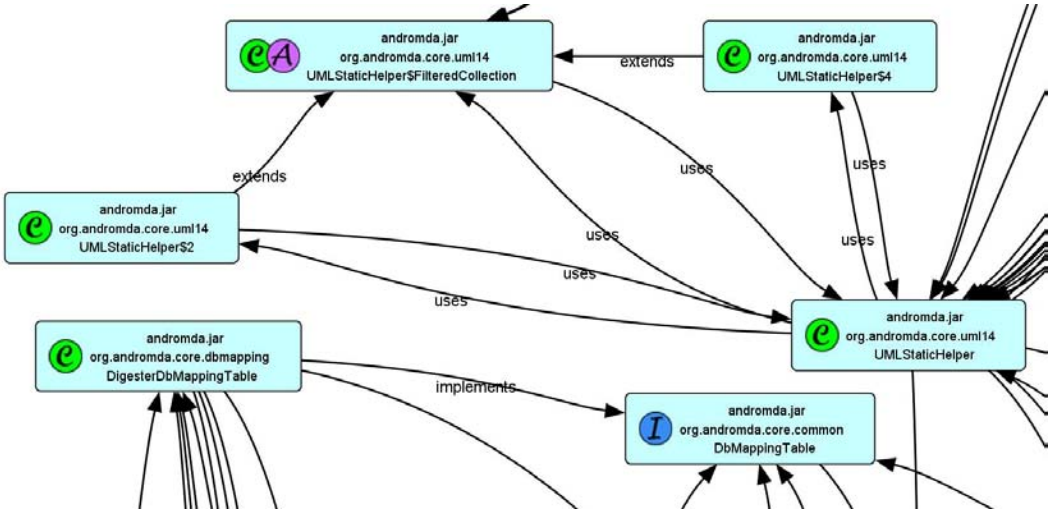

Figure 4.5: Visual edges on the graph

Graph Layout	Description	Advantages	Disadvantages
Force Directed Layout	Algorithm moves nodes on visual graphs, tries to achieve as few intersections of edges as possible.	- Simple view of graphs, - Advanced quality of results, - Easy to understand behavior.	- Low performance algorithm O(n^3).
Tree Layout	Algorithm places nodes on visual graphs in a tree like structure.	- Fast and scalable.	- Some nodes can be hidden behind others.

Table 4.3: Comparing visual graph layouts

where the nodes and edges of the graph are bodies of the system. The nodes have anti-gravitational forces acting between them, edges act as springs between nodes. Figure 4.6 shows that nodes of connected clusters are held close by the edges connecting them, and disconnected elements are repelled from each other.

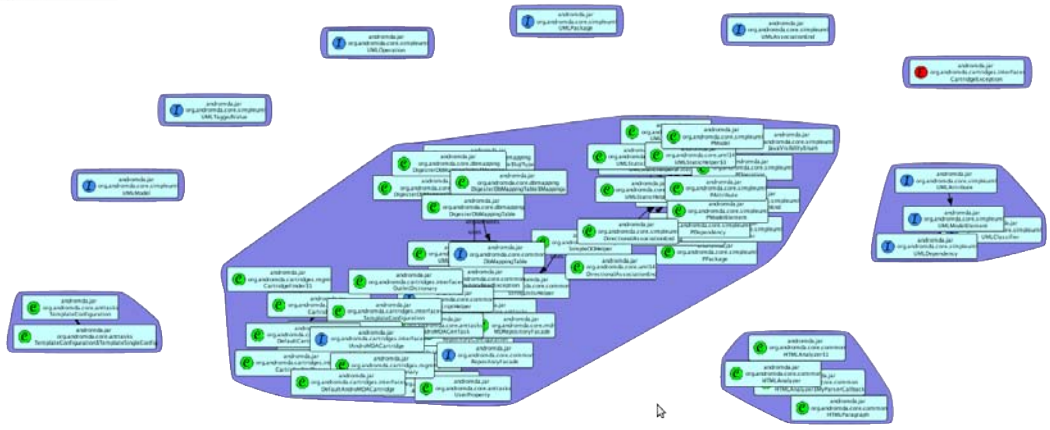

Figure 4.6: Force directed layout

In our implementation the force directed layout and its animation become ineffective or unusable in some cases for large graph data structures (more than 600 nodes). To overcome this problem force directed layout is replaced by a static[2] radial tree layout (figure 4.7) for graphs that contain over 600 nodes.

To display groups of visual nodes we use visual elements called aggregates. Aggregates draw a border around a group of visual graph nodes that contain the same value of some property. Aggregates are usually filled with colour with transparency. For example figure 4.8 shows graph nodes aggregated accordingly to the name of the package they belong to. Figure 4.9 shows packages and dependency clusters in the graph.

[2]does not contain animation

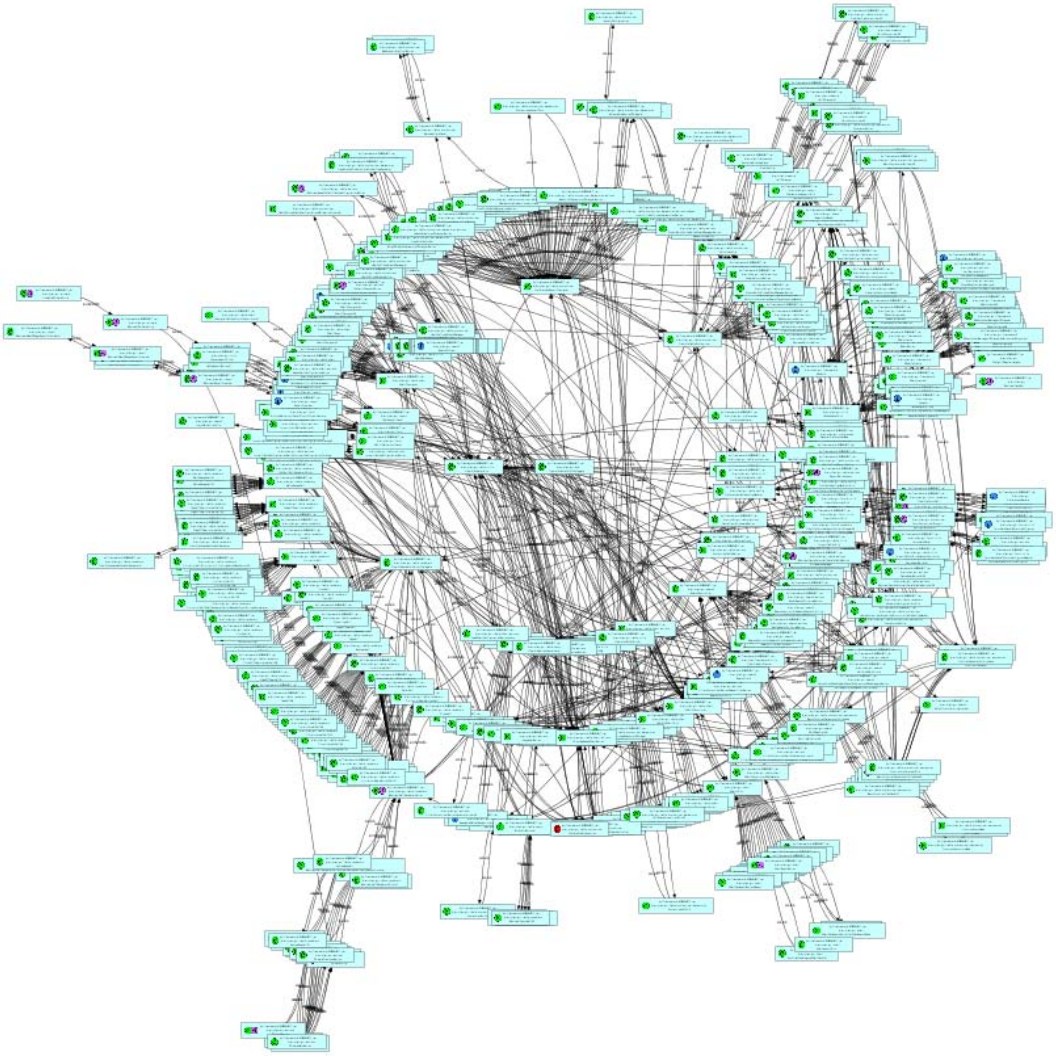

Figure 4.7: Radial tree layout

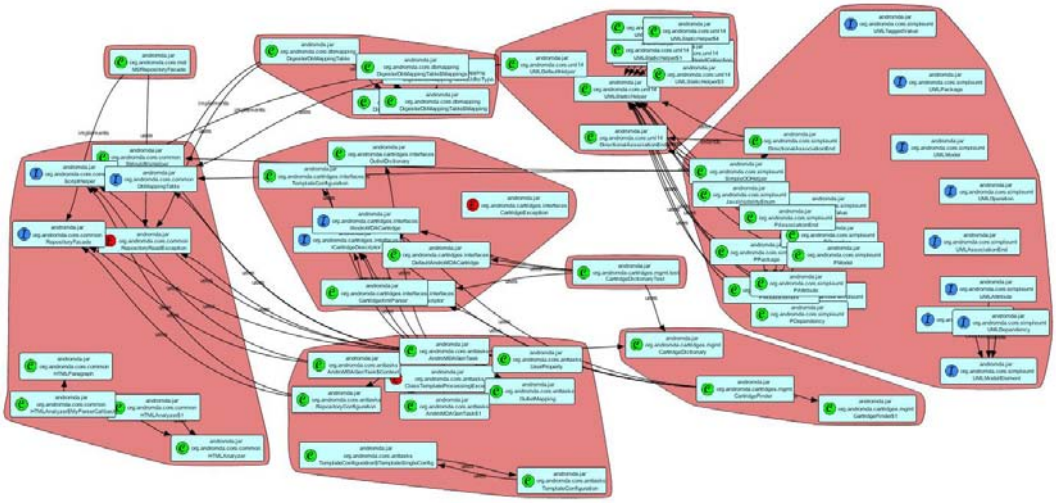

Figure 4.8: Aggregates of packages

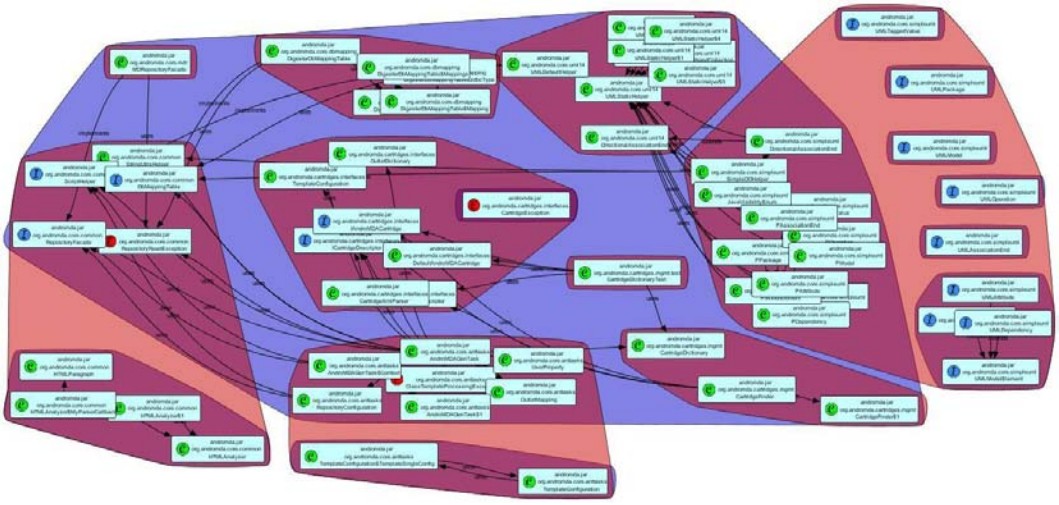

Figure 4.9: Aggregates of packages and dependency clusters

4.3 Interaction

The visual graph provides interaction options such as pan, zoom, visual element drag, etc. The drag of the visual element is controlled by mouse events using click and drag (only works for nodes and aggregates). The aim of the interaction between the user and the graph is to improve layout of visual elements in the way that user receives the display of the analysed system to have a similar layout to the model of the system itself. On user request any visual node can be moved to any position in the graph. In the cases when a user requires to move a group of nodes the aggregate that represents that group must be visible. The drag of the group of visual nodes occurs on mouse click and drag on the aggregate.

The visualisation of the graph allows the user to move the visual graph in 2D space, to pan and to zoom in or out on the visual graph with a mouse click and drag (left click and drag to pan, right click and vertical drag to zoom) or by pushing the buttons on the user interface. These functions allow the user to investigate the dependencies of subparts of the system analysed, e.g package, dependency cluster. The prefuse library [14] offers smooth zoom and pan which help users preserve their sense of position and context. This complies with Schneiderman's mantra regarding interaction techniques [41].

Highlighting occurs on user action. The application only allows to highlight a desired edge, its source and target nodes, on a mouse event such as left click on the edge. The highlighting paints edge and end-nodes with a darker colour and brings end-nodes to the front as shown in figure 4.10.

Note that the edge can be displayed only in front of other edges but always behind visual nodes. To remove highlighting the user is required to left mouse click on the edge again.

4.4 Performance measures

The performance of the visualisation is as follows:

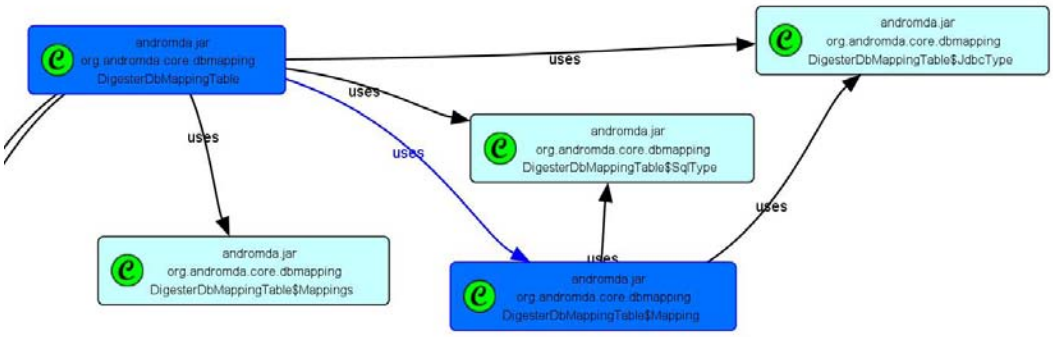

Figure 4.10: Highlighting of the visual elements

We have ran the visualisation on the selected PC with a Pentium 4 3.4GHz single core processor and 2GB of RAM running Windows XP operating system. Visual graphs with around 630 nodes and 3500 edges or less was displayed using animated force directed layout and had appeared almost instantly on the screen. Interaction with visual elements was instant and without noticeable delays.

However, attempts to display larger graphs using force directed layout resulted in delays while rendering visual elements. It took several seconds for the graph to appear on the display. Animation appeared with reduced framerate and animation was significantly less responsive. For this reason the force directed graph layout was replaced by the static radial tree layout.

As the result, initial graph rendering was instant (or took very little time). Animation was not supported by new layout. Interaction was significantly more responsive, but for larger graphs (5000 nodes and over) contained noticeable delays.

Chapter 5

Implementation

5.1 Introduction

Using Java we have implemented a tool we named Barrio. The purpose of Barrio is to test our research of the dependency analysis of object-oriented programs. Barrio builds a directed dependency graph to represent the architecture of the analysed program, then applies graph clustering algorithms to the graph in order to discover tightly connected components. Such components might indicate possibilities for restructuring the program's architecture into a more modular structure. For example, a package with two dependency clusters should be split.

Along with graph processing, Barrio produces interactive visualisation of the graph by two different methods described in Chapter 4: table view and graph view.

Barrio is implemented as a plugin for Eclipse. It takes advantage of the Eclipse Plugin Framework (EPF) by employing the Eclipse user interface elements to implement its own. An example screen-shot of Barrio is displayed in figure 5.1, where the Apache commons-collections[1] project is analysed. In the example, Barrio presents both table and visual graph outputs to the user.

[1] http://commons.apache.org

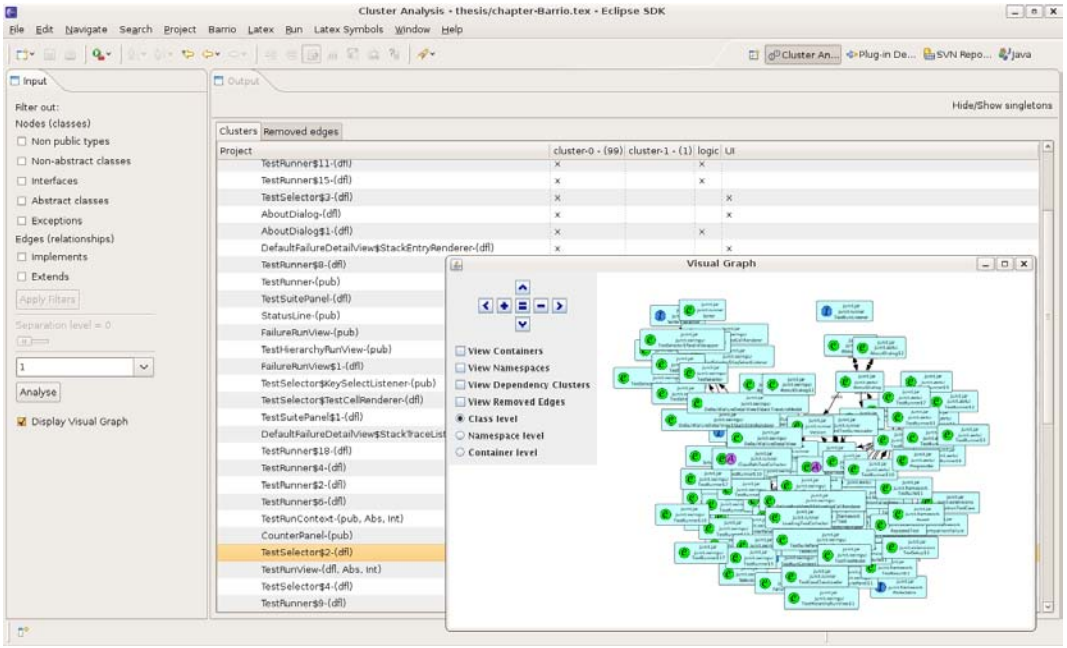

Figure 5.1: Barrio tool

Barrio is open source software under Apache licence[2] version 2. It can be downloaded and installed using the Eclipse update mechanism from its subversion repository[3].

Barrio features:

- Build dependency graph from ODEM[4] files
- Build dependency graph from source code[5]
- Apply filters to the graph elements
- Cluster graph by
  - Similarity of the properties (namespaces, containers)

[2]http://www.apache.org/licenses/LICENSE-2.0.html
[3]update site: http://barrio.googlecode.com/svn/trunk/barrio.update/
[4]produced by CDA tool see section 3.3
[5]only from Eclipse projects, API kindly provided by Jens Dietrich

  - Social network clustering
  - Rule based clustering

- Produce outputs in form of table and visual graph views[6]

5.2 Background

The Barrio project was inspired by Kiwiplan Ltd almost two years ago. The work commenced as an Honours project for my Bachelor of Engineering degree. The Barrio tool was named "Cluster Analyser". It was implemented in Java as a standalone application. The tool functionality included:

- Import .ODEM files
- Build dependency graph based on .ODEM file
- Apply filters to the graph
- Cluster graph using Girvan-Newman clustering algorithm to detect
- Produce interactive visualisation of the graph

The purpose of the project was to develop a tool that would assist Kiwiplan's developers detect possible opportunities for repackaging classes in their product software. The tool had to be scalable to handle Kiwiplan's software, that has included:

- 37 containers
- 237 namespaces
- 3116 classes
- 13095 relationships

[6] as explained in the section 4.2

To create a dependency graph the Java Universal Network/Graph Framework (JUNG[7]) was used. This Java library contains an API for graph processing, including common graph processing algorithms, clustering algorithms, etc. JUNG provides an API for the visualisation of the graph but we chose another Java library for that purpose.

The prefuse visualisation toolkit[8] [14] was used to present the visual dependency graph. The prefuse visualisation toolkit provides set features for data modeling, visualisation and interaction, and offers visual structures for tables, trees and graphs. The prefuse toolkit includes layouts and visual encoding techniques, interaction and animation. The prefuse toolkit is an open source Java library that provides 2D graphics and can be integrated into Java Swing.

5.3 Tool

Barrio takes its name from the Spanish word meaning neighborhood. This implements a metaphor of clustering into social groups of neighbours.

The first step in the implementation of Barrio was converting the standalone "Cluster Analyser" into an Eclipse plugin application. We have decided to carry on with the third party libraries we employed in the past:

- JUNG for graph processing
- Prefuse for graph visualisation

Figure 5.2 displays Barrio architecture.

Eclipse allows us to develop plugin applications by providing extension points[9] that could by used by plugins. Barrio uses four out of many extension point to integrate its user interface into the Eclipse environment.

[7] http://jung.sourceforge.net/
[8] http://prefuse.org/
[9] specific to EPF

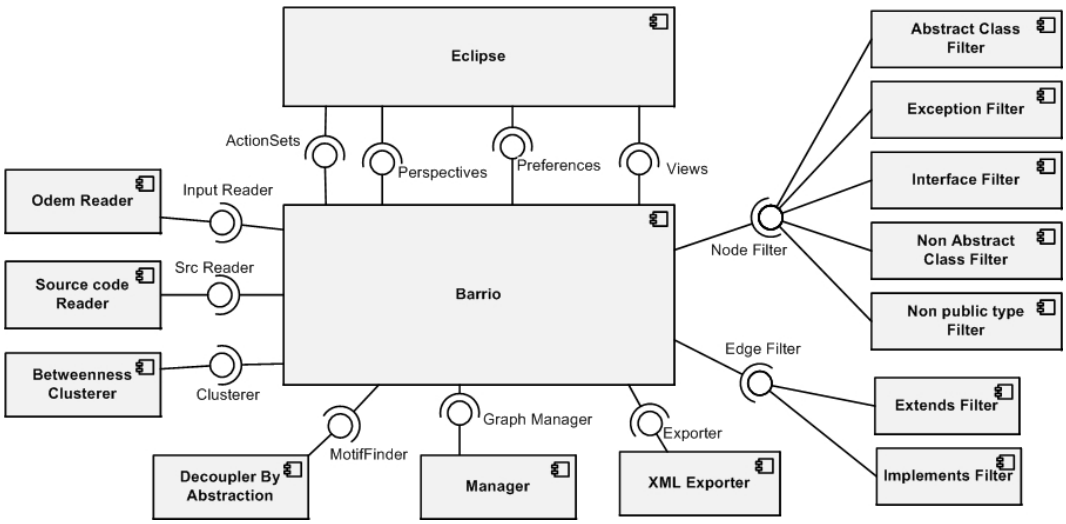

Figure 5.2: Application structure

The main plugin named "Barrio" (figure 5.2) contains extensions to match necessary extension points of Eclipse and contains user interface packages and the management API to manage plugins for itself[10].

Plugins "ODEM Reader" and "Source code Reader" are the input processors. ODEM reader takes an .ODEM file as input and converts it to the GraphML file that can be imported directly into the JUNG API for the purpose of creation of the directed dependency graph. Source code reader takes an Eclipse Java project, that is located in the currently opened workspace, as input and generates an .ODEM object in memory, or optionally an .ODEM file from GraphML file is created for JUNG API.

The next step is passing the ODEM file or object into the JUNG Graph API that returns an object of type graph[11].

Once we have the dependency graph we apply filters that allow us to remove some elements from the graph. The graph filters plugins are:

- Node filters

[10]EPF allows to create plugins for the plugins, and so on...
[11]see JUNG API specification: http://jung.sourceforge.net/doc/api/hindex.html

  - Abstract class filter
  - Non abstract class filter
  - Exception filter
  - Interface filter
  - Non public class filter

- Edge filters
  - Extends relationship filter
  - Implements relationship filter

The filters can be optionally applied to the graph in any combination. The filter selection is available to the user on the user interface displayed in figure 5.3.

We call these filters "plugable" filters which means it is allowed to add or remove them from the application at any point in time. The filter .jar files can be added or removed from Eclipse plugin folder. This feature demonstrates the customisation of software that it is possible to achieve using modular software development.

The "Betweenness Clusterer" plugin takes the filtered graph and applies the Girvan-Newman algorithm to the graph. An implementation of this algorithm is available in the JUNG API but to fit our requirements it was modified via overriding methods. The modified algorithm allows us to detect points of cluster separation[12] in the graph.

The process of the Girvan-Newman algorithm is iterative. It removes one, or several, edges that have the highest value of betweenness, per iteration. The process stops at the final point of cluster separation, which is specified by the user, or runs till all the edges are removed. In both cases the betweenness clusterer plugin returns all detected points of separation. The user specifies

[12]the point of cluster separation is an appearance of newly formed dependency cluster

how many cluster separations are necessary using the drop down menu on the user interface (figure 5.3).

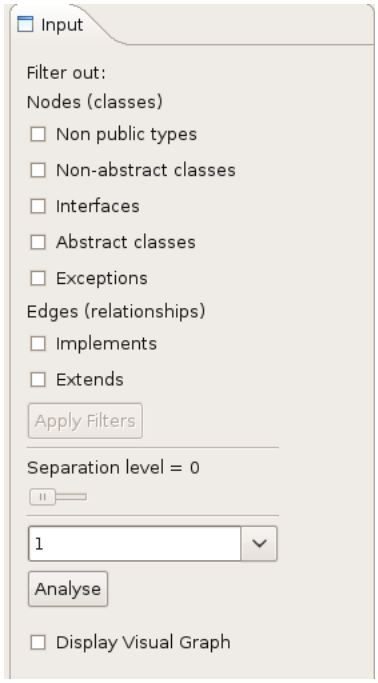

Figure 5.3: Input User Interface

The "Manager" plugin is a simple converter from type Graph to other types that are usable by widgets on the user interface. It is used for generation of the outputs.

The "XML Exporter" plugin is an optional functionality that allow the user to export clustering results into an XML file for further processing and/or comparison. The XML file structure includes a list of the removed relationships (if any) and the advice on separation of the application into components. An example of such a file is given here:

```
<graph file=".../JUnit.odem" separation-level="45">
    <initial-state>
        <nodes count="100"/>
        <edges count="276"/>
```

```
    </initial-state>
    ...
    <removed-relationship source="junit...Logo" target="junit...BaseTestRunner" type="uses"/>
    ...
    <component id="1">
        ...
        <package name="junit.awtui">
            ...
            <class name="TestRunner$6"/>
            ...
        </package>
        <package name="junit.swingui">
            ...
        </package>
    </component>
    <component id="2">...</component>
    ...
    <component id="n">...</component>
</graph>
```

The final plugin is "Decoupler by Abstraction". This is implemented for the purpose of research of alternative anti-patterns we have not defined so far in this project. When we were applying abstract relationship[13] filters to some dependency graphs we discovered that removing such relationships causes cluster separation. This event occurs if there is the following motif[14] (figure 5.4) present in the graph.

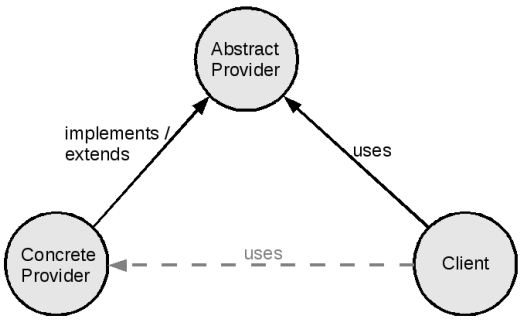

Figure 5.4: Motif

[13]extends and/or implements

[14]recurring structure of the pattern

The Abstract provider (Interface or abstract class) has been used by the concrete class (the client) of the program, and there exists a concrete provider that implements such an abstract provider (if it is an interface class) or extends that abstract provider (if it is an abstract class), and there is no relationship from the client to the concrete provider. In other words when the implementation of the abstract class exists but is not used.

The "Decoupler by Abstraction" plugin was implemented to serve the purpose of detecting all occurrences of the motif described above. All of the occurrences are candidates to be an anti-pattern. Once such an occurrence is detected, the user needs to review it.

Appendix D contains public interface API for defining plugin contracts.

Figure 5.5 presents the display of the information flow within the Barrio application:

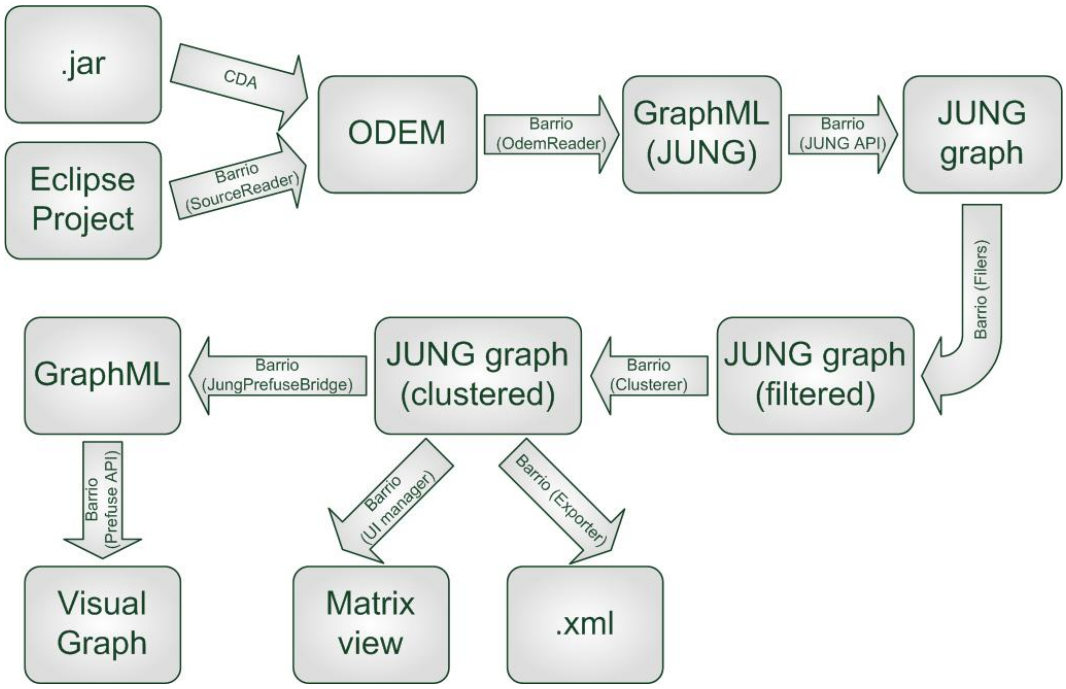

Figure 5.5: Data Flow Diagram

To display the matrix or table view that was described in section 4.2 the

standard widget toolkit is used on the output user interface (figure 5.6). We have integrated dependency clusters of the graph with rule defined clusters[15] in the table view.

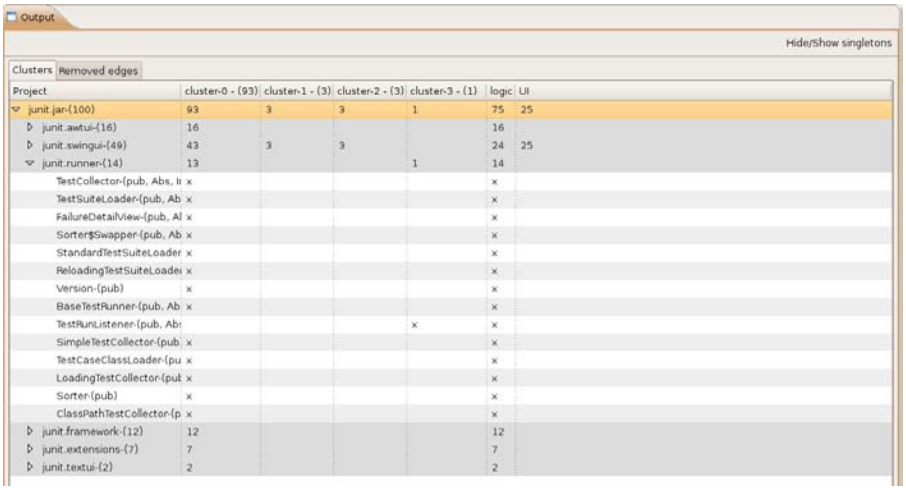

Figure 5.6: Table (Matrix) View

For the visual graph display Barrio uses prefuse [14] interactive visualisation. The example is displayed in figure 5.7.

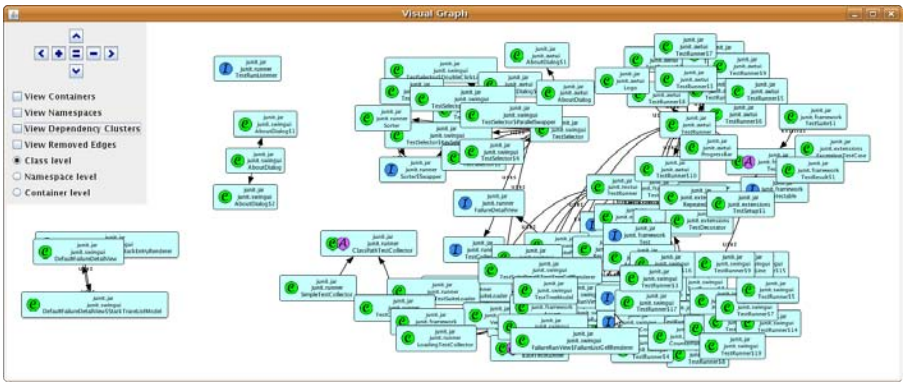

Figure 5.7: Barrio visualisation of the dependency graph

[15] section 6.3

The left top corner of the visualisation is occupied by the navigation controller. It begins with navigational buttons along with zoom functions. This is followed by the visual group selectors that allow the user to visually display a group of the graph elements (or a combination of the groups).

Chapter 6

Evaluation

6.1 Evaluation of the Girvan-Newman clustering algorithm

To evaluate our clustering algorithms analysis of the following software was performed:

- Software applications that were coded at Massey University (smaller scale applications - up to 50 classes, 200 relationships)
- Kiwiplan's product software (larger scale application - over 3000 classes, and about 14000 relationships)
- Open source software applications (various sizes from small - 1 classes and 0 relationships to large - 5300 classes and 30000 relationships)

The open source software applications were supplied to us as a organized collection of various applications that is named Qualitas Corpus[1]. The Qualitas Corpus is managed by Ewan Tempero at the University of Auckland. The version of the Qualitas Corpus that was analysed is 2008.3.12.

[1]http://www.cs.auckland.ac.nz/ ewan/corpus/

We have extracted 768 jar containers from the collection of the software and ignored containers with fewer then 10 classes.

Then we have arranged the remaining containers into 4 groups:

- containers with 10 to 49 classes - 189 containers
- containers with 50 to 199 classes - 230 containers
- containers with 200 to 599 classes - 170 containers
- containers with 600 classes or more - 59 containers

Overall we have analysed 648 java open source programs with 162,526 classes and 744,821 relationships combined.

We have automated the tool to analyse all these applications to the level of three cluster separations[2] each. The selected PC to perform the analysis was a Pentium 4 with 3.4GHz single core processor and 2GB of RAM running Windows XP operating system. The analysis took 5 days, 6 hours and 17 minutes.

The results of the analysis are as follows:

The algorithm successfully detected initial[3] dependency clusters within the program in 100% of the cases these were present. Such clusters are an indication of the possibility of splitting the container (singleton clusters are ignored). This feature proved useful for the Kiwiplan developers. It helped to point out the containers with multiple clusters and allowed developers and designers to analyse possible anti-patterns and consider alterations to the software if necessary.

The algorithm successfully removed edges with highest betweenness and detected newly formed dependency clusters in 100% of cases. The downside

[2]the Girvan-Newman algorithm is applied until new dependency clusters formed three times

[3]separation was not applied, no edges were removed from the graph

was that in many graphs a very large number of edges needed to be removed to achieve cluster separation[4]. In many cases the cluster that gets separated from the parent cluster is significantly smaller in size. A similar effect was reported to us by Keith Cassell from Victoria University. While analysing the collection of the open source software the cluster analysis using Girvan-Newman algorithm was applied to different version of the same software. The results of the clustering did not follow the version progression in all of the test cases. In none of the cases of the analysis was the cluster structure relatively close to packaging. In non of the cases of the analysis the cluster layout followed the packaging in the version progression of the software.

As a result, we conclude that the Girvan-Newman algorithm applied to directed dependency graphs is not effective for finding modular structures in object-oriented programs.

6.2 Evaluation of the visualisation

At first in the early stages of the project the implementation of the visualisation of the graph was outside of the project scope. Due to research progress and having available time the visualisation has been included.

From the initial stage the visual representation of the dependency graph built based on the interaction between classes in object-oriented programs attracted a lot of interest from developers. Almost instantly the visualisation showed an easy way to display the actual architecture of the applications.

After improving the visual appearance of the dependency graph by adding more advanced graphics, layouts, interaction, etc it was presented to developers at Kiwiplan and instantly received positive feedback. It showed that developers were able constantly compare the software architecture while in the process of implementation to the designed architecture and eliminate flaws if there were any as they proceeded.

[4]newly formed cluster occurs

Subsequently, we presented our work [7] at an international software visualisation conference. Again, the visualisation was proved to be effective and useful and received many positive comments.

The detractions of the visualisation are that layout animation and interaction become slower for larger graphs (over 1000 nodes), and searching for a part of the graph is difficult as there is no search mechanism implemented.

6.3 Rule defined clustering

The main idea here is to define rules to cluster sets of elements into subsets. In [43] a tool is presented that uses rules to regulate subsystem interactions.

To start off, lets see what kind of data is available for rule processing. The analysed object-oriented program contains containers (.jar files if a Java program is analysed). Every container holds namespaces (packages). Every namespace has one or more classes within (there is no reason to have an empty namespace.)

Analysis of an object-oriented program at class level will reveal that classes (in most cases) in the program have dependencies between each other, they have dependencies with classes belonging to third party libraries (if such libraries are used) and they have dependencies with classes belonging to the environment the program is designed to run in. For example, Java applications are designed to run in the environment called Java Virtual Machine (JVM), and most classes in such application have reference to java.lang.Object class. The class java.lang.Object is not defined by the system being implemented, it is defined by JVM.

We know that almost every class has classes it depends on. Based on this information a set of rules can be constructed. A simple rule example is shown below where the rule implies that if a class depends on any class in the package javax.swing then this class should be labeled as a User Interface (UI) class. There is a similar rule for classes that depend on the package org.eclipse.swt.

Another rule implies that if a class does not depend on any of the classes from packages javax.swing and org.eclipse.swt then that class labeled as a logic class.

If a class ***references*** *javax.swing* ***then it is*** *User Interface (UI);*
If a class ***does not reference*** *javax.swing*
and
does not reference *org.eclipse.swt* ***then it is*** *logic;*

The user defines all rules and labels. From all user defined rules the set is built, figure 6.1. The set of rules is then applied to the object-oriented program being analysed. Every class in the program that satisfies at least one rule receives a label according to the rule. Note that some classes may satisfy multiple rules and, as the result, may receive multiple labels.

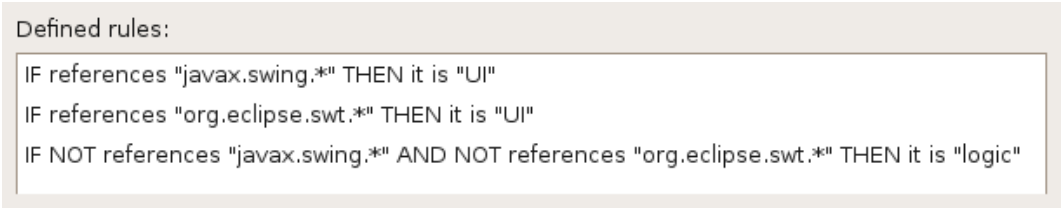

Figure 6.1: Building a set of rules

After applying this set of three rules defined to the java program JUnit we get the results displayed in table 6.1 and table 6.2. Both tables display exactly the same results. On the left hand side of the results tables in the "Project" column there is a representation of the tree structure of the analysed program. Where the top element represents a container (can be multiple), the child elements of the container are namespaces, and the children of every namespace are classes. The number in the brackets next to the container name or the namespace name represent number of classes that element contains. The abbreviation next to the class name is a display of the class properties (dfl=default, pub=public, abs=abstract, etc).

Table 6.1: Rule clustering result of JUnit.jar

Project	UI	logic
junit.jar-(100)	25	75
...junit.swingui-(49)	25	24
...junit.runner-(14)		14
...junit.awtui-(16)		16
...junit.framework-(12)		12
...junit.extensions-(7)		7
...junit.textui-(2)		2
...		

The columns "UI" and "logic" represent results after applying the example set of rules. For example, in the first row: junit.jar-(100) has 25 classes that are labeled User Interface and 75 classes that are labeled logic. At this stage there is no problem detected. However, namespace junit.swingui-(49) contains both UI classes and logic classes within. That means this namespace can be (should be) split into two: one containing only UI classes and the other for logic classes. This would give developers an easier replacement of the user interface in the future if required.

Note that there is a danger of incorrectly defined rules producing incorrect results.

6.4 Evaluation of the rule based clustering

To evaluate rule based clustering the relationship rules were applied to the same set of Java projects as in section 6.1.

Testing the rule based clustering algorithm on some of the open source software revealed software engineering pattern violations in some cases.

The implemented version of the rule processor was integrated into Barrio and was presented to Kiwiplan developers. The Kiwiplan products were tested against a set of defined dependency rules which produced rule based clusters.

Table 6.2: Rule clustering result of JUnit.jar

Project	UI	logic
junit.jar-(100)	25	75
. . . junit.swingui-(49)	25	24
. . . . . FailureRunView$FailureListCellRenderer-(dfl)	⋆	
. . . . . TestHierarchyRunView$1-(dfl)	⋆	
. . . . . TestRunner$3-(dfl)		⋆
. . . . . TestSelector$ParallelSwapper-(dfl)		⋆
. . . . . TestTreeModel-(dfl)	⋆	
. . . . . TestSuitePanel$TestTreeCellRenderer-(dfl)	⋆	
. . . . . ProgressBar-(dfl)	⋆	
. . . . . DefaultFailureDetailView-(pub)	⋆	
. . . . . TestRunner$19-(dfl)		⋆
. . . . . TestRunner$5-(dfl)		⋆
. . .		

The clusters were analysed and in 100% of the cases returned positive feedback from developers, where the rule based clustering process successfully identified anti-patterns if such occurred.

The rule based clustering algorithm proved itself useful in defining stereotypes[5] and providing visual indications of the rule based cluster layout. It helped developers to determine possible rule violations in the software system.

[5]a stereotype is categorization of classes

Chapter 7

Future work

7.1 Refactoring

The next step should be integration of refactoring into the Barrio tool. Automated refactoring will be achieved by generating ANT refactoring scripts. The clustered dependency graph will be used as a basis for the new system architecture. The physical availability of the refactored solution will allow us to compare outputs of different tools and algorithms, calculating modularity metrics [44], [46].

7.2 Analysis of undirected graphs

During our clustering experiments we have applied the Girvan-Newman algorithm to undirected dependency graphs. Analysis of the performance of the algorithm on undirected graphs appeared to indicate that it can separate clusters faster. By faster, we mean that fewer edges are removed to reveal dependency clusters but it takes a longer time to calculate betweenness (because there more shortest paths to consider when calculating). In some cases, the clustering that was highlighted followed packaging more closely than clustering highlighted during processing of the directed graph.

The presented example is an analysis of the application CTools.jar. Figures 7.1 and 7.2 display results of the undirected dependency graph analysis and figures 7.3 and 7.4 display results of the directed graph analysis of this application. The figures display results that contain up to five levels of cluster separation. That is, we have applied the Girvan-Newman algorithm to a graph and removed edges until five newly formed dependency clusters appeared.

The first major interesting difference is that cluster separations occur for different numbers of edges removed. For the directed dependency graph the numbers are: 16, 170, 177, 182 and 206. For the undirected graph the numbers of edges removed are: 8, 19, 22, 30 and 42. The other very interesting observations are that newly formed clusters are relatively similar in size and they follow the package structure much more closely in undirected results compared with the directed results.

Further research will investigate the Girvan-Newman algorithm applied to undirected dependency graphs with the purpose of establishing effectiveness of the algorithm when used for detecting modular structures in object-oriented programs. We will also compare this algorithm, using similarity metrics [29], to other clustering algorithms.

7.3 Improvements

If application of the Girvan-Newman algorithm to undirected dependency graphs proves itself effective in recognizing software modules, then the implementation of the algorithm could be improved using a modified faster algorithm to calculate the value of the betweenness for the graph edges [4].

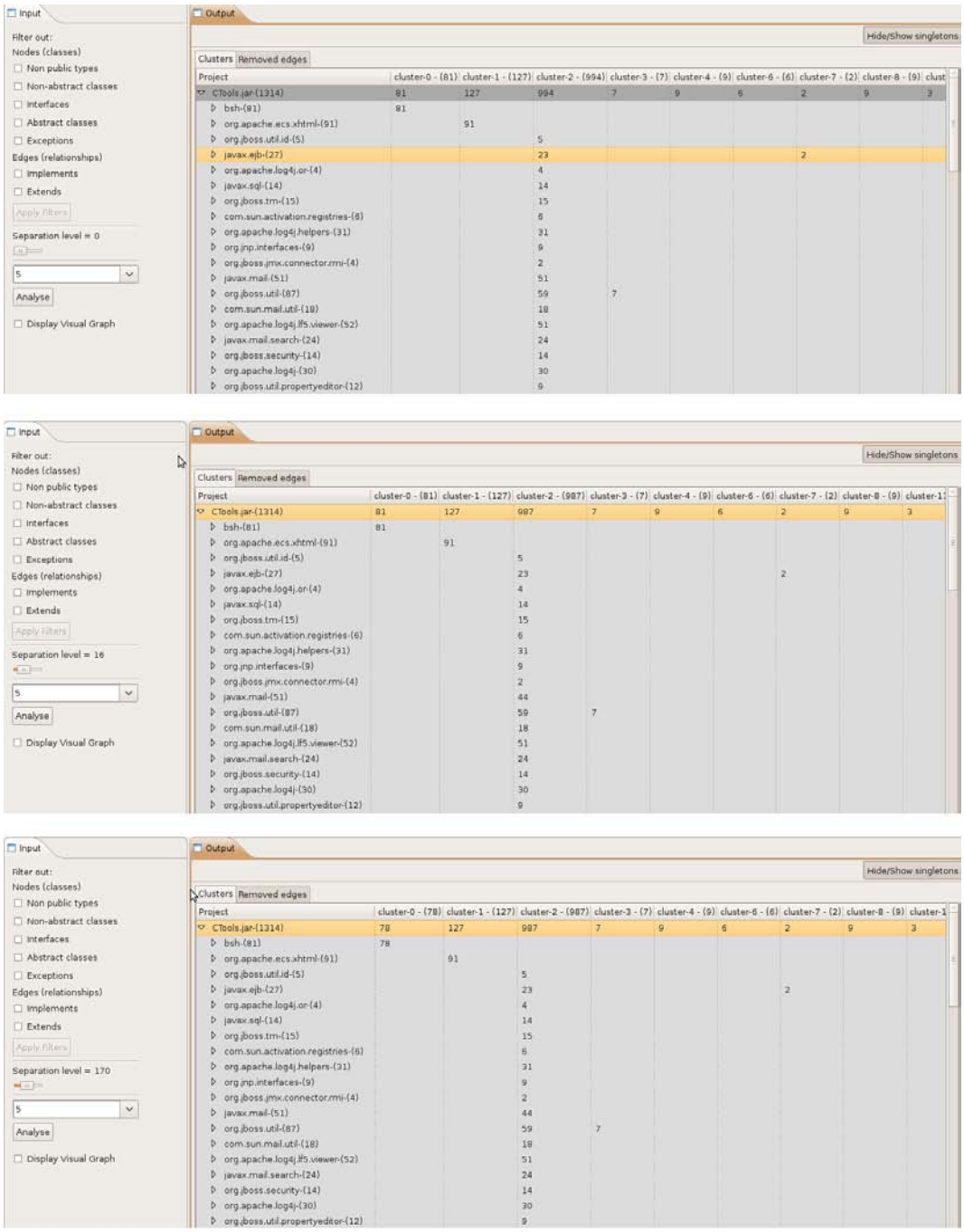

Figure 7.1: Directed analysis

Figure 7.2: Directed analysis (ctd.)

Figure 7.3: Undirected analysis

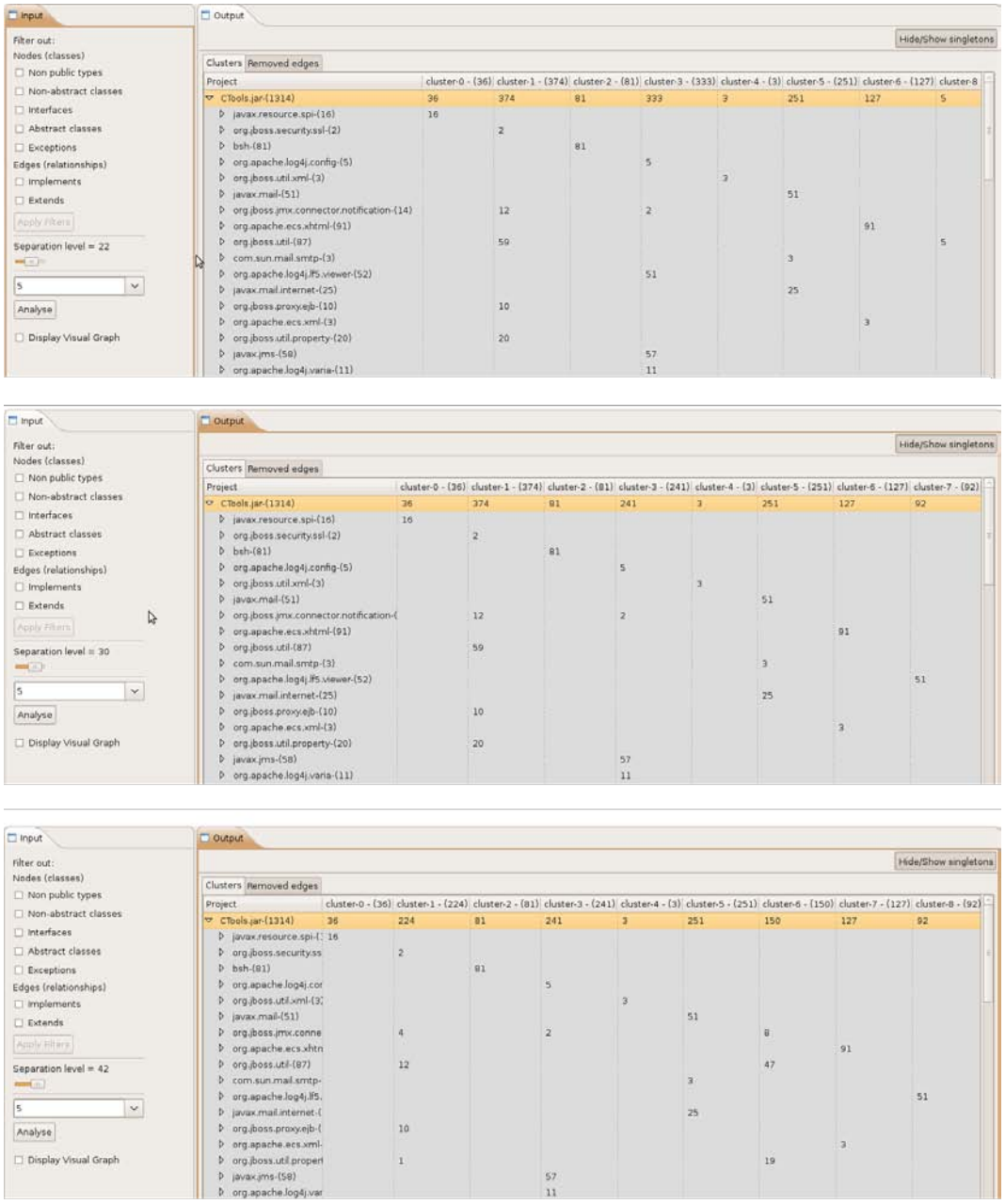

Figure 7.4: Undirected analysis (ctd.)

Chapter 8

Conclusion

The overriding aim of our research is to to recognize potential software modules in monolithic software systems.

Our approach is based on building directed dependency graphs reflecting the interaction between classes in object-oriented programs. Graph clustering algorithms are applied to the dependency graphs 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.

Our work uses the Girvan-Newman clustering algorithm developed to analyse social networks. This algorithm calculates the value of betweenness for every edge in the graph and removes edges that have the highest value of the betweenness. The betweenness value of an edge is the number of shortest paths between every pair of the vertices in the graph, that pass through that edge.

The evaluation of the results of applying the Girvan-Newman graph clustering algorithm did not discover any proof that this clustering algorithm is effective in detecting subsystem candidates when applied to a directed dependency graph. A new evaluation is required in the future for the same algorithm when applied to undirected dependency graphs. In some cases it has shown

potential to detect subsystems in this context.

Rule based clustering was also investigated in this research to add another dimension to the view of the system architecture. The results of applying rule-based clustering show it to be useful to practising software developers.

To demonstrate and test our research we developed 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. BARRIO is distributed as open source software under Apache licence version 2. It is available as an Eclipse plugin from project SVN[1].

The tool contains the implementation of the clustering algorithms and returns visual output of the analysis via user interface widgets and visual interactive graphs. The visualisation has proved to be useful and effective to software developers in academic and commercial environments.

[1]http://code.google.com/p/barrio/

Bibliography

[1] Periklis Andritsos and Vassilios Tzerpos. Software clustering based on information loss minimization. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 334, Washington, DC, USA, 2003. IEEE Computer Society.

[2] D. Auber, Y. Chiricota, F. Jourdan, and G. Melancon. Multiscale visualization of small world networks. In *IEEE Symposition on Information Visualisation*, pages 75–81. IEEE Computer Society, 2003.

[3] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of java applications for multithreaded antipatterns. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[4] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[5] William J. Brown, Raphael C. Malveau, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley, March 1998.

[6] Adam Buchsbaum, Yih-Farn Chen, Huale Huang, Eleftherios Koutsofios, John Mocenigo, Anne Rogers, Michael Jankowsky, and Spiros Mancoridis. Visualizing and analyzing software infrastructures. *IEEE Softw.*, 18(5):62–70, 2001.

[7] Jens Dietrich, Vyacheslav Yakovlev, Catherine McCartin, Graham Jenson, and Manfred Duchrow. Cluster analysis of java dependency graphs. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visuallization*, pages 91–94, New York, NY, USA, 2008. ACM.

[8] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *In Proceedings of Software Technology and Engineering Practice*, pages 73–91, 1999.

[9] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[10] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[12] Michelle Girvan and M. E. J. Newman. Community structure in social and biological networks. Working Papers 01-12-077, Santa Fe Institute, 2001.

[13] Mark Grand. *Patterns in Java, volume 1: a catalog of reusable design patterns illustrated with UML*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[14] Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceeding of the SIGCHI conference on Human factors in computing systems*, pages 421–430, New York, NY, USA, 2005. ACM Press.

[15] Reid Holmes and Robert J. Walker. Task-specific source code dependency investigation. *vissoft*, 0:100–107, 2007.

[16] Andrew Koenig. Patterns and antipatterns. *Journal of Object-Oriented Programming.*, 8:46–48, 1995.

[17] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute for Computer Science, University of Stuttgart, 2000.

[18] Phil Laplante, Robert R. Hoffman, and Gary Klein. Antipatterns in the creation of intelligent systems. *IEEE Intelligent Systems*, 22(1):91–95, 2007.

[19] Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz. Tracking objects to detect feature dependencies. *icpc*, 00:59–68, 2007.

[20] Adrian Lienhard, Adrian Kuhn, and Orla Greevy. Rapid prototyping of visualizations using mondrian. *vissoft*, 0:67–70, 2007.

[21] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *In Proceedings of International Conference of Software Maintenance*, pages 50–59. IEEE Computer Society Press, 1999.

[22] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 45, Washington, DC, USA, 1998. IEEE Computer Society.

[23] Lennart Hofland Arie van Deursen Marco Glorie, Andy Zaidman. Splitting a large software archive for easing future software evolution. Technical report, Delft University of Technology, 2008.

[24] Makoto Matsushita. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Eng.*, 31(3):213–225, 2005. Member-

Katsuro Inoue and Member-Reishi Yokomori and Member-Tetsuo Yamamoto and Member-Shinji Kusumoto.

[25] M. D. McIlroy. Mass produced software components. In *Proceedings NATO Software Eng. Conf., Garmisch, Germany*, pages 138–155, 1968.

[26] Brian S. Mitchell and Spiros Mancoridis. Craft: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 93, Washington, DC, USA, 2001. IEEE Computer Society.

[27] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng.*, 32(3):193–208, 2006.

[28] Brian Scott Mitchell. *A heuristic search approach to solving the software clustering problem*. PhD thesis, Drexel University, Philadelphia, PA, USA, 2002. Adviser-Spiros Mancoridis.

[29] B.S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 744, Washington, DC, USA, 2001. IEEE Computer Society.

[30] Naouel Moha. Detection and correction of design defects in object-oriented designs. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 949–950, New York, NY, USA, 2007. ACM.

[31] Hausi A. Mller, Mehmet A. Orgun, Scott R. Tilley, James, and S. UhlMller. A reverse engineering approach to subsystem structure identification, 1993.

[32] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, 2004.

[33] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. *Object-oriented software composition*, Prentice Hall International (UK) Ltd.:3–28, 1995.

[34] Andreas Noack. *Unified quality measures for clusterings, layouts, and orderings of graphs, and their application as software design criteria*. PhD thesis, Brandenburg University, 2007.

[35] Trevor Parsons. *Automatic Detection of Performance Design and Deployment Antipatterns in Component Based Enterprise Systems*. PhD thesis, University College Dublin, 2007.

[36] Colin Neill Phillip Laplante. *AntiPatterns: Identification, Refactoring, and Management*. CRC Press, Taylor and Francis Group, 2006.

[37] Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: From object-interactions to feature-interactions. *icsm*, 00:72–81, 2004.

[38] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. 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*, pages 164–165, New York, NY, USA, 2005. ACM.

[39] Amit P. Sawant and Naveen Bali. Softarchviz: A software architecture visualization tool. *vissoft*, 0:154–155, 2007.

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

[41] Ben Shneiderman. 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*, page 336, Washington, DC, USA, 1996. IEEE Computer Society.

[42] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[43] Martin Traverso and Spiros Mancoridis. On the automatic recovery of style-specific architectural relations in software systems. *Automated Software Engineering*, 0:331–360, 2002.

[44] Vassilios Tzerpos and R. C. Holt. Mojo: A distance metric for software clusterings. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 187, Washington, DC, USA, 1999. IEEE Computer Society.

[45] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 246–255, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[46] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. *iwpc*, 00:194, 2004.

[47] Zhihua Wen and Vassilios Tzerpos. Evaluating similarity measures for software decompositions. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 368–377, Washington, DC, USA, 2004. IEEE Computer Society.

[48] T.A. Wiggerts. Using clustering algorithms in legacy systems remodularization. *wcre*, 0:33, 1997.

[49] Rebecca J. Wirfs-Brock. Valuing design repair. *IEEE Softw.*, 25:76–77, 2008.

Appendix A

Anti-pattern definitions

Antipattern Name:	God Class
Also Known As:	Monster Class, The Blob
Defined by:	http://sourcemaking.com/antipatterns/the-blob
Description:	A class with a high number of incoming references
Example:	java.lang.System
Motivation:	Glues together components, reduces modularity
Detection:	Count incoming references, look for number larger than threshold

Antipattern Name:	Busy Bridge
Also Known As:	Harbour Bridge :-)
Defined by:	Barrio development members
Description:	Link with high betweenness
Example:	Two interconnected clusters have a connection between them. That connection is a busy bridge.
Motivation:	Glues together components, reduces modularity
Detection:	Use Girvan-Newman algorithm to detect and remove such a link

Antipattern Name:	Domain Model depends On User Interface Layer
Related to:	Domain Model depends On Persistence Layer
Description:	Domain model (entity) classes depend on user interface classes such as windows, buttons and other widgets. Occurs if domain logic is embedded into UI classes
Motivation:	Becomes a problem if UI is replaced (e.g. desktop UI is replaced by a web UI, or another UI library is used: swing to SWT migration)
Detection:	Use rules to detect whether classes are domain or UI classes - rules define stereo types, check whether there are links between such classes. Example rules (defined using dependencies): "if a class references a class in java.awt then it is a UI class"

Antipattern Name:	Circular Dependencies between Packages
Related to:	Domain Model depends On Persistency Layer
Motivation:	Dependencies should go one way from top to bottom layer
Detection:	May use JDepend. Jung does not support for detection circular dependencies. Using JUNG's path finding algorithms find all paths from the node to itself and check if those paths contain nodes from more than one namespace.

Antipattern Name:	To Far away from Main Path
Related to:	Domain Model depends On Persistence Layer
Defined by:	http://clarkware.com/software/JDepend.html
Description:	A package's balance between abstractness and stability is not right
Detection:	May use JDepend

Antipattern Name:	Boat anchor
Also Known As:	Dead weight
Defined by:	http://sourcemaking.com/antipatterns/boat-anchor
Description:	Pieces of old code that are no longer used.
Example:	deprecated
Motivation:	Unused classes appear as disconnected clusters in the graph and affect modularity metric of the system if there is one.
Detection:	That implies that we have to recognize old code - src code analyzer watch out for @deprecated, look for unreferenced code

Appendix B

ODEM.dtd

```
<!-- ============================================== -->
<!--                                                -->
<!-- Object Dependency Exploration Model (ODEM) DTD   -->
<!--                                                -->
<!-- Author  : Manfred Duchrow                      -->
<!-- Version : 1.1 , 16/08/2007                     -->
<!--                                                -->
<!-- History                                        -->
<!-- 1.0  19/01/2007 mdu Created                    -->
<!-- 1.1  16/08/2007 mdu Added visibility isAbstrct   -->
<!--                 and isFinal to type            -->
<!--                                                -->
<!-- Copyright (c) 2007, by Manfred Duchrow.        -->
<!-- All rights reserved.                           -->
<!-- ============================================== -->

<!ELEMENT ODEM (header,context)>
<!ATTLIST ODEM
  version CDATA #REQUIRED>
```

```
<!ELEMENT header (created-by)>

<!ELEMENT context (description?, container*)>
<!ATTLIST context
  name CDATA #REQUIRED>

<!ELEMENT description (#PCDATA)>

<!ELEMENT created-by (exporter,provider)>

<!ELEMENT container (namespace*)>
<!ATTLIST container
  name CDATA #REQUIRED
  classification (dir|jar|component|unknown) #IMPLIED>

<!ELEMENT exporter (#PCDATA)>
<!ATTLIST exporter
  version CDATA #REQUIRED>

<!ELEMENT provider (#PCDATA)>

<!ELEMENT namespace (type*)>
<!ATTLIST namespace
  name CDATA #REQUIRED>

<!ELEMENT type (dependencies)>
<!ATTLIST type
  name CDATA  #REQUIRED
  classification (class|interface|annotation|enum|unknown) #IMPLIED
```

```
  visibility (public|protected|private|default) #IMPLIED
  isAbstract (yes|no) "no"
  isFinal (yes|no) "no"
>

<!ELEMENT dependencies (depends-on*)>
<!ATTLIST dependencies
  count CDATA #REQUIRED>

<!ELEMENT depends-on EMPTY>
<!ATTLIST depends-on
  name CDATA #REQUIRED
  classification (uses|extends|implements) #IMPLIED>
```

Appendix C

Junit ODEM example

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE ODEM PUBLIC "-//PFSW//DTD ODEM 1.1" "http://pfsw.org/ODEM/schema/dtd/odem-1.1.dtd">

<ODEM version="1">
  <header>
    <created-by>
      <exporter version="1.1.1">
        org.pf.tools.cda.ext.export.xml.XmlFileODEMExporter
      </exporter>
      <provider>Manfred Duchrow Consulting &amp;amp; Software</provider>
    </created-by>
  </header>

  <context name="New Workset">
    <container classification="jar" name="C:/junit.jar">
      <namespace name="junit.awtui">
        ...
        <type visibility="public" isAbstract="yes"
              classification="interface" name="junit.framework.TestListener">
          <dependencies count="4">
            <depends-on classification="extends" name="java.lang.Object"/>
            <depends-on classification="uses" name="junit.framework.Test"/>
            <depends-on classification="uses" name="java.lang.Throwable"/>
            <depends-on classification="uses" name="junit.framework.AssertionFailedError"/>
          </dependencies>
        </type>
        ...
```

Appendix D

Interfaces API

nz.ac.massey.cs.barrio.inputReader

Interface InputReader

```
public interface InputReader
```

Method Detail

read

```
edu.uci.ics.jung.graph.Graph read(java.lang.Objectinput)
                                  throws
nz.ac.massey.cs.barrio.inputReader.UnknownInputException,
                                         java.io.IOException
```

Builds the instance of the jung.graph.Graph object

Parameters:
`input` - the ODEM object or the filepath of the ODEM file
Returns:
the built graph
Throws:
`nz.ac.massey.cs.barrio.inputReader.UnknownInputException`
`java.io.IOException`

nz.ac.massey.cs.barrio.srcReader

Interface SourceReader

```
public interface SourceReader
```

Method Detail

getProjectReadingJob

```
org.eclipse.core.runtime.jobs.Job
getProjectReadingJob(java.util.List<org.eclipse.jdt.core.IJavaProject>projects)
```

Creates an EPF job that uses a list of the Eclipse Java projects. The job uses only first item from the list in order to create an ODEM object in memory.

Parameters:
`projects` - the list of Eclipse Java projects
Returns:
job that creates ODEM object in memory

getProjectReadingJob

```
org.eclipse.core.runtime.jobs.Job
getProjectReadingJob(java.util.List<org.eclipse.jdt.core.IJavaProject>projects,
                                                    java.lang.Stringfilepath)
```

Creates an EPF job that uses a list of the Eclipse Java projects and an output filepath. The job uses only first item from the list in order to create an ODEM file in the filepath.

Parameters:
`projects` - the list of Eclipse Java projects
`filepath` - the desired ODEM filepath
Returns:
job that creates ODEM file

getBuffer

```
byte[] getBuffer()
```

Returns byte array of the ODEM object

Returns:
byte array to represent ODEM object

nz.ac.massey.cs.barrio.clusterer

Interface Clusterer

```
public interface Clusterer
```

Method Detail

cluster

```
void cluster(edu.uci.ics.jung.graph.Graphgraph)
```

Removes one or several edges from the graph that have the same highest betweenness value.

Parameters:
`graph` - the graph

getEdgesRemoved

```
java.util.List<edu.uci.ics.jung.graph.Edge> getEdgesRemoved()
```

Retrieves the list of all edges that were removed by method "cluster(Graph graph)". The edges returned are stored in order in which they were removed

Returns:
java.util.List list of the removed edges.

nameClusters

```
java.util.HashMap<java.lang.String,java.lang.Integer>
nameClusters(edu.uci.ics.jung.graph.Graphgraph)
```

Assigns String attachment of the name of the cluster to the vertex objects for the purpose of identity. Cluster names are generated as follows: "cluster-0", "cluster-1", ..., "cluster-n"

Parameters:
`graph` - the graph
Returns:
the map of cluster names mapped with the cluster size.

nz.ac.massey.cs.barrio.motifFinder

Interface MotifInstance

```
public interface MotifInstance
```

This class represents a single instance of a motif. By motif we mean an element of a pattern.

Method Detail

getInstance

```
edu.uci.ics.jung.graph.Vertex getInstance(java.lang.StringroleName)
```

Retrieves jung.graph.Vertex type that acts the specified rolename

Parameters:
`roleName` - the rolename
Returns:
the jung.graph.Vertex

nz.ac.massey.cs.barrio.motifFinder

Interface Motif

```
public interface Motif
```

Method Detail

findAll

```
java.util.Iterator<MotifInstance> findAll(edu.uci.ics.jung.graph.Graphgraph)
```

Finds all motif instances that occur in the dependency graph

Parameters:
`graph` - the graph
Returns:
iterator of the intances of the motif

getRoleNames

```
java.util.List<java.lang.String> getRoleNames()
```

Gets the list of all possible rolenames in current motif

Returns:
list of rolenames

nz.ac.massey.cs.barrio.graphManager

Interface GraphManager

```
public interface GraphManager
```

Method Detail

setGraph

```
void setGraph(edu.uci.ics.jung.graph.Graphgraph)
```

Required method to run in order to use any of the other methods of GraphManager. Sets 'tree' structure of the project.

Parameters:
`graph` - instance of edu.uci.ics.jung.graph.Graph

getProjectClusters

```
java.util.List<java.lang.String> getProjectClusters(booleanshowSingletons)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`showSingletons` - true if you want to have singleton clusters, false otherwise
Returns:
List of cluster names within the project

getProjectClusterSize

```
int getProjectClusterSize(java.lang.StringclusterName)
```

setGraph(Graph graph) must be called before using following method. returns number of classes in the specified cluster

Parameters:
`clusterName` - name of the cluster
Returns:
number of classes within the cluster

getContainerClusters

```
java.util.List<java.lang.String> getContainerClusters(java.lang.StringcontainerName,
                                                       booleanshowSingletons)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`showSingletons` - true if you with to see singleton clusters, false otherwise
Returns:
List of cluster names within the container

---

getNamespaceClusters

```
java.util.List<java.lang.String> getNamespaceClusters(java.lang.StringcontainerName,
                                                       java.lang.StringnamespaceName,
                                                       booleanshowSingletons)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`namespaceName` - name of the namespace
`showSingletons` - true if you with to see singleton clusters, false otherwise
Returns:
List of cluster names within the namespace

---

getContainers

```
java.util.List<java.lang.String> getContainers()
```

setGraph(Graph graph) must be called before using following method.

Returns:
list of container names within the project

---

getNamespaces

```
java.util.List<java.lang.String> getNamespaces(java.lang.StringcontainerName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of container
Returns:
list of namespace names within the container

---

getClasses

```
java.util.List<java.lang.String> getClasses(java.lang.StringcontainerName,
                                            java.lang.StringnamespaceName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`namespaceName` - name of the namespace
Returns:
List of class names within namespace

---

getClassCluster

```
java.lang.String getClassCluster(java.lang.StringcontainerName,
                                 java.lang.StringnamespaceName,
                                 java.lang.StringclassName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`namespaceName` - name of the namespace
`className` - name of the class
Returns:
cluster name the class belongs to

---

getClassAnnotation

```
java.lang.String getClassAnnotation(java.lang.StringcontainerName,
                                    java.lang.StringnamespaceName,
                                    java.lang.StringclassName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`namespaceName` - name of the namespace
`className` - name of the class
Returns:
class parameters string

---

getProjectRuleDefinedClusters

```
java.util.List<java.lang.String> getProjectRuleDefinedClusters()
```

setGraph(Graph graph) must be called before using following method.

Returns:
List of the names of rule defined clusters within the project

---

getContainerRuleDefinedClusters

```
java.util.List<java.lang.String>
getContainerRuleDefinedClusters(java.lang.StringcontainerName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
Returns:
List of the names of the rule defined clusters within the container

---

getNamespaceRuleDefinedClusters

```
java.util.List<java.lang.String>
getNamespaceRuleDefinedClusters(java.lang.String containerName,
                                java.lang.String namespaceName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`namespaceName` - name of the namespace
Returns:
List of names of the rule defined clusters within the namespace

---

getClassRuleDefinedClusters

```
java.util.List<java.lang.String>
getClassRuleDefinedClusters(java.lang.String containerName,
                            java.lang.String namespaceName,
                            java.lang.String className)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`namespaceName` - name of the namespace
`className` - name of the class
Returns:
List of the names of the rule defined clusters the class belongs to

---

getContainerSize

```
int getContainerSize(java.lang.StringcontainerName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
Returns:
number of classes within the container

getNamespaceSize

```
int getNamespaceSize(java.lang.StringcontainerName,
                     java.lang.StringnamespaceName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`namespaceName` - name of the namespace
Returns:
number of classes within the namespace

---

getIntersectionSize

```
int getIntersectionSize(java.lang.StringcontainerName,
                        java.lang.StringclusterName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`clusterName` - name of the cluster
Returns:
number of classes within the container that belong to the cluster

---

getIntersectionSize

```
int getIntersectionSize(java.lang.StringcontainerName,
                        java.lang.StringnamespaceName,
                        java.lang.StringclusterName)
```

setGraph(Graph graph) must be called before using following method.

Parameters:
`containerName` - name of the container
`namespaceName` - name of the namespace
`clusterName` - clusterName name of the cluster
Returns:
number of classes within the namespace that belong to the cluster

---

nz.ac.massey.cs.barrio.exporter

Interface Exporter

```
public interface Exporter
```

Method Detail

export

```
void export(edu.uci.ics.jung.graph.Graphgraph,
            intseparation,
            java.util.List<edu.uci.ics.jung.graph.Edge>removedEdges,
            java.util.List<java.lang.String>filters,
            java.lang.StringfolderName)
```

Writes graph clustering result information to XML file.

Parameters:

`graph` - the graph
`separation` - separation level (number of iterations)
`removedEdges` - list of the removed edges
`filters` - list of the applied filters
`folderName` - name of the folder of the XML file

nz.ac.massey.cs.barrio.filters

Class EdgeFilter

```
java.lang.Object
  └ nz.ac.massey.cs.barrio.filters.EdgeFilter
```

All Implemented Interfaces:

edu.uci.ics.jung.graph.filters.Filter

```
public class EdgeFilterextends java.lang.Objectimplements
edu.uci.ics.jung.graph.filters.Filter
```

Constructor Detail

EdgeFilter

```
public EdgeFilter()
```

Method Detail

filter

```
public edu.uci.ics.jung.graph.filters.UnassembledGraph
filter(edu.uci.ics.jung.graph.Graphg)
```

Applies filter to the graph by removing edges that contain filter property

Specified by:

`filter` in interface `edu.uci.ics.jung.graph.filters.Filter`

Parameters:

`g` - the graph

Returns:

UnasembeledGraph, can be assembled by calling assemble() method.

---

chooseGoodEdges

```
public java.util.Set<edu.uci.ics.jung.graph.Edge>
chooseGoodEdges(java.util.Set<edu.uci.ics.jung.graph.Edge>edges)
```

Iterates over a set of edges and checks which ones will remain in new graph

Parameters:

`edges` - the set of edges

Returns:

set of edges that will remain in the new filtered graph

---

acceptEdge

`public boolean acceptEdge(edu.uci.ics.jung.graph.Edgee)`

Needs to be overridden in the subclass. Decides whether an edge is to be kept in the graph or to be removed

Parameters:
e - the edge
Returns:
true if edge remains or false if edge is removed

---

getName

`public java.lang.String getName()`

Needs to be overridden in the subclass.

Specified by:
`getName` in interface `edu.uci.ics.jung.graph.filters.Filter`
Returns:
Name of the filter

nz.ac.massey.cs.barrio.filters

Class NodeFilter

```
java.lang.Object
  └ nz.ac.massey.cs.barrio.filters.NodeFilter
```

All Implemented Interfaces:

edu.uci.ics.jung.graph.filters.Filter

```
public class NodeFilterextends java.lang.Objectimplements
edu.uci.ics.jung.graph.filters.Filter
```

Constructor Detail

NodeFilter

```
public NodeFilter()
```

Method Detail

filter

```
public edu.uci.ics.jung.graph.filters.UnassembledGraph
filter(edu.uci.ics.jung.graph.Graphg)
```

Applies filter to the graph by removing vertices that contain filter property

Specified by:

`filter` in interface `edu.uci.ics.jung.graph.filters.Filter`

Parameters:

`g` - the graph

Returns:

UnasembeledGraph, can be assembled by calling assemble() method.

---

chooseGoodVertices

```
public java.util.Set<edu.uci.ics.jung.graph.Vertex>
chooseGoodVertices(java.util.Set<edu.uci.ics.jung.graph.Vertex>verts)
```

Iterates over a set of vertices and checks which ones will remain in new graph

Parameters:

`verts` - the set of vertices

Returns:

set of vertices that will remain in the new filtered graph

---

acceptVertex

```
public boolean acceptVertex(edu.uci.ics.jung.graph.Vertexv)
```

Needs to be overridden in the subclass. Decides whether a vertex is to be kept in the graph or to be removed

Parameters:
v - the vertex
Returns:
true if vertex remains or false if vertex is removed

---

getName

```
public java.lang.String getName()
```

Needs to be overridden in the subclass.

Specified by:
getName in interface edu.uci.ics.jung.graph.filters.Filter
Returns:
Name of the filter
