On the Classification of Cyclic Dependencies in Java Programs

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

Master of Science
in
Computer Science

at Massey University, Manawatū, New Zealand

Hussain Abdullah A. Al-Mutawa

2013

ABSTRACT

Software engineering guidelines and rules discourage cyclic dependencies between modules, yet empirical studies have shown that many software systems are burdened with them. This might indicate that not all cycles are as detrimental to software quality as previously thought. Clearly, a better understanding of the types of cyclic dependencies and their effect on software quality is required. As a first step in this direction, we look closely at the shapes formed by software dependency graphs containing cyclic dependencies. Such cyclic dependencies correspond to the concept of strongly connected components in graph theory. We propose an approach to classify strongly connected components according to their topologies. This allows us to distinguish between dense and sparse, symmetric and asymmetric structures. We extend on previous studies and investigate the relationship between cyclic dependencies and the package containment tree. We validate our approach with experiments based on a corpus of 103 open–source Java systems. We find that cyclic dependencies tend to form in branches of the package containment tree around parent packages that are not critical according to some researchers.

ACKNOWLEDGEMENTS

I am in debt to my advisors, Jens Dietrich, Catherine McCartin, and Stephen Marsland for supervising me throughout the course of this thesis. This thesis would not have been possible without their guidance and continual help. When I had been doing my undergraduate degree, I once asked Jens: *"why cyclic dependency matters?"* Jens's response was: *"why don't you do some postgraduate research with me and find out for yourself?"* My deepest thanks goes to Jens for motivating me towards doing this research and for his continual and invaluable help in putting this thesis together. My thanks also goes to Catherine and Stephen for being tolerant and supportive and for the warm encouragement and guidance they provided.

My sincere thanks also goes to Janet George, Russell Johnson and Briony Coote for their help in proofreading this thesis.

Last but not least, my heartfelt thanks goes to my wife, for her kindness and support she has shown during the past eighteen months it has taken me to finalise this thesis. I would also like to thank my daughters Ayah and Alaa, and my son Ali, for their patience during my study. Furthermore, I would like to offer my special thanks to my parents for teaching me how to be persistent.

This project has been supported by funding from the Ministry of Higher Education, Saudi Arabia as part of the King Abdullah Foreign Scholarship Program (KASP).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ADP Acyclic Dependency Principle

ANTLR ANother Tool for Language Recognition

ASPL Average Shortest Path Length

AWT Abstract Window Toolkit

DAG Directed Acyclic Graph

DFS Depth First Search

JAR Java Archive

JRE Java Runtime Environment

JSON JavaScript Object Notation

LOC Lines of Code

MFES Minimum Feedback Edge Set

ORM Object-Relational Mapping

OSI Open System Interconnection

PCT Package Containment Tree

PDG Package Dependency Graph

SCC Strongly Connected Component

SVM Support Vector Machine

UML Unified Modelling Language

CHAPTER 1

INTRODUCTION

❝ *The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.* ❞
[Brooks]

Contents

Summary – This chapter provides a discussion of the causes and effects of the increasing complexity of software systems. We discuss some of the preventive measures that can be taken to avoid, cure and tame the growing complexity of software systems. Then we state the research questions and present the thesis outline.

1.1 The Growing Size and Complexity of Software

Software systems grow in size and complexity with time [1, 2, 3]. Brooks claims that software and complexity are not separable [4]. Lehman [5, 6, 7] suggests that the complexity of a software system is directly proportional to its size [8]. For instance, the size of Microsoft's Windows operating system has increased from 5 million Lines of Code (LOC) to 50 million LOC between the years 1993 and 2003 (Figure 1.1). Similarly, the size of the development and testing teams has increased rapidly with a rate of more than 100 developers per year, which suggests that the cost of maintaining the system has increased accordingly [9, 10].

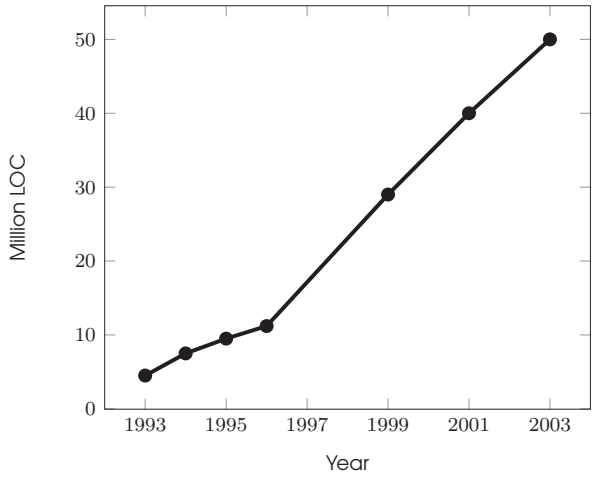


Figure 1.1 *Microsoft Windows operating system size over a decade. The number of LOC increased by a rate of approximately 4.5 million LOC per year, which is almost equal to the initial size.*

A software system may start with a simple design, clear architecture and, most importantly, a specific purpose. However, as various developers of different experience levels work on the same project, requirements change often and issues are being addressed to resolve problems in the design; the whole structure of the system starts to get complicated and signs of decay start to appear [11]. Features that were not part of the main purpose of the system are added. In addition, enhancements and upgrades are carried out as a response to compatibility issues imposed by changes in the operating system or virtual machine. The design gradually moves from simplicity to complexity, unless some work has been devoted to maintaining the system's architecture [7].

The relationships between software components can be analysed and visualised using a simple model from Graph theory[1]. A dependency graph $G = (V, E)$ is a finite discrete structure composed of a set of vertices V, which represents the components of the system and a set of edges $E \subseteq V \times V$, which represents the relationships between those components (Figure 1.2). For simplicity, in this thesis the term 'graph' means "directed graph". A Java program is a collection of classes that can be grouped into packages. The PDG is a directed graph whose vertices are packages and edges are the dependencies between packages. Package P_1 depends on package P_2, i.e $P_1 \rightarrow P_2$, if there is at least one reference from a class in P_1 to a class in P_2. A package is not dependent on itself, i.e. $P_i \not\rightarrow P_i$.

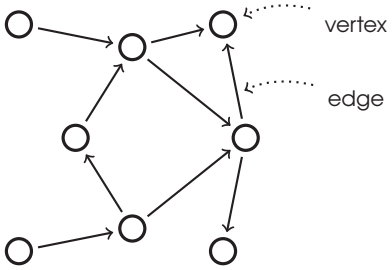


Figure 1.2 *A small example graph with eight vertices and nine edges*

In order to illustrate the evolution of size and complexity in software systems, two open–source systems – ANother Tool for Language Recognition (ANTLR)[2] and Hibernate have been investigated. ANTLR started with few packages in its early phases of development. However, the number of packages in the system has increased substantially since version 2.7.2. Figure 1.3 shows the PDGs of some selected versions of ANTLR starting from 2.4.0 to 3.2. After every release, the system increases not only in size, but also in complexity. Hibernate is an Object-Relational Mapping (ORM)[3] library that helps developers map between Relational Database Management System (RDBMS) entities, i.e. tables and views, and classes. Hibernate started with a relatively simple design composed of eight packages and then progressively became very complex with 76 packages (Figure 1.4).

[1]Graph theory is the study of graphs which are mathematical structures used in modelling the relationships between objects. A graph is made of nodes or vertices that are linked by edges or arcs [12].

[2]ANTLR is a language tool that provides a framework for constructing recognisers, compilers, and translators from grammatical descriptions containing Java, C++, or C# actions [13].

[3]ORM is a programming technique for converting data between incompatible type systems in object–oriented programming languages.

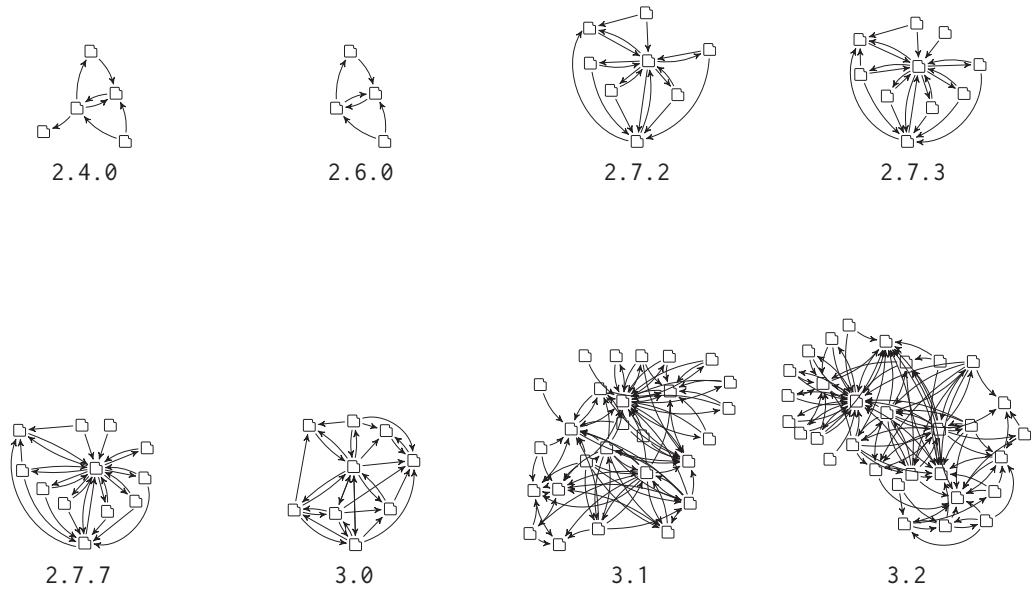


Figure 1.3 *PDGs of different versions of ANTLR. The number under each graph represents the version number of the system.*

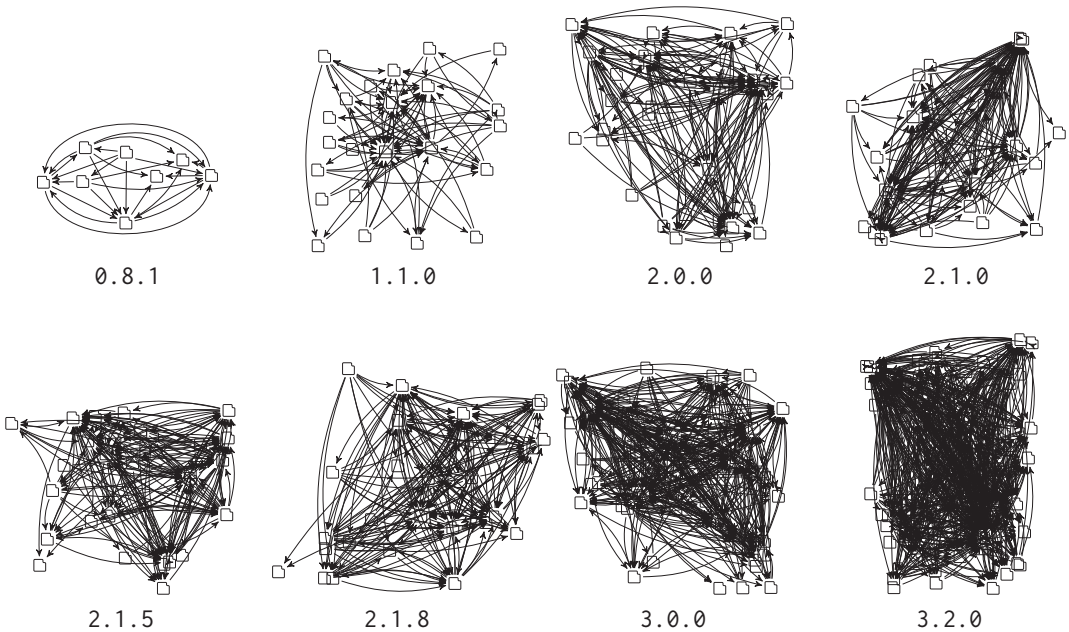


Figure 1.4 *PDGs of different versions of Hibernate. The number under each graph represents the version number of the system.*

1.2 Coping with Complexity

Many solutions have been proposed to cope with the increasing complexity of software systems as they evolve. Many can be seen as constraints that restrict the dependencies between system modules in some way.

1.2.1 Layered Design

Dijkstra [14] and Parnas [15, 16, 17] suggest organising software systems in layers. A well known example of a successful layered architectural design is the Open System Interconnection (OSI) network model [18, 19]. Although the OSI model is not entirely an integrated software system, its strength tends to come from the enforcement of abstraction between layers. For example, the presentation layer is not concerned with how the connection to the other end is established, how data packets are transferred or what connection medium is used in transmission.

Organising software components into layers is not sufficient by itself. The flow of information has to be managed as well. For instance, cyclic and bypassing references are two violations of proper layered design (Figure 1.5). A layer should only know its immediate neighbours. The flow of dependencies should move in one direction and does not make a cyclic path. In graph theory, this kind of graph is called a Directed Acyclic Graph (DAG).

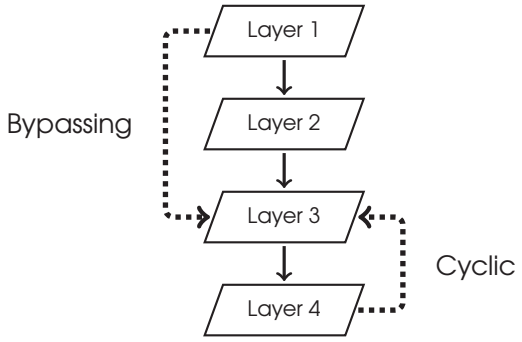


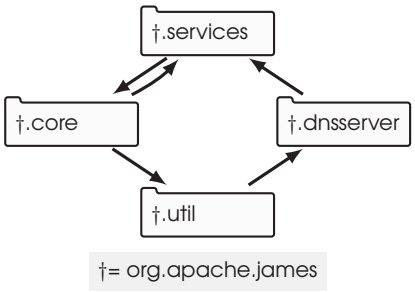
Figure 1.5 *System modules should be organised into layers where every layer provides a specific functionality. Dotted arrows represent some violations of proper layered design.*

1.2.2 The Acyclic Dependency Principle

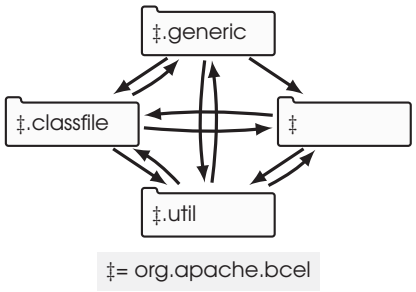
One of the widely established design principles dates back to early research in software design [17, 20] and mandates that software should be organised into modules with managed dependencies. For instance, Parnas suggests that modules should be

organised in a hierarchy based on dependency relationships, thereby keeping dependencies "loop free" [20]. In object-oriented programming, this is known as the Acyclic Dependency Principle (ADP): *"The dependencies between packages must not form cycles."* [21].

Dependency graphs may contain a group of components that are highly tangled, i.e. that are in a cyclic dependency. A tangled component is equivalent to the graph theoretic concept of a strongly connected component (SCC). A graph is *strongly connected* when there is a directed path between any two vertices in the graph. The shape of SCCs varies from one dependency graph to another. For instance, consider these two extreme cases: (a) a simple cycle where each package depends exactly on another package, which forms a circle, and (b) a clique where each package depends on all other packages within the clique. Figure 1.6 shows an actual simple circle and a clique formed by cyclic dependencies extracted from James and JREFactory systems. The removal of a single link breaks the circle structure. However, in order to break the clique structure, many links need to be chosen and removed. Therefore, we are interested with investigating the topology and the size of tangles, and aim to classify different types of tangles so that their effect on software quality can be evaluated.



(a) A circle tangle from James-2.2.0



(b) A clique tangle from JREFactory-2.9.19

Figure 1.6 *A comparison between sparse and dense cyclic dependencies extracted from James and JREFactory systems.*

Although the ADP discourages cycles in dependencies, many large software systems are riddled with them [22, 23] including the Java Runtime Environment (JRE) [24]. Figures 1.7 and 1.8 show the same dependency graphs presented earlier in this chapter for ANTLR and Hibernate systems with emphasis on the proportion of relationships between packages that belong to cyclic dependencies. More information about ANTLR and Hibernate versions statistics can be found in Appendix B.

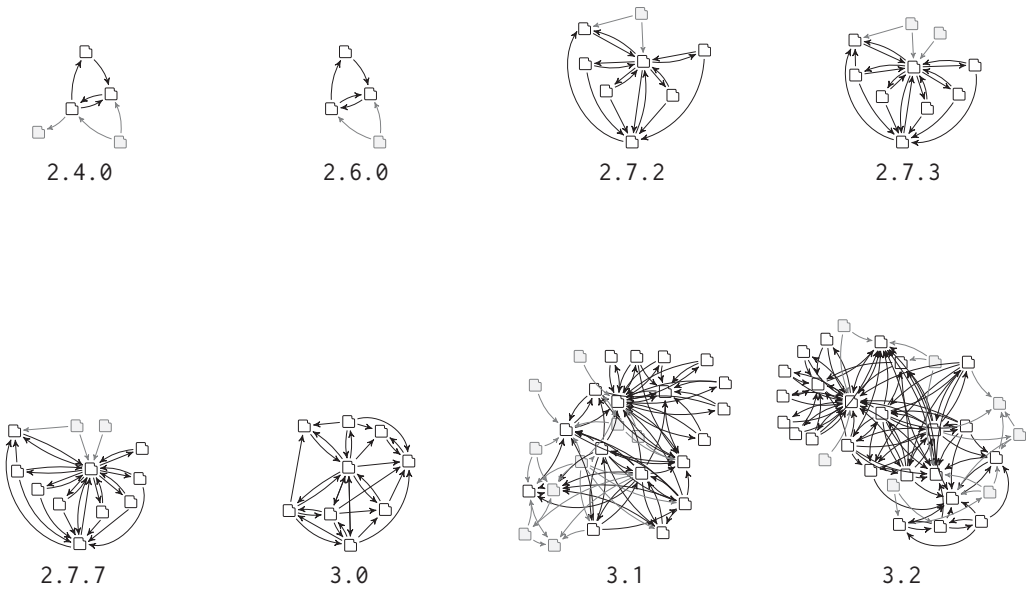


Figure 1.7 *PDGs of different versions of ANTLR with emphasis on cyclic dependencies (bold edges).*

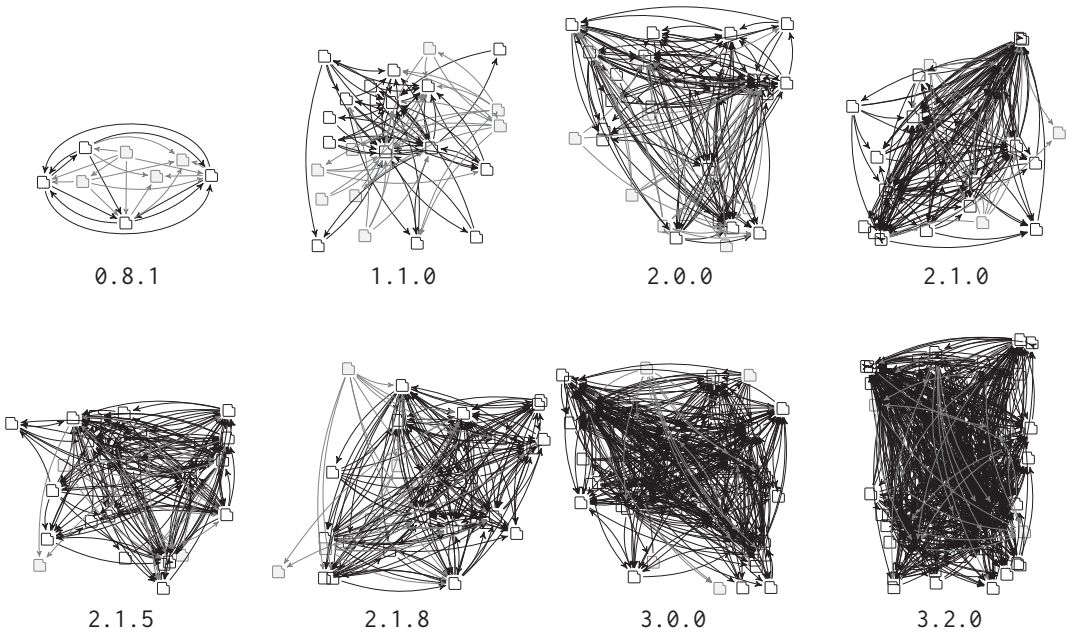
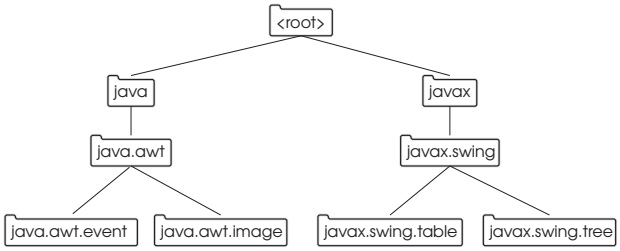


Figure 1.8 *PDGs of different versions of Hibernate with emphasis on cyclic dependencies (bold edges).*

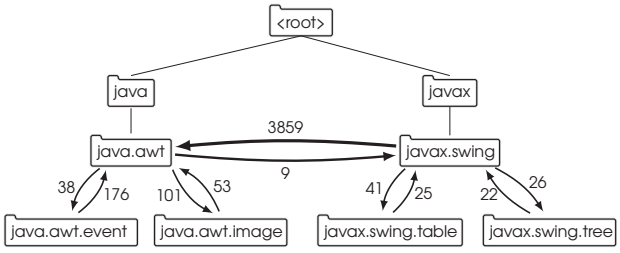
1.2.3 The Acyclic Dependency Principle and the Package Containment Tree

In Java, packages can be organised into hierarchical levels (tree structure). The tasks done by a package are distributed among different levels. Sub-packages at the bottom of the tree can have a more specific functionality. For instance, the `javax.swing` package provides a set of lightweight platform-independent user interface widgets. The package `javax.swing.event` provides event handlers for actions performed by Swing components. In this example, a cyclic reference can result because a component fires events and an event handler manages events fired by a component. This cyclic reference is tolerable, and in fact may be desirable, from the point of view of the developers [25]. The package containment tree (PCT) is a graph that shows the relationships between packages and their sub-packages (Figure 1.9).



`Figure 1.9` *A partial PCT of* `java.awt`, `javax.swing` *and some of their sub-packages.*

The JRE contains many cyclic dependencies that occur between packages that are not nested under the same parent package [26]. In particular, the `java.awt` and `javax.swing` packages depend on each other (Figure 1.9). A cyclic dependency that passes through the root, such as the cyclic dependency between `javax.swing` and `java.awt`, is less desirable than a cyclic dependency formed between a package and its sub-packages, e.g. `javax.swing` and `javax.swing.tree` (Figure 1.10) [25].



`Figure 1.10` *A partial PCT of* `java.awt`, `javax.swing` *and some of their sub-packages. The labels on edges represent the number of class-to-class dependencies made from one package to another.*

1.3 Research Questions

Cyclic dependency in software design has attracted attention for a long time. Cyclic dependencies have been considered as a design defect that inhibits modularity and affects software quality, yet many well-known software systems are known to have this design anomaly. This implies that cyclic dependencies may not be as detrimental to software quality as previously thought. Some of the detected cycles are very dense and large while others are small, simple and can be easily understood. Splitting the tasks performed by a package to its sub-packages may introduce some kind of cyclic references. Therefore, the shape of cyclic dependencies might be affected by the PCT. This leads to our research questions:

- *What are the shapes that cyclic dependencies form?*
- *Can we construct an efficient and robust algorithm to classify strongly connected components according to their shape?*
- *What is the relationship between cyclic dependencies and the package containment tree?*

1.4 Thesis Outline

In Chapter 2, we present a background of the problem of cyclic dependencies and a review of the existing literature. In particular, we present some of the existing metrics and methods used to detect and classify cyclic dependencies. In Chapter 3, we provide a study of levels of dependencies in Java programs and the way in which dependency graphs and different types and classes of tangles are constructed. The methodology followed to classify tangles is presented in Chapter 4, where we explain the experiments we conducted to build the tangles classifier and describe the set of metrics that we considered in order to classify tangles and identify the computational complexity of these metrics. We discuss the robustness of analysis methods we used to ensure that the metrics we used are resistant to the size or the orientation of tangles. We present the results and findings of the classification algorithm on a corpus of 103 open–source Java programs. In Chapter 5, we try to answer the last research question related to the package containment tree and cyclic dependency. Finally, we summarise our findings and discuss some of the possible future directions of our research in Chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

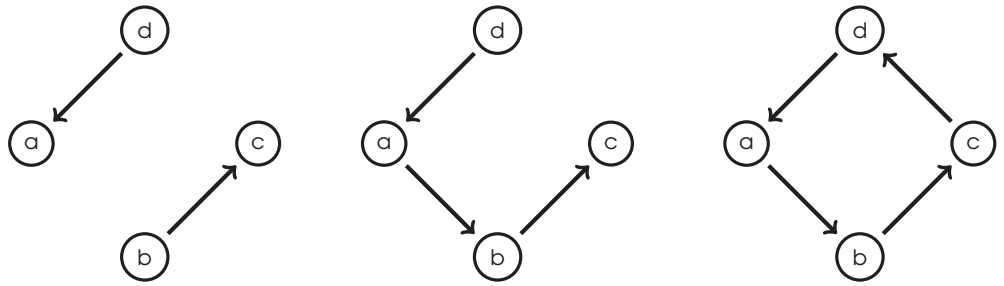
❝ *In the beginner's mind there are many possibilities, in the expert's there are few.*❞
[Shunryu Suzuki]

Contents

Summary – This chapter presents background and a review of the work reported in the literature related to our research. In each section, definitions are provided, followed by relevant theories and applications. Briefly, the following topics are discussed: cycles in directed graphs and their properties which can be extracted using some standard metrics from graph theory; the package containment tree and its relationships, and its significance in studying cyclic dependencies on the package dependency level; previous work on classifying graph topologies and identifying patterns in software dependency graphs. Finally, the effects of cyclic dependencies on refactoring is discussed from the point of view of other researchers.

2.1 Cycles in Directed Graphs

A *directed graph* $G = (V, E)$ consists of a set of vertices V and a set of edges E, where edges have directions associated with them. For simplicity, whenever the term 'graph' will be used later on, it refers to a 'directed graph'. An *edge* is an ordered pair $(v, w) \in V \times V$ where v and w are called the endpoints of the edge. A *directed path* is a sequence of vertices $\{v_1, v_2, \ldots, v_n\} \subseteq V$ such that $(v_i, v_{i+1}) \in E$. A path is simple if it contains no repeated vertices. A *cycle*, a.k.a. *circuit*, is a path which starts and terminates at the same vertex. A cycle is simple or *elementary* if every vertex in it is visited exactly once. Reversing the direction of every edge in a graph $G = (V, E)$ results in its *transposed* graph $G^T = (V, \{(b, a) \mid (a, b) \in E\})$. A graph G is called a *connected component* if there exists a path between any pair of vertices in $G \cup G^T$. A graph is called *disconnected* if it is composed of more than one component. A *strongly connected component (SCC)* is a maximal subset of vertices if a path exists in both directions between every pair of vertices in the subset [27] (Figure 2.1).



(a) Disconnected (b) Connected (c) Strongly connected

Figure 2.1 *Three families of directed graphs, namely: disconnected, connected and strongly connected.*

Software modules and the dependencies between them can be modelled as a directed graph whose vertices represent the various modules in the system and whose edges represent the dependencies between modules. The dependency takes place when a module references another one. Dependencies in Java classes[1] can be divided into two categories which are: `extends` and `uses` [23]. Listing 2.1 and 2.2 illustrate the difference between these two dependencies using an example for each category. However, we do not make distinction between these two categories in our study. A set of modules are considered in "cyclic dependency" if there exists a dependency path that starts and terminates at the same module. Cyclic dependencies can be detected by the presence

[1]We use the term Java class to refer to units of byte code, not source code. This notion also includes interfaces, enumerations and annotation types.

of SCCs in the dependency graph. Consider a graph $G = (V, E)$, a *tangle* $T \subseteq G$ is a maximal SCC that represents software modules and their dependencies. Figure 2.2 shows an example of a connected graph composed of 8 vertices and 14 edges. The removal of edges that do not make cyclic paths $\{ (b,c), (b,f), (e,f), (c,g), (h,g) \}$ results in three subgraphs which are strongly connected *'tangles'*.

```
class A extends B {   // extends reference
   ...
}
```

Listing 2.1 *An example of extends reference*

```
class A{
    public A(){
       B.doSomething(); // uses reference
    }
}
```

Listing 2.2 *An example of uses reference*

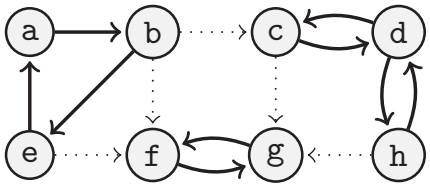


Figure 2.2 *A dependency graph that contains three tangles which are* $\{ a,b,e \}, \{ f,g \}, \{ c,d,h \}$. *Dotted edges are disregarded because they do not make cyclic paths.*

Decomposing a directed graph into SCCs is a fundamental graph theoretic problem [28]. Computing chains of recurrent sets [29], data-flow analysis [30], and compiler optimisation [31] are some of the applications of the study of SCCs. Several algorithms have been proposed to decompose directed graphs into SCCs using a path-based approach [32, 33, 34, 35, 36]. Path-based algorithms use Depth First Search (DFS) in combination with two stacks; one to keep track of visited vertices and one to keep a record of the path traversed.

Tarjan presented an algorithm that enumerates a list of SCCs of a graph in linear time $\mathcal{O}(|V| + |E|)$ [37]. Unlike path-based SCC decomposition algorithms, only one stack is used and vertices are labelled in the order they are visited. Although several algorithms and heuristics have been proposed to parallelise and speed up decomposing

a graph into SCCs [38, 39, 40, 41, 38, 42, 43, 44] and [45, 46, 28, 47], we found that Tarjan's algorithm is adequate for the type of graphs that we deal with.

The term *"coupling"* is used to denote the degree to which a software module is dependent on another. High coupling may indicate a higher chance of change propagation (ripple effect) [48]. If a dependency path is acyclic, then the change applied to one module can propagate in one direction and eventually stops at some module. On the other hand, when a dependency path is cyclic, the change effect may entail visiting a module more than once. Therefore, dependencies between software modules need to be managed to avoid cyclic paths. Fowler suggests that if a change in a module requires changes in another, then they are highly coupled [49]. Fowler states that code duplication, which is considered a bad code "smell" [50], can result in high coupling between modules. Ambler et al. [51] argue that if a system contains two or more packages in cyclic dependency, then a change in one package may enforce the change in another. Therefore, cyclic dependencies make the system fragile and hard to test.

2.2 Standard Metrics from Graph Theory

This section presents some standard measures and metrics for quantifying graph structure which are derived from graph theory and have wide applications, such as in the study of social networks. We look for metrics that are computationally efficient and resilient to minor changes of the graph structure. For each metric presented, a definition is provided, and its significance in detecting the shape of graphs and its computational complexity are discussed. More metrics that we have developed are discussed in Section 4.3.

2.2.1 Density (DENSE)

A single component graph is considered ***dense*** if its number of edges is close to the maximum possible number of edges. In contrast, a single component graph is considered ***sparse*** if its number of edges is close to the minimum number of edges required for it to be connected. If the number of edges compared to the number of vertices is considered as a measure of graph density, then the ratio $\delta(G) = |E|/|V|$ can be used as a measure of the graph's density. The problem with the ratio is that it is not normalised. The maximum number of edges possible occurs in cliques where every vertex is linked to every other vertex in the graph. Hence, the density of a clique is $\delta_{max}(G_{clique}) = |V| - 1$. On the other hand, the minimum density for the same size graph occurs in simple circles, which is $\delta_{min}(G_{circle}) = 1$. In order to normalise the density, we can apply the min-max normalisation as suggested by Shah et al. [52]:

$$DENSE(G) = \frac{|E| - |V|}{|V|^2 - 2 \times |V|} \tag{2.1}$$

2.2.2 Ratio of number of vertices to number of edges

The ratio of a tangle (RATIO) is the number of vertices relative to edges ($|V|/|E|$). The complexity of both RATIO and DENSE is $\mathcal{O}(1)$ if the number of vertices and edges are known, and $\mathcal{O}(|V| + |E|)$ if they must be counted. Figure 2.3 shows a comparison between RATIO and DENSE for some tangle shapes.

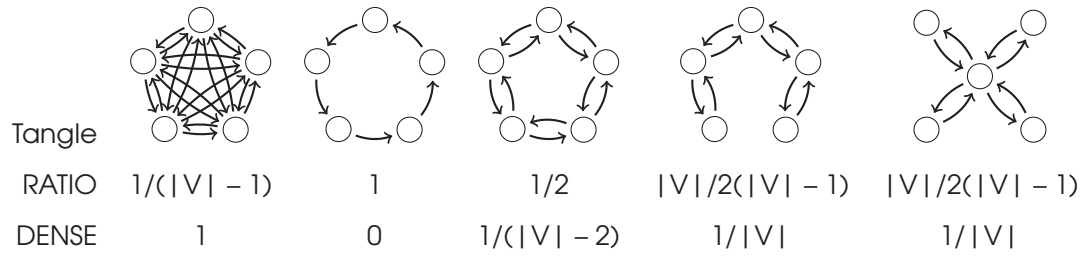


Figure 2.3 *The relationship between RATIO, DENSE and the shape of tangle.*

2.2.3 Minimum Feedback Edge Set

The Minimum Feedback Edge Set (MFES) is the minimum cardinality of a set of edges whose removal makes a graph acyclic. This measure can be useful in detecting some topologies formed by SCCs. For instance, the MFES can indicate the degree of complexity of a network. In particular, the MFES is useful in determining the number of dependencies that need to be broken in order to resolve the issue of cyclic dependency. Figure 2.4 shows different cyclic directed graphs of the same size[2]. Dense graphs have higher MFES sizes than sparse graphs.

The problem of finding the minimum feedback edge set is known to be NP-hard [53, 54, 55, 56]. There have been many heuristics and algorithms that can provide approximations to solve the MFES problem [55, 57, 58, 59]. However, we decided not to consider this metric because of its complexity.

[2]We use the number of vertices in a graph $|V|$ to measure its size.

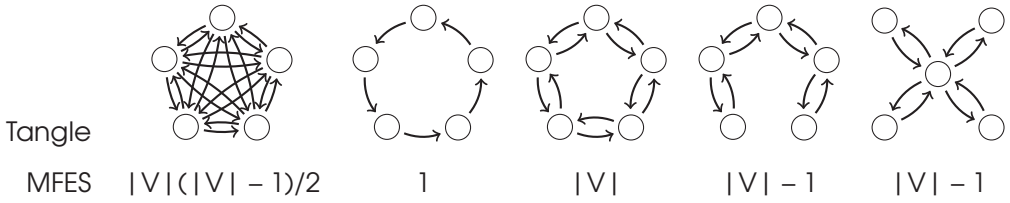


Figure 2.4 *The relationship between MFES and the shape of tangle.*

2.2.4 Mean Degree Centrality (DEG)

The *centrality* of a vertex can be defined as a measure of its relative importance within a graph. The degree of a vertex is the number of edges which are incident on it. Let $G = (V, E)$ be a graph. The *indegree* of a vertex $v \in V$ denoted $d^-(v)$ is the number of edges that terminate at v. On the other hand, the *outdegree* of a vertex $v \in V$ denoted $d^+(v)$ is the number of edges that originate from v. The degree of a vertex $v \in V$ denoted $d(v)$ is the sum of its indegree and outdegree. A vertex is called *sink* if its outdegree is zero. In contrast, a vertex is called *source* if its indegree is zero. A strongly connected component contains neither sinks nor sources. The mean degree of a graph can be computed by using this formula $DEG(G) = \sum_i^n d(v_i)/n$, where $d(v_i)$ denotes the degree of vertex v_i. DEG is $\mathcal{O}(|V|)$ in dense graphs, and is $\mathcal{O}(1)$ in sparse graphs (Figure 2.5). The complexity of DEG is $\mathcal{O}(|V| + |E|)$.

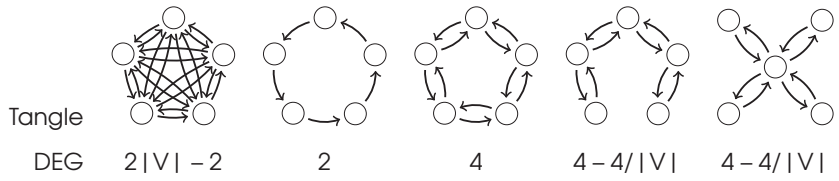


Figure 2.5 *The relationship between DEG and the shape of tangles.*

2.2.5 Diameter (DIAM)

The distance between two vertices in a graph is the length of the shortest path between them. In different applications, every edge is associated with a numerical value which is refered to as the cost or the weight of the edge. An *unweighted graph* is a graph $G = (V, E)$ such that each edge weight is 1. In unweighted graphs, the distance is the cardinality of the path connecting a pair of vertices. The distance in weighted graphs is the sum of the edges' weights in the path. The *diameter* is the longest distance between any pair of vertices [27]. The diameter is considered one of the fundamental

topological parameters of real world networks [60]. Note that DIAM is ∞ if the graph is disconnected [61, 62]. DIAM tends to be large in sparse graphs. In contrast, DIAM is close to 1 in dense graphs. The complexity of computing DIAM is $\mathcal{O}(|V||E|)$ using the all-pairs shortest path algorithm (APSP)[3] [60]. DIAM can be calculated in $\mathcal{O}(|V|^3)$ with Floyd-Warshall's algorithm [63]. Johnson's algorithm [64] can be used to calculate DIAM which runs in $\mathcal{O}(|V|^2 \log |V| + |V||E|)$.

2.2.6 Longest Path Length (LONG)

LONG is the length of the longest geodesic (simple) path that can be found in a graph. Small values for LONG indicate that a tangle is compact and dense. Note that LONG is different from DIAM in dense graphs, but is similar in sparse graphs (Figure 2.6). The longest possible path length is $|V| - 1$. Finding the longest path length is known to be NP-Hard [65]. Thus, we do not consider this measure in our study.

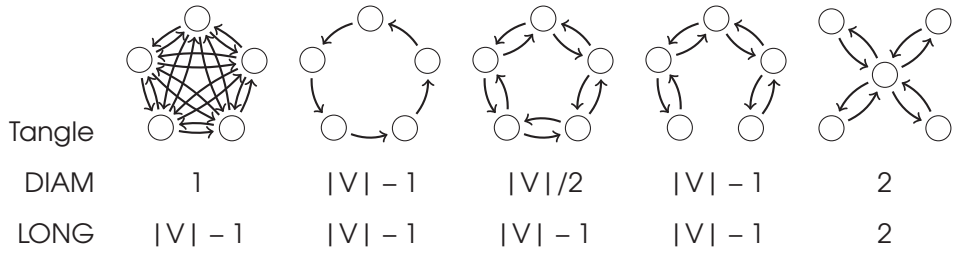


Figure 2.6 *The relationship between DIAM, LONG and the shape of tangles.*

2.2.7 Betweenness Centrality

Betweenness is a measure of network centrality that counts the paths between vertex pairs in a graph that pass through a given vertex/edge. The study of betweenness centrality is a fundamental measurement concept for the analysis of social networks [66]. Betweenness centrality can be seen as a measure of the influence a vertex has over the spread of information throughout the network [67]. Let $G = (V, E)$ be a graph, and $v, s, t \in V$. The betweenness centrality of v is the number of shortest paths between s and t that pass through v, relative to the total number of shortest paths between s and t, over all pairs s and t such that $s \neq v \neq t$. In a SCC, there exists a path between any pair $s, t \in V$ such that $s \neq v \neq t$; therefore, the total number of shortest paths we need to consider is $(|V| - 2)(|V| - 1)$. The betweenness centrality of edges can be calculated similarly. Let $\sigma_{s,t}$ be the total number of shortest paths between the vertices s

[3]This algorithm is an approximation one.

and t. Also, let $\sigma_{s,t}(v)$ be the number of those paths that pass through v, and then the betweenness centrality of a given vertex v is computed as follows:

$$B_C(v) = \sum_{s \neq v \neq t} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} \tag{2.2}$$

Freeman [66] proposed some formulae that can be used to measure the centrality of undirected graphs (Equation 2.2). White et al. generalised the concept of betweenness centrality to directed graphs [68]. Brandes proposed a method to calculate betweenness centrality whose complexity is $\mathcal{O}(|V|^2 log|V| + |V||E|)$ [63]. Newman [67] claims that the betweenness centrality can be calculated for all vertices in time $\mathcal{O}(|V||E|)$.

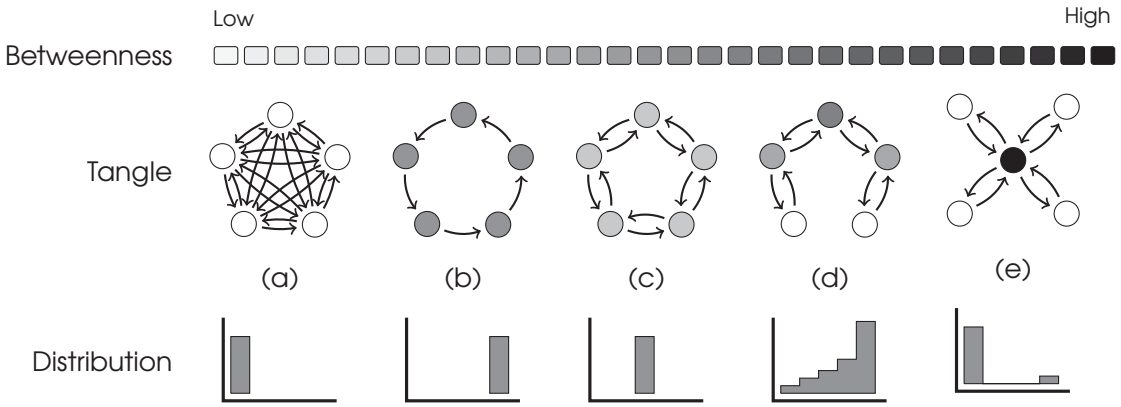


Figure 2.7 *The relationship between vertex betweenness and the shape of tangles. The darker vertices are those of high betweenness score. It can be seen that the average betweenness and standard deviation is affected by the shape of the tangle.*

The betweenness centrality measure can help in determining the shape of a tangle by testing the skewness in betweenness distribution (Figure 2.7). In a like manner, the inequality of betweenness scores can be used as an indicator of the shape of the tangle. Specifically, the inequality is represented by the presence of a few vertices in a tangle of very high betweenness relative to other vertices. There are several metrics used to detect and assess the inequality of distributions. For instance, the range [69] and the range ratio, the McLoone index [70], the coefficient of variation [71], the Gini coefficient [69, 72, 73], the Theil index [74], the Palma ratio [75], the Hoover index [76], the relative mean deviation [69], the variance and log variance [69]. An inequality measure of choice has to be widely applicable, preferably applied to software metrics in the past, and simple to compute.

2.2.8 Tangledness (TANGL)

TANGL is based on the average of back-reference path lengths between every pair of adjacent vertices. According to the definition of SCCs, if there is a directed edge from s to t in a tangle, then there is at least one simple path from t to s (*back-reference*). The average length of such paths (BRPL) has a minimum of 1, and maximum of $|V| - 1$. TANGL can be measured using Equation 2.3 as suggested by [52]. "Untangledness" is defined as the min-max normalisation of the average back-reference path length, and tangledness is the complement (1 – untangledness).

$$TANGL(G) = 1 - \frac{BRPL - 1}{|V| - 2} \tag{2.3}$$

Note that DENSE and TANGL are not always correlated. For instance, a star-like structure with a central hub vertex, similar to the tangles in the class graph created by anonymous inner classes[4], will have a high TANGL value (the value will be 1, unless nested inner classes are used), but will have a relatively low DENSE value, since all of the edges connecting the outer vertices (representing references amongst inner classes) are missing. In order to compute TANGL, we can proceed as for DIAM – simply computing all-pairs shortest paths using the Floyd-Warshall algorithm, or Johnson's algorithm, and then extract the paths that correspond to the pairs of adjacent vertices. The running time using this approach is upper bounded by $\mathcal{O}(|V|^3)$ [77].

2.2.9 Transitivity (TRANS)

TRANS measures the ratio between connected triples with "shortcuts" between the first and the last vertex. It corresponds to the clustering coefficient in undirected graphs [27]. A closed path is a sequence of vertices $(v_1, v_2, \ldots, v_n)$ where adjacent vertices are connected by edges $(v_k, v_{k+1}) \in E$ and the first vertex is also connected to the last vertex $(v_1, v_n) \in E$. Transitivity is then defined as the ratio between the number of closed paths on three vertices and the number of all simple paths connecting three vertices. Alternatively, TRANS can be defined by counting the number of triangles and triples in the graph. A *triangle* $\triangle = (V_\triangle, E_\triangle)$ of graph $G = (V, E)$ is a subgraph of three vertices $V_\triangle = \{v_1, v_2, v_3\} \subset V$ and $E_\triangle = \{(v_1, v_2), (v_2, v_3), (v_1, v_3)\} \subset E$. A *triple* $\tau = (V_\tau, E_\tau)$ of a graph $G = (V, E)$ is a graph of three vertices such that $V_\tau = \{v_1, v_2, v_3\} \subset V$ and $E_\tau = \{(v_1, v_2), (v_2, v_3)\} \subset E$. Let $\triangle(G)$ and $\tau(G)$ denote the number of triangles and the number of triples of a graph G respectively. Then the transitivity of the graph can be computed using the formula stated in Equation 2.4 [78]. TRANS measures the level

[4]The definition of inner classes and anonymous inner classes will be provided with some examples in section 3.1.3

of robustness of a tangle. If an edge between vertices in a closed path of three vertices is removed (e.g., through refactoring of the program), the dependency between the first and the last element of the triple will not be broken. We can compute TRANS in time $\mathcal{O}(|V|^3)$ by considering the two-hop neighbourhood of each vertex in turn.

$$TRANS(G) = \frac{3\triangle(G)}{\tau(G)} \tag{2.4}$$

2.2.10 Size of the Automorphism Group (AUTO)

AUTO measures the degree of symmetry of a tangle. Both simple cycles and cliques have relatively large automorphism groups, indicating that each of these topologies has a very regular, symmetric structure. For a simple cycle, the size of the automorphism group is $|V|$, and each element in the group corresponds to a "rotation" of the structure, which maps each vertex to another some fixed distance further on in the cycle. For a clique, the size of the automorphism group is $|V|!$, and every vertex permutation corresponds to an element of the group. The problem of constructing the automorphism group is in NP. Although there is a polynomial time algorithm for solving the graph automorphism problem for graphs where the vertex degrees are bounded by a constant [79], a significant drawback of using AUTO as a metric for classifying tangles in our context is that it is very sensitive to even small mutations in the structure. Adding or removing a single edge in a simple cycle or a clique, for example, will result in a reduction of the automorphism group size by a factor of $|V|$. For this reason we opted not to employ this measure in our study.

2.3 The Package Containment Tree

Software systems have different levels of packaging that are used to organise and sort modules that interact together to build a larger system. A *package* is a bundle of some resources and compilation units. Packages are needed to prevent naming conflicts in the context of large class libraries not managed by a central authority. In addition, the visibility and accessibility of some classes can be restricted to immediate neighbour classes. Gosling et al. [80] suggest that packages should be organised in a hierarchical structure. According to the naming conventions in Java specifications [80], a package name starts with the organisation reversed domain such as `com.sun` or `org.wikipedia`. A package `S` is considered a sub-package of another package `P` if its fully qualified name is `P.S`.

Grouping classes into packages [5] has been adopted to organise the program into "logical units" [2]. The package containment tree (PCT) is a hierarchical structure that shows the relationships between packages and its sub-packages.

The relevance of PCT to our research lies in the following points:

1. it has been assumed that the majority of cyclic dependencies on the package level are within branches of the PCT [25].
2. the shape of the PCT may have some influence on the shape of cyclic dependencies formed by the packages in the tree.

Laval et al. presented an approach to rank tangles by their level of undesirability [25]. This is based on the assumption that tangles that consist of packages with similar names (probably because they are close in the package containment tree) are less critical than tangles consisting of packages that are far apart in the package containment tree. We will demonstrate in Chapter 5 that tangles tend to form within branches of the PCT, by measuring the relative closeness of packages common to single tangles. According to the argument of Laval et al., this implies that most package tangles are not critical.

In order to investigate the question of whether packages within tangles are closely related, we have built the package containment trees (PCT) with sets of packages from both the entire dependency graphs and tangles. PCTs are trees that are constructed as follows. The vertices of such trees are obtained by tokenising package names. Package names themselves are included, and if a package name contains more than one token, its parent package is obtained by removing the last token, and then also added to the tree. The edges connect vertices with their parents. A virtual root node is also added to the graph to ensure that the graph is connected. Figure 2.8 shows the PCT built from some selected core Java packages.

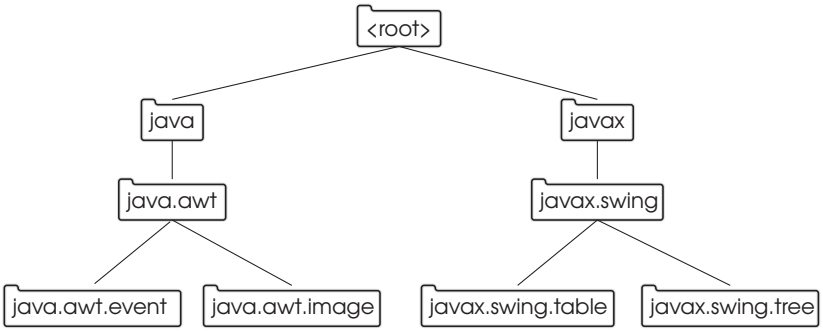


Figure 2.8 *The package containment tree for selected core Java packages.*

[5] Packages in Java correspond to namespaces in other programming languages.

2.3.1 Building the Package Containment Tree

A PCT has a root node called **<root>** which glues base packages that are not from the same hierarchy. The dot character (.) is used as a separator between packages and sub-packages, as used in Java specifications. Each sub-package is added as a child of its parent package until a package has no parent but the root. Consider the packages {java.awt.event, java.awt.image, javax.swing.table, javax.swing.tree}. The package containment tree of these packages is shown in Figure 2.8.

2.3.2 Metrics on Package Containment Tree

In this section, we present some of the metrics that are used in the analysis of package containment trees. In particular, we discuss the diameter (TDIAM) and PASTA metrics that can be used as indicators of the desirability of cyclic dependency.

Tree Diameter (TDIAM)

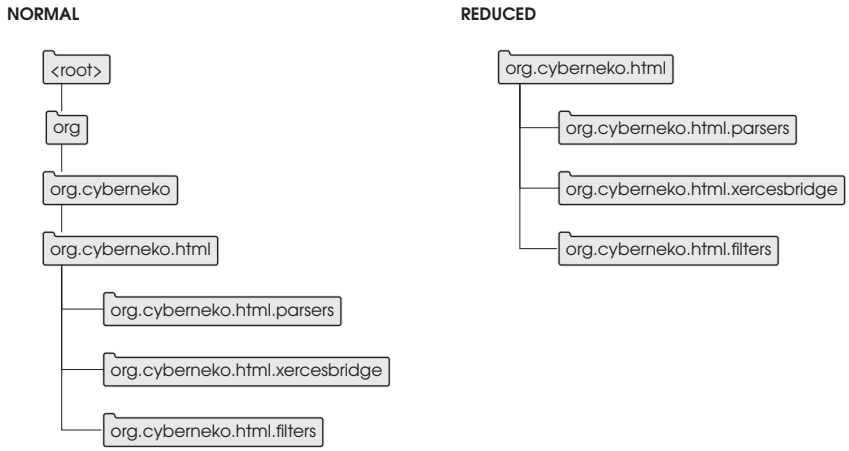
Laval et al. used TDIAM to denote the maximum distance between any pair of packages in the PCT [25]. Large values of TDIAM indicate that packages do not come from the same branch. Consider the following two packages from the JRE, javax.swing.events and java.awt. TDIAM is large because the path between the Swing and AWT packages passes through the root (Figure 2.8). In order to normalise TDIAM, Laval et al. suggested assigning a weight for each link in the tree based on the distance from the root. In order to measure the desirability of a cyclic dependency, Laval et al. associated the desirability of a cyclic dependency with lower values of TDIAM. The suggested weight for links between nodes in the PCT is 2^{-level}.

The PASTA Metric

Hautus proposed a metric for measuring the level of software modularity that is called the PASTA metric [81]. This metric can be applied to the software package-containment-tree and is defined as "the weight of all desirable dependencies in all packages divided by the total weight of the dependencies in all packages". The weight of a dependency is the number of class references made from one package to another. Hautus suggested giving higher weight for high level packages, which corresponds to the notion that high level architecture is more important than lower level architecture, which is consistent with the Laval et al. approach [25].

2.3.3 Reduced Package Containment Trees

The average diameter of PCTs might be affected because the nested sub-packages that do not have classes in them but may have direct sub-packages. In order to overcome this issue, we collapse consecutive empty sub-packages. For instance, Figure 2.9 shows the package containment tree built from `NekuHTML-1.9.14` system. Since the package `org` and `org.cyberneko` are empty, they are collapsed to the upper level. This shortens the height of the PCT from 4 to 2.



`Figure 2.9` *The normal and reduced PCTs of NekuHTML-1.9.14 system.*

2.4 Graph Topology

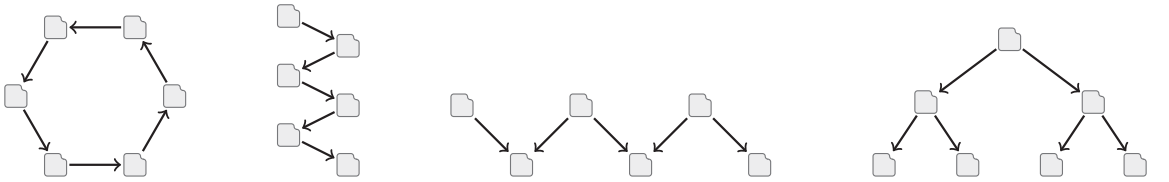
The term '*Topology*' is the "study of geometrical properties and spatial relations unaffected by the continuous change of shape or size of figures" [82]. In graph theory, the word topology can refer to the arrangement of vertices and edges in some pattern. Classifying graph topologies has several applications. For instance, the arrangement of computers in a network can be used to identify the physical topology of the network.

Some graphs may have no definite shape because of the complexity in their layout. However, a complex graph tends to be a result of accumulated building blocks that may have been combined in a way that obscures the original structure.. Although some researchers looked into identifying patterns in subgraphs, so-called motifs[6] [84, 85, 86, 87], we found this to be out of the scope of our study.

There has been some relevant research on the classification of arbitrary graphs, which is related to our work. For example, several authors proposed using supervised machine learning techniques such as Support Vector Machines (SVMs) [88, 89, 90, 91, 92].

[6] A *motif* represents patterns of interconnections that occur in a graph. [83]

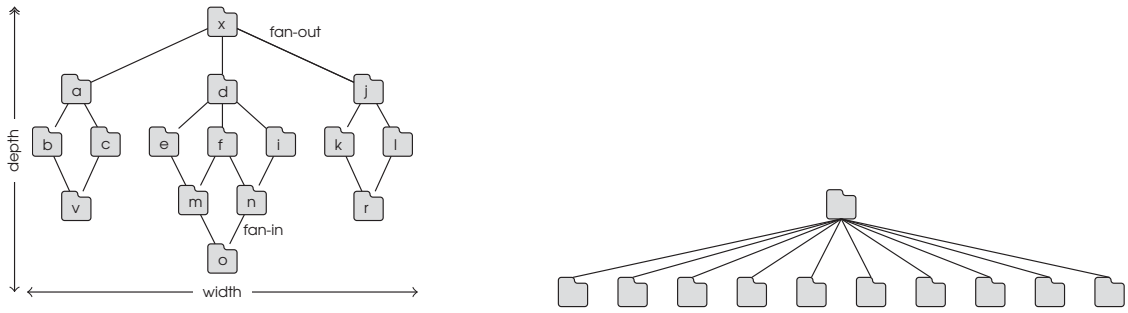
These approaches require a corpus of labelled training data, which we do not have. Zhu et al. [93] suggested some metrics that are suitable for arbitrary graphs. Zhu et al. formalised the problem of property–based graph classification and provided a greedy algorithm that selects some discriminative metrics and a classifier that can be used to classify general graphs. However, our requirements are rather more specific to classifying different types of SCCs.



(a) Cycle (b) Tall (c) Flat (d) Balanced binary tree

Figure 2.10 *Cycle, tall, flat and balanced binary PDGs.*

In regard to classifying software dependency graphs, Melton and Tempero [94, 95, 96] discussed some shapes formed by PDGs and pointed out that organising packages in a flat structure is the most preferred among other shapes (Figure 2.10). In addition, Pressman [2] suggested some design heuristics to organise the packages of a system. Pressman recommends avoiding excessive fan-out, which can result in what is referred to as "pancaked structure" (Figure 2.11). Pressman attests that a program should be maintained in a way to control the number of fan-outs to be balanced with fan-ins, resulting in diamond shape subtrees. Lower level sub-packages should provide the entry to the package by providing enough abstraction to the functionality implemented in the top levels. The depth and width indicate the level of control and span of control respectively [2].



(a) Controlled structure (b) Pancaked structure

Figure 2.11 *Controlled vs. Pancaked structure.*

2.5 Removal of Cyclic Dependencies – Refactoring

In spite of the design violation caused by cyclic dependencies, the functionality of the system may not be affected. For instance, the two systems discussed in Section 1.1 (ANTLR and Hibernate) contain many cyclic dependencies, yet are widely used and have been proven successful [97, 98]. "Code smells", design violations and potential design defects are some of the issues that need to be addressed in order to improve the overall quality of the system. The techniques used to remove cyclic dependencies are a subset of refactoring methods used in software design. Code refactoring can be defined as the process of restructuring an existing internal code by altering its content without changes in external behaviour [50]. Refactoring can be applied on class level or even architectural level. An example of design level refactoring is the correction of code smells, while architectural level refactoring is concerned with the placement of classes within packages or introducing new packages. We list here some of the work done in the literature which is related to refactoring in general and the removal of cyclic dependencies as a special case of refactoring methodologies in particular.

Ambler et al. suggest the elimination of cyclic dependencies, either by moving classes between packages or introducing a new package that contains all of the classes that are cyclically dependent [51]. Bourquin and Keller presented an experience report on high impact refactoring based on architectural violations [99]. The refactoring methods can be summarised as fixing code smells, removing duplicated segments of code and relocating classes to other packages, or creating new packages if necessary and introducing dynamic proxies. Bourquin and Keller claim that the number of classes involved in cyclic references has declined to half as a result of the refactoring methods applied.

Simon et al. presented a metric based refactoring tool that has a visualisation component [100]. Simon et al. claim that automating refactoring is an issue and the developer is the last authority to determine if a refactoring should be carried out or not. Therefore, for a system that has a large number of dependencies in cycles, human factors have to be involved in rectifying those cyclic dependencies. Simon et al. focused their research on four refactoring candidates which are: move method, move attribute, extract class and inline class.

Melton and Tempero presented a tool called `Jepends` that analyses the source code of Java programs in order to identify classes that are possible refactoring candidates [101]. Their main motivation is the removal of cycles that are considered detrimental to software understandability, testing and reuse. `Jepends` mainly targets cycles of a large size. Melton and Tempero investigated the presence of cycles in class graphs

[22]. These graphs are characterised to assess the "refactorability" of tangles using an approximation algorithm for the minimum feedback arc set problem. This is similar to our approach. However, we investigate different types of dependency graphs and use a more fine-grained method of classification that avoids the use of metrics with high computational complexity. Melton and Tempero presented the CRSS metric which can be used to identify candidates for refactoring and therefore improve the package structure of software systems [94].

Moha provided a systematic approach for the detection and correction of design defects in object-oriented architectures [102]. Moha claims that defect detection and correction are two related activities, yet previously they have been studied separately. According to Moha, auto-refactoring is a difficult task because of the lack of specification of defects and automated tools to support defect detection and correction. Moha adds that existing refactoring tools require manual interaction. Moreover, software metrics are not sufficient to understand the complexity of large software architecture, and therefore they cannot be solely used to detect design defects.

Abdeen et al. presented work focused on optimising software modularity by minimising package coupling and cyclic dependencies [103]. Abdeen et al. presented an algorithm and a set of measures to optimise the modularity of software systems. In order to minimise cyclic dependencies, some classes are moved between existing packages without introducing new packages, therefore maintaining the size and complexity of the system on the package and class levels.

Dietrich et al. investigated the presence of various class level anti-patterns in programs, including different types of cyclic dependencies [23]. They found that these anti-patterns are very common. Based on their findings, they have proposed an algorithm based on edge scoring to effectively remove these anti-patterns from programs [87], and investigated several refactoring techniques to automate the removal of dependencies from programs [104, 52, 105].

Shah et al. [52] presented several refactoring patterns that can break cyclic dependencies. These are: (1) type abstraction, (2) dependency injection, (3) relocating classes and packages, and (4) inlining. Shah et al. suggest that a set of pre- and post-conditions have to be met for each refactoring to be successful.

2.6 Conclusion

To sum up, this chapter has presented a background and a review of the work related to the research questions stated in Section 1.3. In particular, this included a discussion of the methods used in order to analyse and detect cyclic dependencies in software

design which can be modelled as strongly connected components in graph theory. We presented some metrics and tested their ability to detect the shape of tangles and, more importantly, we have taken in consideration their computational complexities. Furthermore, the notion of the package containment tree has been discussed to analyse cyclic package dependency graphs. Some metrics and theories related to PCTs have been discussed and will be further expanded in Chapter 5 with some results that validate the assumption that most of the cyclic references tend to be formed around parent packages. A definition of graph topology was provided and we discussed some common dependency graphs shapes used in the literature related to software. Finally, we looked at the cyclic dependencies problem from the refactoring and code change point of view.

In the next chapter, we provide a study of dependency graphs and the method used to build them and to extract cyclic dependencies. In addition, we look at different types of tangles on class and package dependency levels.

CHAPTER 3

DEPENDENCY GRAPHS AND TANGLES

“ It’s difficult to offer hard pieces of guidance when trying to define a well-controlled set of dependencies”
[Martin Fowler]

Contents

Summary – In this chapter, we explore the notion of dependency graphs and tangles in more detail. It turns out that there are not only different types of dependency graphs on class and package levels, but also several methods to build such graphs and to define tangles. We discuss this in detail, propose several precise definitions, and review the definitions used in existing work. We also present a comparison between building dependency graphs from source code and byte code, and which method was applied in this study and the reason behind our choice.

3.1 The Level of Dependencies in Software

Software dependency can be defined as the degree to which a program module relies on each of the other modules in order to compile and function properly [49]. In general, a dependency exists when two pieces of code have some relationship. A *dependency* of a module *A* on another module *B* exists when there exists at least one reference made from module *A* to module *B*. The dependency relationship is transitive. If module *A* depends on module *B*, and module *B* depends on module *C*, then module *A* indirectly depends on module *C* [106]. A module that depends on another module is called a *dependent module*. A module that is required by another is called a *dependee module*. In this section, we discuss different levels of dependencies that can be extracted from Java programs.

Our interest is mainly focused on dependencies that result within Java programs (internal). Examples of external dependencies are references to file system, database or network resources. In Java, internal dependencies can be analysed on different levels which are:

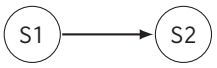
- statement
- method
- class
- package
- component

In this chapter, we provide some definitions for those levels of dependencies and the methods available to build dependency graphs from them.

3.1.1 Statement Level Dependency Graph

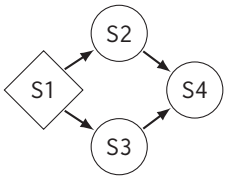
A *statement level dependency* occurs when a resource in a statement is accessed in another. Analysing dependencies at this level can be useful in detecting opportunities for parallelism and optimisation [107]. For instance, two or more consecutive statements can be executed in parallel if they do not share a common resource, i.e. the execution of a statement is dependent on the execution of the preceding one. The types of dependencies in this level can be of two types: *data* and *control* (Listings 3.1 and 3.2). A statement level dependency graph *"Dependence Graph"* is a set of vertices that represent statements in program code and a set of edges that represent the flow direction between statements [108, 107, 109, 110]. Data dependencies deal with

variables and resources while control dependencies deal with loops and branches. Based on this type of dependency graph, the Cyclomatic complexity is a popular metric which measures the complexity of the program code [111].




```
a = b * c;   // S1
d = a + 1;   // S2
```

Listing 3.1 *An example of data dependence*




```
if ( a == 1 )    // S1
    d = a + 1;   // S2
else
    d = a - 1;   // S3
c = d + a;       // S4
```

Listing 3.2 *An example of control dependence*

3.1.2 Method Level Dependency Graph

A *method level dependency* occurs when a method executes another method. *"Call Graph"* is a term used to describe the relationships between methods in a program [112, 113]. The analysis of this kind of dependency is useful in detecting methods that are never referenced and thus might be candidates for removal. Call graph can be used to measure the complexity of program execution and detecting opportunities for parallelism. In particular, when methods are mutually exclusive, they can be executed in parallel. Listing 3.3 and Figure 3.1 show a simple example of a call graph.

```
class Program{
  void run(){ ... }
  void main(){ run(); }
}
```

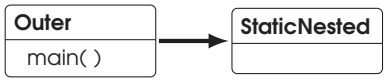
Listing 3.3 *An example Java Program*

Figure 3.1 *An example of call graph*

3.1.3 Class Level Dependency Graph

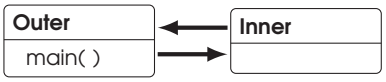
A *class level dependency* can be found when a class references another one. The common types of class-to-class relationships in Java are `extends` and `uses` [52, 23]. Note that

there is an implicit dependency between a class and its nested classes. Nested classes are divided into two groups: static and non-static (Listing 3.4 and 3.5). A nested class that is declared `static` is called *nested static class*. Non-static nested classes are called *inner classes* [114, Sec. 8.1.3]. The need for this distinction is important because inner classes naturally form cyclic dependencies with their outer classes. Specifically, the outer class makes its instance **"this"** available to its inner classes but not to static nested classes on the byte code level. In addition, static nested classes reference outer classes and vice versa in source code. However, inner classes reference outer classes by using an implicit reference to the instance of the outer class. Note that on the source code level, both cases can be considered in a cyclic dependency.




```
class Outer{
    static class StaticNested { }
    void main(){
        new StaticNested();
    }
}
```

Listing 3.4 *An example of a static nested class*




```
class Outer{
    class Inner { }
    void main(){
        new Inner();
    }
}
```

Listing 3.5 *An example of an inner class*

In addition to non-static inner classes and static nested classes, anonymous classes explicitly make a cyclic dependency with their outer classes. Anonymous classes are just like other classes except that they do not have names. Listing 3.6 shows an example of an anonymous class. Compiling the code in Listings 3.4 and 3.5 will result in byte code files with the names `Outer$StaticNested.class` and `Outer$Inner.class` respectively. However, since anonymous classes do not have names, the Java compiler assigns a unique identifier (number) to those anonymous classes. The compiled byte code of the anonymous class shown in Listing 3.6 would be `Outer$1.class`.

We treat *inner*, *nested static* and *anonymous* classes the same way when we build class dependency graphs.

```
abstract class Anonymous{ abstract void go(); };
class Outer{
  void main(){
    new Anonymous(){
      void go(){ }
    };
  }
}
```

Listing 3.6 *An example of an anonymous class*

3.1.4 Package Level Dependency Graph

Two packages form a *package level dependency* when a package P_1 depends on package P_2 if a class exists in P_1 that references a class in P_2. Fowler [49], Tessier [115] and Laval et al. [25] argue that analysing the dependencies among packages has more priority than lower levels. This is mainly due to the fact that managing the dependencies on the package level hides redundant relationships between packages. Furthermore, a package level dependency provides a higher level of abstraction that simplifies the understanding of program structure.

3.1.5 Component Level Dependency Graph

A *component graph* is a Unified Modelling Language (UML) diagram that illustrates the relationships between the components of a system. A *software component* contains implementation items such as executables and logic binaries. Components can be seen as encapsulation units within a system or a subsystem that requires the functionality of other components in order to provide their own functionality. Java class files and any associated metadata and resources can be assembled into distributable containers which are called Java Archives (JARs). A JAR can be considered as a component that can be deployed and integrated with other components to build an entire system.

3.2 Types and Levels of Tangles

In this section, we present the method used to build tangles from dependency graphs and present different types of tangles. Two tangle types are extracted from class level dependency graphs and two tangle types are on the package level. The definitions of tangle types provided later in this section are based on the dependency graphs depicted in Figures 3.2, 3.3, 3.4 and 3.5.

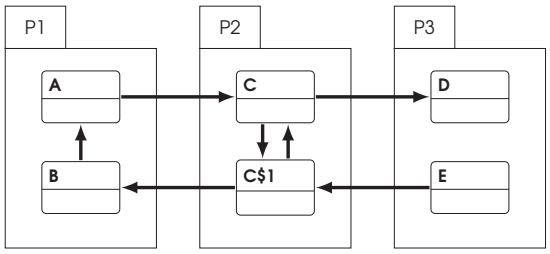


Figure 3.2 *Example classes and packages (UML class diagram). Note that C$1 is an inner class of C.*

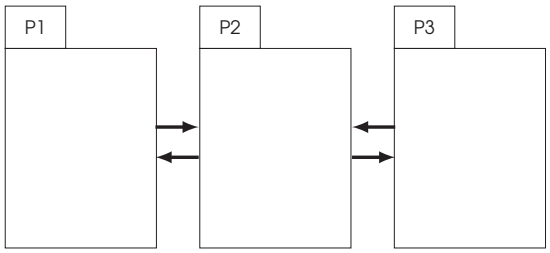


Figure 3.3 *The package graph G_p.*

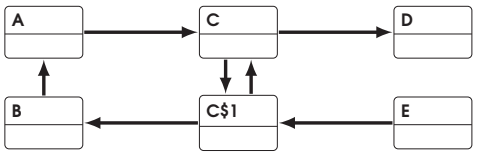


Figure 3.4 *The class graph G_c.*

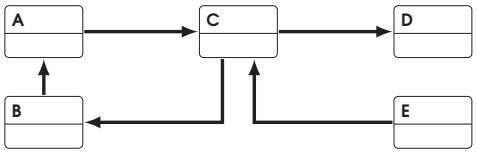


Figure 3.5 *The top-level-class graph G_{tlc}.*

3.2.1 Extracting Tangles from Dependency Graph

We explored the available methods for identifying and extracting SCCs from graphs. The straightforward method is running a DFS from every vertex (source) to all other vertices (targets) in the graph. Whenever there is a path from source to target and vice

versa, the two vertices belong to the same SCC. Using a shortest path algorithm, such as Dijkstra's shortest path algorithm, [116, 117] can make the process faster. The problem is that this approach does not scale up with very large graphs. A good algorithm available to enumerate SCCs is Tarjan's algorithm which runs in linear time $\mathcal{O}(|V| + |E|)$ [37] while the other methods are polynomial in terms of running time and space. Listing 3.7 shows the Java implementation of the Tarjan's algorithm.

```
static<V,E> Collection<Collection<V>> tarjan(Graph<V,E> graph){

  Collection<Collection<V>> sccs = new ArrayList<>();

  for(V v : graph.getVertices()){
    if(!v.visited){
      strongconnect(sccs,graph,v,new Stack<V>());
    }
  }
  return sccs;
}

static<V,E> void strongconnect(Collection<Collection<V>> sccs,
                               Graph<V,E> graph,V v, Stack<V> stack){
  v.lowlink = v.dn = count++;
  v.visited = true;
  stack.push(v);

  for(V w : graph.getSuccessors(v)){
    if(!w.visited){

      strongconnect(sccs,graph,w,stack);

      if(w.lowlink < v.lowlink){
        v.lowlink = w.lowlink;
      }
    }else if(stack.contains(w) && w.dn < v.dn && w.dn < v.lowlink){
      v.lowlink = w.dn;
    }
  }
  if(v.lowlink == v.dn){
    Collection<V> scc = new ArrayList<>();

    V w = null;
    do{
      w = stack.pop(); scc.add(w);
    } while(w != v);

    sccs.add(scc);
  }
}
```

Listing 3.7 *Tarjan's algorithm implementation in Java*

Listing 3.8 shows the method used to build tangles from dependency graphs. It accepts a dependency graph as an input and returns a collection of tangles that are subgraphs of the input graph. Note that tangles of a single vertex are not included because we do not consider them.

```
static<V,E> Collection<Graph<V,E>> getTangles(Graph<V,E> depGraph){
  Collection<Graph<V,E>> tangles = new ArrayList<>();
  Collection<Collection<V>> sccs = tarjan(depGraph);
  for(Collection<V> scc : sccs){
    if(scc.size()==1) continue;
    Graph<V,E> tangle = new DirectedSparseGraph<>();
    for(V v: scc){
      for(V w: scc){
        E e = depGraph.findEdge(v,w);
        if(e != null) tangle.addEdge(e,v,w);
      }
    }
    tangles.add(tangle);
  }
  return tangles;
}
```

Listing 3.8 *The method used to build tangles from dependecy graphs.*

3.2.2 Class Tangles

The set of *class* tangles T_c is defined as follows: a class tangle t_c is a strongly connected component extracted from $G_c = (V_c, E_c)$. A tangle has to have at least two vertices. Figure 3.4 and 3.6 show an example of a class tangle extracted from a class graph. We build class tangle graphs using the method shown in Listing 3.9.

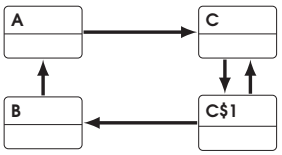


Figure 3.6 *Class tangles,* $T_c = \{\{A, B, C, C\$1\}\}$.

```
static<V,E> Collection<Graph<V,E>> getCTangles(Graph<V,E> cdg){
  return getTangles(cdg);
}
```

Listing 3.9 *The method used to build class tangles.*

3.2.3 Top-Level-Class Tangles

The set of *top-level-class* tangles T_{tlc} is defined as follows: a top-level-class tangle t_{tlc} is a strongly connected component extracted from G_{tlc} (Figure 3.5 and 3.7). Top-level-class tangles can be built the same way class tangles are built except that all inner, nested static, and anonymous classes are collapsed to the top level class. Therefore, the class dependency graph is modified and then the same method that is used to get class tangles is applied to generate top-level-class tangle graphs (Listing 3.10).

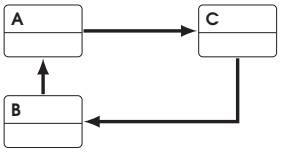


Figure 3.7 *Top Level Class tangles,* $T_{tlc} = \{\{A, B, C\}\}$.

```
static<V,E> getTopClass(Graph<V,E> cdg, V v){
  for(V p : cdg.getVertices()){
    if(p.toString().contains("$")==false){
      if(v.toString().startsWith(p.toString())){
        return p;
      }
    }
  }
  return v;
}

static<V,E> Collection<Graph<V,E>> getTLCTangles(Graph<V,E> cdg){
  Graph<V,E> topClassDG = new DirectedSparseGraph<>();
  for(E e : cdg.getEdges()){
    V v = getTopClass(cdg, cdg.getSource(e));
    V w = getTopClass(cdg, cdg.getDest(e));
    if(v!=w && topClassDG.findEdge(v,w)==null){
      topClassDG.addEdge(e,v,w);
    }
  }
  return getTangles(topClassDG);
}
```

Listing 3.10 *The method used to build the top-level-class tangles.*

3.2.4 Weak Package Tangles

The set of *weak package* tangles T_p^w is defined as follows: a weak package tangle $t_p \in T_p^w$ is the package graph built from the packages and their relationships in G_p (Figure 3.8). In order to build weak package tangles, we use the same method `getTangles(Graph)` presented in Listing 3.8, with the exception that the graph dependency graph is built in advance (Listing 3.11).

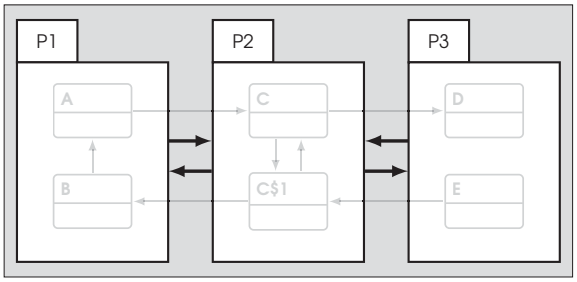


Figure 3.8 *Weak package tangles, $T_p^w = \{\{P1, P2, P3\}\}$.*

```
/*
 *  getPackage(Graph, V) returns a the package vertex of the given
 *  class vertex
 */
static<V,E> Collection<Graph<V,E>> getWPTangles(Graph<V,E> cdg){
  Graph<V,E> pdg = new DirectedSparseGraph<>();
  for(E e : cdg.getEdges()){
    V v = getPackage(cdg, cdg.getSource(e));
    V w = getPackage(cdg, cdg.getDest(e));
    if(v!=w && pdg.findEdge(v,w)==null){
      pdg.addEdge(e,v,w);
    }
  }
  return getTangles(pdg);
}
```

Listing 3.11 *The method used to build the weak package tangles.*

3.2.5 Strong Package Tangles

The set of *strong package* tangles T_p^s is defined as follows: for each class tangle $t_c \in T_c$, a strong package tangle $t_p \in T_p^s$ is the package graph built from the classes and their relationships in t_c as described above. It follows directly from the definition that each strong package tangle is either a weak package tangle or is embedded in a weak package tangle (Figure 3.9). In order to build strong package tangles, we enumerate the list of class tangles. Then, from each one we build a package tangle as used in Listing 3.11, given that a class tangle vertices come from the different packages (Listing 3.12).

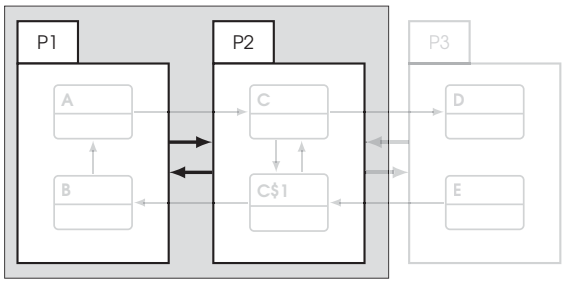


Figure 3.9 *Strong package tangles, $T_p^s = \{\{P1, P2\}\}$.*

```
static<V,E> Collection<Graph<V,E>> getSPTangles(Graph<V,E> cdg){
  Collection<Graph<V,E>> pdgs = new ArrayList<>();
  Collection<Collection<V>> sccs = tarjan(cdg);
  for(Collection<V> scc : sccs){
    if(scc.size()==1) continue;
    Graph<V,E> graph = new DirectedSparseGraph<>();
    for(V v : scc){
      V s = getPackage(cdg, v);
      for(V w : scc){
        V t = getPackage(cdg, w);
        E e = cdg.findEdge(v,w);
        if(!(s==t || e == null || graph.containsEdge(s,t))){
          graph.addEdge(e,s,t);
        }
      }
    }
    if(graph.getEdgeCount()>1){
      pdgs.addAll(getTangles(graph));
    }
  }
  return pdgs;

}
```

Listing 3.12 *The method used to build the strong package tangles.*

In order to illustrate these concepts, consider the dependency graph in Figure 3.2 which shows a simple scenario using a UML class diagram. The respective class graph is shown in Figure 3.4 and the respective package graph in Figure 3.3. The class graph contains a single class tangle $\{A, B, C, C\$1\}$ (Figure 3.6). By reducing inner classes, there is a top-level-class tangle which is $\{A, B, C\}$ (Figure 3.7).

There is a single weak package tangle $\{P1, P2, P3\}$, as this is the sole strongly connected component in the package graph. However, this differs from the strong package tangle $\{P1, P2\}$ obtained from the class tangle (Figure 3.9).

Note that the two notions representing package tangles differ if packages are not

cohesive. In particular, this means that weak package tangles could be removed by splitting packages. Strong package tangles are related to the strong cyclic dependencies anti-pattern used in [23] and [87].

3.3 Building Dependency Graphs

Dependencies in Java programs can be analysed either from *source code* or *byte–code*. Source code is a set of instructions written in human readable programming language, mostly using text. Source code files generally have a `.java` extension. On the other hand, the byte code is a compiled code which is composed of a set of instructions. Byte code files can be executed by a virtual machine and usually have a `.class` extension. More differences are shown in Table 3.1.

	source code	byte-code
APIs Used	PMD, JavaNCSS	ASM, BECL
Fully qualified names	no	yes
Reference to constants	yes	no
Annotations	yes	some, depending on retention policy

Table 3.1 *Some differences between source code and byte code.*

Because we extract dependencies from byte-code, references made to String and primitive type constants (static final fields) are copied into the referencing class, a technique called *"constant inlining"* [114]. Listings 3.13 and 3.14 show the source code and byte–code of two classes: A and B respectively. Class B makes a reference to a constant declared in class A. However, the class B is independent as shown by analysing the byte–code. The compiler may copy such constant value to improve the performance of the running time.

```
class A {
    public static final int a = 42;
}
//javap -constants A.class
//Compiled from "A.java"
class A {
  public static final int a = 42;
  A();
}
```

Listing 3.13 *The source code and byte-code of class A*

```
class B {
    public static final int b = A.a;
}
//javap -constants B.class
//Compiled from "B.java"
class B {
  public static final int b = 42;
  B();
}
```

Listing 3.14 *The source code and byte-code of class B*

On the source code level, an explicit dependency can be made from a class to another by specifying its fully qualified name in the statement call. Alternatively, external classes can be imported, either individually such as `import java.util.List`, or as part of an imported package such as `import java.util.*`. Byte–code based

analysis has the advantage that all import statements are resolved by the compiler, and all references to other classes are resolved via fully qualified names.

Table 3.2 shows some of the existing tools and publications that analyse the dependencies on the different levels of dependency stated in this chapter.

Tool/Paper	Extraction	Statement	Method	Class	Package	JAR
ByeCycle [118]	B			x		
Classycle [119]	B			x	w	
CodePro [120]	B			x	w	x
Dependency Finder [115]	B			x	w	x
Eisenbarth et al. [121]	S		x			
JDepend [122]	B				w	
Jepends [101]	S		x	x		
JooJ [118]	S			x		
Lattix LDM [123]	B				w	
McCabe [111]	S	x				
Melton & Tempero [22]	S+B			x	w	
Popsycle [25, 124]	B				w	
Sarkar et al. [125]	S				w	
Shah et al. [52]	B				w+s	
XplrArc [126]	B			x	w+s	x

Table 3.2 *Some tools and research done in analysing cyclic dependencies. B: Byte-code, S: Source code, w: weak, s: strong*

3.4 Conclusion

In summary, in this chapter we presented different levels of dependencies that can be analysed from software systems. Most work in dependency analysis is concerned with analysing the graph defined by packages and their relationships. We use $G_p = (V_p, E_p)$ to refer to this graph, where V_p (the vertices) represents a set of packages and E_p (the edges) a set of dependencies between packages. This means that the package graph is built from the class graph. We use $G_c = (V_c, E_c)$ to refer to the graph consisting of classes and their relationships. Unlike Shah et al. [52], we treated `extends` and `uses` dependencies equally in G_c. In order to measure the effects of the cycles generated by the compiler between outer classes and their inner classes, we also investigated the graph consisting of top-level-classes and their relationships, ($G_{tlc} = (V_{tlc}, E_{tlc})$). The vertices in G_{tlc} are top level classes. A dependency exists between two top level classes $(c_1, c_2) \in E_{tlc}$ *iff* c_1 or any of the inner classes of c_1, depending on c_2 or any of the inner classes of c_2. The relationships between containers (JARs) are not investigated. Although cyclic dependencies exist in these graphs as well [23], such graphs tend to be small and can be analysed manually.

We also looked at two ways of building the dependency graphs, which are by

source code and byte code. We prefer to use the latter method because it is easier than analysing the source code. Furthermore, analysing the source code to extract dependencies requires an extensive knowledge of the language compiler and its role of detecting referenced classes. Therefore, building the dependencies from source code is redundant because it tends to require building a compiler. We also presented a summary of the existing tools and publications that analyse software dependencies.

In the next chapter, we look at the different shapes of tangles, their properties and some metrics specifically developed to detect them. We also present an algorithm to classify tangles and the methodology followed to test its robustness.

CHAPTER 4

TANGLES CLASSIFICATION

“ The classification of facts, the recognition of their sequence and relative significance is the function of science, and the habit of forming a judgment upon these facts unbiassed by personal feeling is characteristic of what may be termed the scientific frame of mind.”

[Karl Pearson]

Contents

Summary – In this chapter, we present the shapes of cyclic dependencies we used which are categorised according to the symmetry. Also, we present some new metrics that are tailor–made to detect some tangle shapes. We show how tangles were auto–generated to perform experiments and measure the robustness of metrics. In addition, we present our tangles classification algorithm and show its correctness and the method followed to test its correctness. We also apply the classifier on a data set of open–source Java systems to classify tangles and present the result of their classification. Finally, we present some results related to the classification of tangles and the shift of shape by moving between different levels.

4.1 Introduction

In order to classify tangles, we use a set of archetypical reference topologies. While these topologies are modelled using visual metaphors, we cannot rely on manual identification for several reasons. Firstly, the tangles can be fairly large. For instance, the JRE 1.6.0_17 contains class tangles containing 2110 classes and weak package tangles containing 228 packages. Secondly, how end users perceive a topology depends not only on the graph, but also on the layout that is being used to render the graph: different presentations of the same graph, particularly large graphs, hide different properties of it. Thirdly, we are interested in automating the process of classifying tangles, which makes a manual identification unsuitable.

This means that a set of metrics must be derived and used to classify tangles. A particular challenge is the computational complexity of the classification: many standard graph algorithms are in NP, which means that they are not suitable for the classification of tangles of non-trivial size. We discuss several metrics, archetypes and a classification algorithm based on these metrics in Section 4.5.

We cannot expect that large tangles are easily classified in the sense that they have the exact shape conforming to a particular topology. A classification that is useful in practice must have some robustness built into it. For example, instead of looking for perfect cliques, we want to look for clique-like structures.

We first assess the robustness of our classifier by generating tangles that are perfect archetypes of a particular topology, then subject these tangles to random modifications and checking whether the classification result remains stable. This is done in Section 4.4.

Secondly, we investigate tangles in real world Java programs. For this purpose, we use the Qualitas Corpus version 20101126 [127]. This is a set of 103 open source projects that has been widely used in empirical studies. For many programs, the corpus contains several versions. In this case, only the latest version is used. The class graphs are extracted from the compiled programs using the **Dependency Finder** [115].

In some programs, classes with the same fully classified class name occur in multiple jar files. In this case, it is not clear how the dependency graph should be built, as this depends on how class loaders are used by the programs. In other words, building the dependency graph requires semantic analysis. We therefore decided to remove these programs from the data set we used. The eight systems where this is the case are listed in Table 4.1.

aspectj-1.6.9	c_jdbc_2.0.2	castor-1.3.1	cayenne-3.0.1
drjava-stable-20100913	gt2-2.7-M3	jtopen-7.1	pooka-3.0-080505

Table 4.1 *List of projects in Qualitas Corpus which contain multiple classes that have the same qualified name.*

4.2 Reference Topologies

We use a set of reference topologies to classify tangles. We expect that tangles with the same topology share certain properties, including how change propagates through these networks ("ripple effects"), and how easy it is to comprehend and break these tangles. This is directly related to software maintenance issues. We describe these topologies and how they can be characterised using the metrics discussed in Section 2.2 and 4.3.

We selected these topologies based on the following criteria:

1. *Comprehensibility*: a topology is associated with a common visual metaphor.
2. *Computability*: a topology can be described using a combination of metrics with low computational complexity. This means that it is possible to build scalable tools to decide whether tangles have this topology.
3. *Relevance*: tangles with a given topology occur in significant numbers in real world programs. This will be shown in Section 4.7.
4. *Stability*: The values for the chosen combination of metrics remain robust under small random modifications of the topology. This will be shown in Section 4.4.

We divide tangles into two main categories, which are symmetric and asymmetric, (Figure 4.1 and 4.2 respectively). Symmetric topologies comprise *tiny, circle, clique, chain and star*. Asymmetric topologies comprise *semi–clique and multi–hub*. These topologies will be explained next.

4.2.1 Symmetric Topologies

Symmetric tangles (Figure 4.1) are composed of 1) tiny, 2) circle, 3) clique, 4) chain and 5) star. Although these tangle shapes look perfectly symmetric, tangles in real dependency graphs may contain minor alterations. Therefore, we consider a tangle symmetric if a large proportion of it is symmetric.

4.2.1.1 Tiny

A tiny tangle is a tangle with two vertices. This implies its topology: there are exactly two edges connecting the two vertices, one in either direction. Tiny tangles are

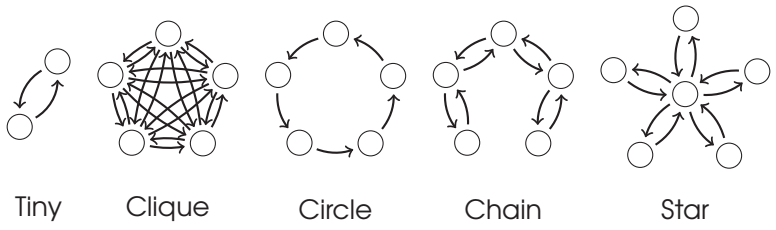


Figure 4.1 *Symmetric tangle topologies.*

pathological cases that have the characteristics of most other topologies. In particular, tiny tangles are simple cycles as well as cliques. This justifies treating them as a separate category. Tiny tangles can be automatically generated using Listing 4.1.

```
public static Graph<Integer,String> getTiny(){
  Graph<Integer,String> tiny = new DirectedSparseGraph<,>();
  tiny.addEdge("0->1",0,1);
  tiny.addEdge("1->0",1,0);
  return tiny;
}
```

Listing 4.1 *Tiny tangle generation function.*

4.2.1.2 Circle

A circle tangle is a simple tangle structure where vertices $\{v_1, v_2, \ldots, v_n\}$ are only connected by edges linking adjacent vertices: $E = \{(v_i, v_{i+1})\} \cup \{(v_n, v_1)\}$. In simple circles, the number of vertices equals the number of edges. We expect that circles exist in small tangles. Circle tangles can be automatically generated using Listing 4.2.

```
public static Graph<Integer,String> getCircle(int size){
  Graph<Integer,String> circle = new DirectedSparseGraph<,>();
  for (int i = 1; i < size; i++) {
     circle.addEdge(String.format("%d->%d", i, i + 1), i, i + 1);
  }
  circle.addEdge(String.format("%d->%d", size, 1), size, 1);
  return circle;
}
```

Listing 4.2 *Circle tangle generation function.*

4.2.1.3 Clique

A clique tangle is a tangle where each vertex has an edge connecting it with any other vertex. This indicates that $|E| = |V| \times (|V| - 1)$, which implies that edges are

quadratically proportional to the number of vertices. Finding very large perfect cliques is unlikely due to the fact that they require a large number of links between software components. Clique-like structures may appear due to the fact that features are added to software systems gradually. Existing modules may not be modified as frequently as newly introduced ones. Therefore, a clique cyclic dependency can occur if developers reference new features in old ones and vice versa. Clique tangles can be automatically generated using Listing 4.3.

```
public static Graph<Integer,String> getClique(int size){
  Graph<Integer,String> clique = new DirectedSparseGraph<,>();
  for (int i = 0; i < size; i++) {
    for (int j = i+1; j < n; j++) {
      clique.addEdge(String.format("%d->%d", i, j), i, j);
      clique.addEdge(String.format("%d->%d", j, i), j, i);
    }
  }
  return clique;
}
```

Listing 4.3 *Clique tangle generation function.*

4.2.1.4 Chain

A chain tangle is a sequential structure, with edges connecting adjacent vertices in both directions. Sometimes, the start and end point of a chain are connected as well. Chain tangles can be automatically generated using Listing 4.4.

```
public static Graph<Integer,String> getChain(int size){
  Graph<Integer,String> chain = new DirectedSparseGraph<,>();
  for (int i = 1; i < size; i++) {
    chain.addEdge(String.format("%d->%d", i, i+1), i, i+1);
    chain.addEdge(String.format("%d->%d", i+1, i), i+1, i);
  }
  return chain;
}
```

Listing 4.4 *Chain tangle generation function.*

4.2.1.5 Star

A star tangle is a topology with a central hub vertex, which is an endpoint (either incoming or outgoing) for all edges in the tangle. Star tangles can be automatically generated using Listing 4.5.

```
public static Graph<Integer,String> getStar(int size){
  Graph<Integer,String> star = new DirectedSparseGraph<,>();
  for (int i = 2; i <= size; i++) {
    star.addEdge(String.format("%d->%d", 1, i), 1, i);
    star.addEdge(String.format("%d->%d", i, 1), i, 1);
  }
  return star;
}
```

Listing 4.5 *Star tangle generation function.*

4.2.2 Asymmetric Topologies

Asymmetric tangles (Figure 4.2) are composed of two types, which are 1) semi–clique and 2) multi–hub. Initially, they are generated as symmetric and then altered, based on the size of the graph, to make them asymmetric. Listing 4.6 shows the function used to randomly add some edges to a tangle to make it asymmetric.

```
public static void noisify(Graph<Integer,String> graph){
  List<Integer> vs = new ArrayList<Integer>(graph.getVertices());
  int threshold=graph.getVertexCount()/10;
  for(int i=0;i<threshold;i++){
    Collections.shuffle(vs);
    Integer s = vs.get(0);
    Integer t = vs.get(1);
    String  e = String.format("%d->%d", s, t);
    if(graph.containsEdge(e) == false){
      graph.addEdge(e, s, t);
    }
  }
}
```

Listing 4.6 *Noisfy tangle function.*

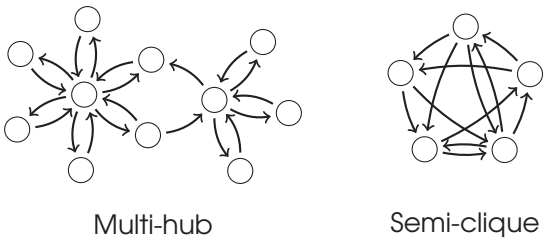


Figure 4.2 *Asymmetric tangle topologies.*

4.2.2.1 Semi–clique

A semi-clique tangle is a tangle that has approximately half as many edges as a clique. Semi-cliques are relatively dense tangles that lack the symmetric properties of any of the topologies described above. Semi-clique tangles can be automatically generated using Listing 4.7.

```
public static Graph<Integer,String> getSemiClique(int size){
  Graph<Integer,String> semiClique = new DirectedSparseGraph<,>();
  for (int i = 1; i <= size; i++) {
    for (int j = i+1; j <= size; j++) {
      semiClique.addEdge(String.format("%d->%d", i, j), i, j);
    }
  }
  semiClique.addEdge(String.format("%d->%d", size, 1), size, 1);
  noisify(semiClique);
  return semiClique;
}
```

Listing 4.7 *Semi-clique tangle generation function*

4.2.2.2 Multi–hub

A multi–hub tangle is a tangle that contains more than one hub. A vertex is considered as a hub if it has a relatively high betweenness than other vertices in the same tangle. In order to automatically generate a multi–hub tangle, we generate a random number of pure stars whose number of vertices in total equals to the requested multi–hub tangle size. Then these stars are joined. Some arbitrary edges are added until the generated tangle is strongly connected. Finally, some noise is added to the generated tangle to disturb the symmetry of its shape using `noisify(Graph)` method.

4.3 A Set of Custom Metrics

We considered using some of the standard metrics from graph theory to classify the tangles that were discussed in Section 2.2. However, we found such metrics computationally inefficient in most cases. Therefore, we have developed an alternative set of simple metrics that proves to be sufficient for our purpose, as demonstrated next.

4.3.1 The Depth of the SCC Decomposition Graph

A *cycle* is a directed path that starts and terminates at the same vertex. A cyclic path is called *elementary* if every vertex in the path is visited once. For instance, Figure 4.3

shows a graph which is composed of five vertices, nine edges and has six elementary cycles, i.e. circuits. Note that the cycles $(bdcb)$, $(cbdc)$ and $(dcbd)$ are permutations of the same elementary cycle composed of the vertices $\{b, c, d\}$. Disjoint elementary cycles are those which have no shared edges or vertices. We call cycles non-overlapping if they do not share an edge. For instance, the following sets of cycles extracted from Figure 4.3 are composed of cycles that are non-overlapping:

- $S_1 = \{(aba), (bdcb), (ded)\}$
- $S_2 = \{(aba), (bdecb)\}$
- $S_3 = \{(ded), (bdcab)\}$

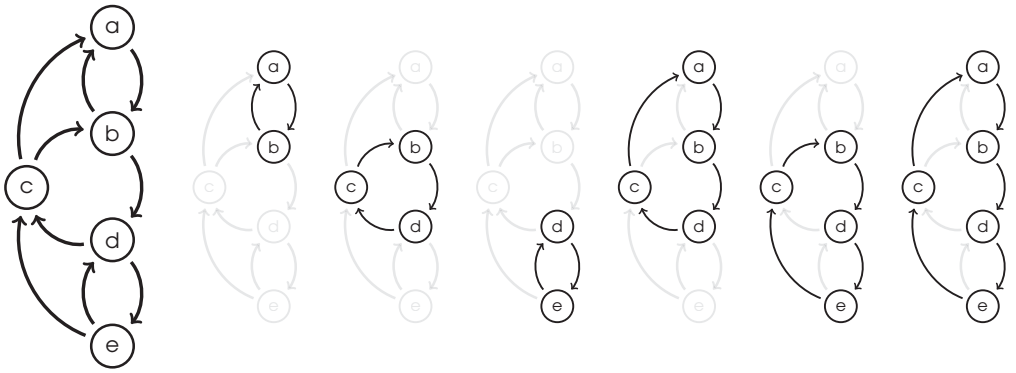


Figure 4.3 *A SCC and its elementary cycles.*

Figure 4.4 shows a decomposition graph of the SCC presented in Figure 4.3. The decomposition graph of a SCC can be built as follows: vertices are the SCC and the elementary cycles within the SCC, and edges are defined by set inclusion of vertices. The height of the decomposition graph can be measured as the length of its diameter.

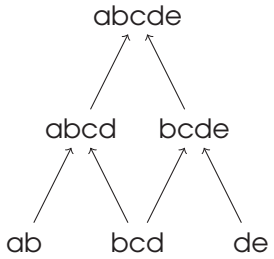


Figure 4.4 *Decomposition graph.*

Elementary cycles can be used to analyse and visualise cyclic dependencies between software modules. Elementary cycles are easier to understand and visualise relative to the whole SCC from which the cycle is extracted. As shown in Figure 4.3, the cycles (aba) and (ded) are the simplest among other cycles, and therefore can be easily

understood. McCabe [111] asserts that the Cyclomatic complexity number for a strongly connected graph G is equal to the maximum number of linearly independent circuits.

To the best of our knowledge, the most efficient algorithm that can enumerate all simple cycles in a directed graph is Johnson's algorithm [128] whose complexity is $\mathcal{O}((|V| + |E|)(|C| + 1))$ though the number of cycles $|C|$ can be exponential [128, 129].

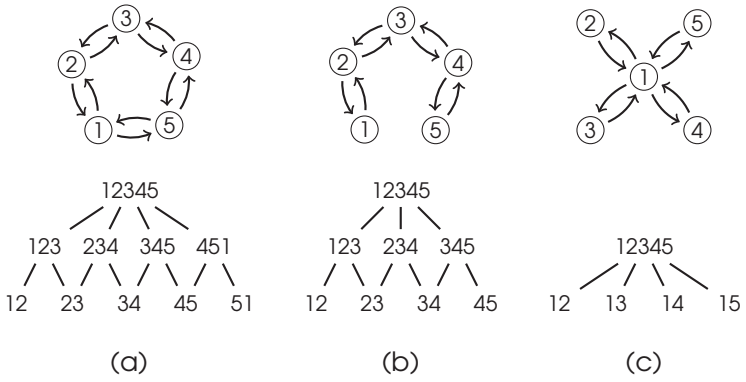


Figure 4.5 *SCC decomposition graphs of some tangles.*

Figure 4.5 shows some SCC-decomposition graphs of different tangle shapes. Note that the diameter of the decomposition graph is affected by the shape of the tangle. Due to the high complexity of generating the maximal set of non-overlapping elementary cycles from tangles, we considered this metric but did not use it.

4.3.2 Immediate Back-reference (BCKREF)

BCKREF has a similar flavour to TANGL (Section 2.2.8), but is faster to compute. It measures the relative number of edges $(v_1, v_2) \in E$ where the back-reference path between v_1 and v_2 consists of a single edge, i.e. $(v_2, v_1) \in E$. BACKREF can be computed using Equation 4.1. Using suitable data structures, BCKREF can be computed in time $\mathcal{O}(|E|)$.

$$BCKREF(G) = \frac{|\{(v_1, v_2) \in E \mid (v_2, v_1) \in E\}|}{|E|} \tag{4.1}$$

4.3.3 Starness (STAR)

STAR is the ratio between the maximum degree of vertices in a tangle and the number of edges. This metric is useful in detecting star-like topologies, similar to the tangles caused by classes and their non-static inner classes. In order to calculate the maximum degree, we sum the indegree and outdegree for each vertex. For a simple cycle or

a clique, both very regular symmetric topologies, STAR will be a very small value, tending to zero as $|V|$ gets large, $\frac{2}{|V|}$ for a simple cycle and $\frac{1}{|V|}$ for a clique. For a star-like tangle with a single high-degree central hub, STAR will be a value close to 1. STAR can be computed using Equation 4.2.

$$STAR(G) = \frac{\max_{v \in V} (deg(v))}{|E|} \tag{4.2}$$

Using suitable data structures, we can compute STAR in time $\mathcal{O}(|E|)$ by simply considering each of the edges in turn and building up a degree count for each vertex. We stipulate that a tangle classified as a star needs to have at least four vertices.

4.3.4 Chainness (CHAIN)

CHAIN is a measure designed to detect chains, i.e., tangles where each vertex references only two neighbours and is only referenced by these two neighbours, (with the exception of the two vertices at the end of the chain). We formally define CHAIN as follows. For a given vertex $v \in V$, we call another vertex $v' \in V$ a *friend* of v *iff* $(v, v') \in E$ and $(v', v) \in E$. Let $friends(v)$ be the number of friends of a vertex in V. Then CHAIN measures the relative number of vertices that have precisely two friends. CHAIN can be computed using Equation 4.3.

$$CHAIN(G) = \frac{\min_{v \in V} (|\{v \mid friends(v) = 2\}|, |V - 2|)}{|V - 2|} \tag{4.3}$$

For perfect chains, CHAIN is 1 (except where $|V| = 4$ and the chain is closed). Otherwise, CHAIN is still 1 for closed chains, i.e. chains where the start and the end vertices are connected. For both simple cycles and cliques with more than three vertices, CHAIN is 0. CHAIN can be computed in time $\mathcal{O}(|E|)$, since both the incoming and outgoing edges for each vertex must be considered.

4.3.5 Hubs (HUB)

HUB is a measure we used to detect the existence of hub vertices in tangles. We define hubs in tangles as vertices with high 'importance'. A vertex is considered a hub if it has a higher betweenness score relative to other vertices in a tangle. The betweenness centrality measure can indicate the likelihood of a vertex being passed through for all of the possible shortest paths in the tangle. The presence of hubs can indicate unequal distribution of betweenness for all vertices. Therefore, the distribution of betweenness is expected to be skewed in tangles containing hubs.

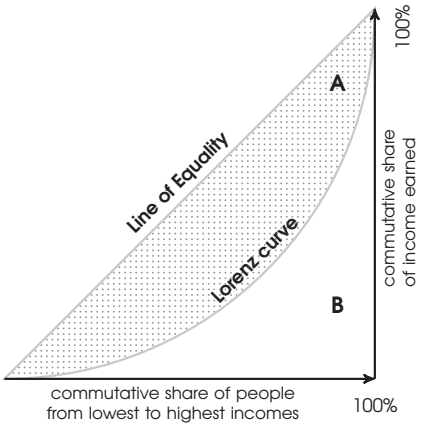


Figure 4.6 *Graphical representation of the Gini coefficient.*

In order to measure the skewness in vertex betweenness distribution, we use the Gini coefficient which has been used successfully in economic studies [71, 73, 72] and has also been applied in the measurement of skewness of software engineering metrics, e.g. [130, 131]. The Gini coefficient, a.k.a. *Gini index* or *Gini ratio* is defined mathematically, based on the Lorenz curve which plots the proportion of the total income that is cumulatively earned by population (Figure 4.6) [69, p.22]. Allison [132] defines the Gini coefficient as the ratio between the area formed between the line of equality and Lorenz curve (A) over the total area (A+B), i.e. $Gini = A/(A + B)$. Lerman and Yitzhaki [133] define the Gini coefficient as *"1 minus twice the area between the Lorenz curve and the diagonal line representing perfect equality"*. When we apply the Gini coefficient measurement, we consider the vertices in the tangle to be the population and the wealth is their betweenness centrality measure. In tangle shapes containing no hubs such as circles and cliques, the Gini coefficient is 0 (perfectly equal distribution), whereas tangle shapes with few hubs such as stars have a Gini coefficient value close to 1. Figure 4.7 shows some graphs and the respective Gini coefficient values of the betweenness scores. In cliques and simple circles, HUB is very close to 0 because the betweenness scores for such tangles is distributed evenly.

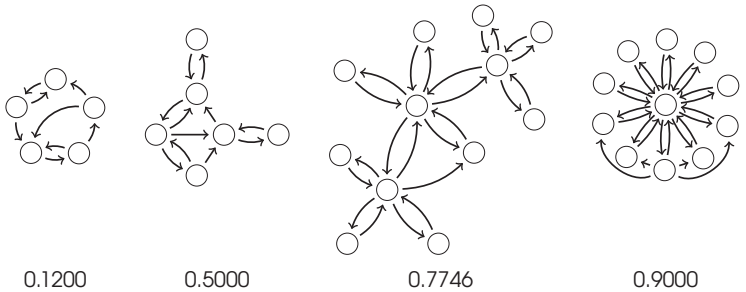


Figure 4.7 *Some example of tangles and their HUB values.*

There are some cases where the HUB metric might be misleading if it was used on very dense structures. Dense tangles usually have a small diameter ≈ 1, which implies that every vertex is linked to every other vertex. Therefore, almost all vertices have an equal betweenness. If a vertex is added to the tangle, then the betweenness score of the vertex that is linked directly to the newly added vertex strongly increases, and this vertex is classified as a hub. Figure 4.8 illustrates this scenario and shows that the HUB measure has been dramatically raised from 0 to 0.91.

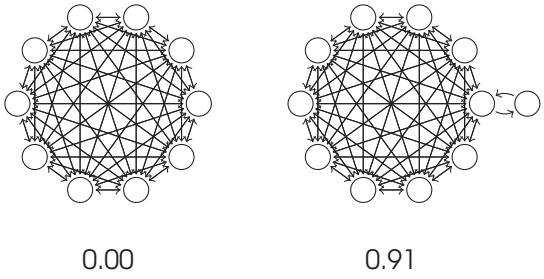


Figure 4.8 *HUB metric can be misleading on very dense tangles.*

We experimented with each of the computationally tractable metrics described above on a set of reference topologies, which are described in Section 4.2. We performed a robustness analysis in order to identify the most useful metrics. The results of this are given in Section 4.4.

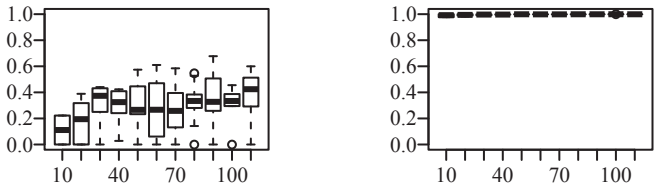
4.4 Robustness Analysis

In order to test the ability of the metrics to detect tangles, we manually create tangles of each topology that we have identified, initially with a variety of sizes (where we define the size of a tangle by the number of vertices in it). The tangle that we initially construct is a perfect example of the topology. However, this is clearly highly unlikely in real software systems. We therefore modify our graphs by using a set of mutations: *vertex addition*, *vertex deletion*, *edge addition*, or *edge deletion*. A set of modified graphs are created by randomly by choosing vertices or edges to mutate (and which mutation to apply at each point). After performing any kind of mutation, the graph is tested for the SCC property, and mutations that result in the graph not being strongly connected are not used. Note that the addition of a vertex entails adding two edges in order to maintain strong connectivity, while removing vertices can easily break the strong connectivity property of tangles.

Although vertex mutations have a greater effect on the shape of tangles, we decided to apply an equal proportion of all mutations to simulate normal scenarios of software change. The number of mutations applied is relative to the initial size of the tangle

which is 10%. This has been taken into consideration in order to largely preserve the structure and properties of each tangle topology.

Using the approach described in the Section 4.2, we created a set of graphs by starting from tangles of each of the topologies described in the previous section (with sizes between 5 and 100) and performing a number of mutations on each of them, where the numbers of mutations was one-tenth of the initial tangle size. This generated 30 new graphs, which all had approximately, but not exactly, the same size, since some had vertices added and some had vertices removed.



a) Circle (High variability) b) Star (Low variability)

Figure 4.9 *TANGL score for variations of the Circle and Star topologies. The x-axis indicates the size of the tangle while the y-axis indicates the TANGL value.*

Figure 4.9 shows a box plot (the black line is the median, the box shows the inter-quartile range, and the whiskers the full range) of the values of the TANGL metric for variations of the circle and star topologies, for different sizes of original tangles. Mutations of a circle that break an edge can result in a graph that is no longer strongly connected, therefore such an edge deletion mutation would not be applied. Edge addition mutations result in a graph that becomes progressively more tangled. As a result, the TANGL scores of the circle topology have wide ranges for all sizes. This high variability means that the TANGL metric cannot be used to detect this type of topology. However, for stars, the variability of TANGL is generally low. Since TANGL has a low variability for stars, we consider it robust. RATIO is a robust measure that can be used to detect circles.

Table 4.2 shows the box-plots matrix for the shapes of tangles and the metrics used. The size of tangles used to generate these plots are within the range of 5 to 100. The x-axis represents the size of the tangle while the y-axis represents the metric score. The CHAIN metric is robust because the only topology that scores higher than 0.5 is the *chain* topology while other topologies have a CHAIN score of, at most, 0.2. The range of values of each metric on the different topologies are shown in Table 4.3. We use the first quartile (Q1) and third quartile (Q3) bounds to determine the range of a metric for each topology.

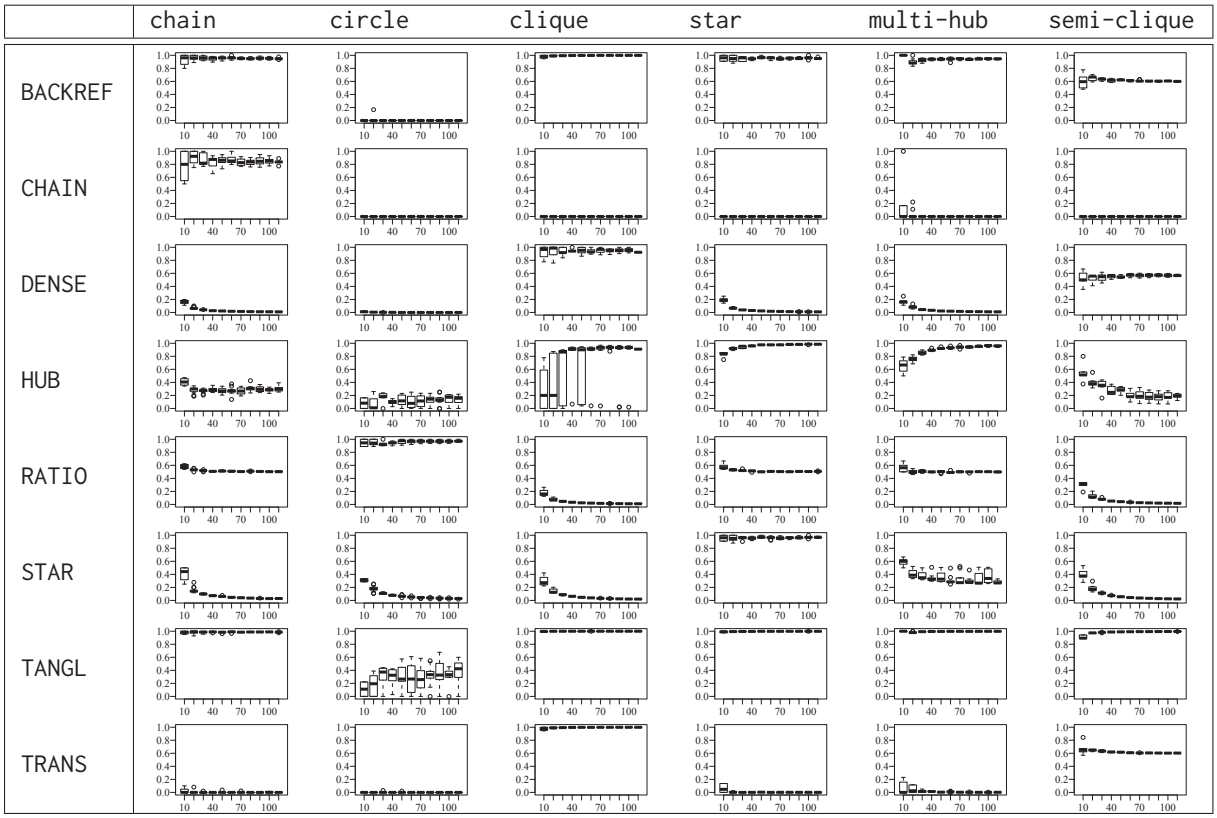


Table 4.2 *Boxplots of metrics scores on different tangles topologies. The x-axis indicates the size of the tangle while the y-axis indicates the metrics value.*

	chain		circle		clique		star		multi-hub		semi-clique	
	Q1	Q3	Q1	Q3	Q1	Q3	Q1	Q3	Q1	Q3	Q1	Q3
BACKREF	0.94	0.97	0.00	0.00	1.00	1.00	0.94	0.97	0.93	0.95	0.60	0.62
CHAIN	0.81	0.91	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DENSE	0.01	0.04	0.00	0.00	0.92	0.98	0.01	0.03	0.01	0.04	0.55	0.58
HUB	0.26	0.31	0.03	0.18	0.77	0.94	0.96	0.98	0.88	0.95	0.17	0.32
RATIO	0.50	0.52	0.94	0.98	0.01	0.04	0.50	0.52	0.49	0.51	0.02	0.05
STAR	0.04	0.09	0.03	0.09	0.03	0.07	0.95	0.98	0.28	0.42	0.03	0.07
TANGL	0.98	0.99	0.16	0.42	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00
TRANS	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.01	0.60	0.62

Table 4.3 *Metrics scores on different tangles topologies. Q1: first quartile, Q3: third quartile.*

We performed random mutations on tangles in order to add some noise and test the robustness of the metrics that were chosen for the classifier. However, adding a vertex requires adding two edges, which can easily distort the shape of relatively small tangles. Specifically, symmetric tangles can be easily affected by such mutation. This has been done to simulate real world programs when some code change is applied. The problem with applying random mutations is that the change in real software tends not to be random.

We used the robustness analysis to identify which metrics are robust for which topologies, and also to set values that could be used in a classifier. This is described in Section 4.5.

4.5 Classification of Tangles

We devised a classification algorithm based on some of the metrics discussed in both Section 2.2 and Section 4.3. The algorithm is shown in Figure 4.10. The metrics used in the algorithm were chosen based on their computational efficiency. The metric SIZE refers to the number of vertices in the tangle. It is designed to satisfy three criteria: (1) *completeness* - to be able to classify a high percentage of tangles, (2) *scalability* and (3) *stability/robustness*.

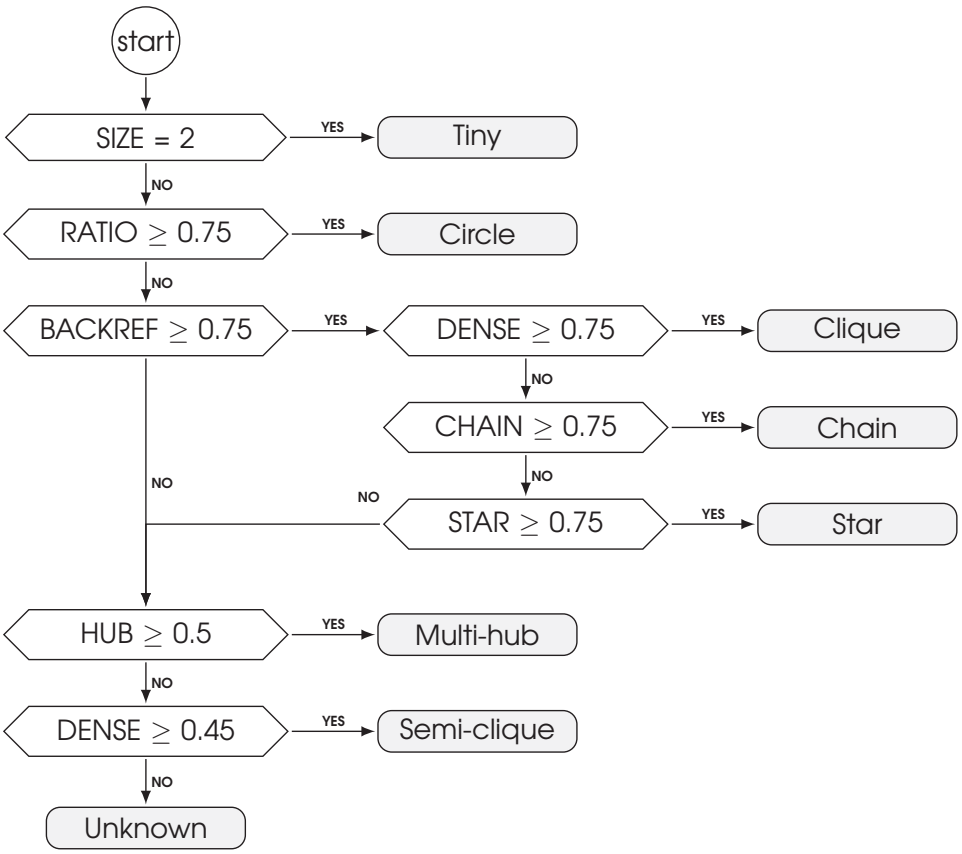


Figure 4.10 *Classification algorithm*

4.6 Validating the Classifier

In order to validate the correctness of the classifier, we generated a set of 30 graphs of each topology and size (8 topologies, 40 sizes ranging from 5 to 200 in steps of 5), each graph being a mutated version of the ideal topology, as in Section 4.4. We then classified this collection of graphs. The success rate of the classifier is given as the ratio of correct classifications to the total number of classifications. In particular, in order to test the correctness of the classifier, we used a confusion matrix (error matrix) [134]. Tables 4.4 and 4.5 show the confusion matrices for un-mutated and mutated sets of tangles. The numbers shown in both tables represent percentages. We found that the classification result remained stable in 96% of cases.

shape	chain	circle	clique	star	multi--hub	semi--clique	unknown
chain	100						
circle		100					
clique			100				
star				100			
multi--hub					100		
semi--clique					3	97	

Table 4.4 *Un-mutated tangles classification confusion matrix*

shape	chain	circle	clique	star	multi--hub	semi--clique	unknown
chain	95	5					
circle		98					2
clique			93		7		
star				95	1		4
multi--hub					100		
semi--clique					4	93	3

Table 4.5 *Mutated tangles classification confusion matrix*

4.7 Classification of Qualitas Corpus Tangles

In order to test the classifier, we applied the classification algorithm presented in Section 4.5. The specifications of the system used to run the experiments is shown in Table 4.6. The execution time of the classifier, including the time required to load and initialise graphs and then apply the classification, is shown in Table 4.7.

Property	Value
Device	iMac 24-inch, Early 2008
Processor	Intel Core 2 Duo
Speed	2.8 GHz
Architecture	64 bit kernel
Memory	4GB
Operating System	Mac OSX 10.7.5 build
JRE settings	1.7.0_21-b12, Java HotSpot(TM) 64-Bit Server VM (build 23.21-b01, mixed mode) Java Virtual Machine (JVM) -Xms40m -Xmx512m

Table 4.6 *System specifications of the workstation used in performing the experiment.*

Tangle type	Time	Avg	Max
Class	41660	8.3	13234
Top-Level-Class	14828	10.4	2501
Weak Package	532	1.7	363
Strong Package	206	0.6	120
All together	59256	9	13234

Table 4.7 *Execution time of the classification algorithm on the corpus in milli-seconds.*

4.7.1 Raw Result Data

For each system in the corpus of Java programs, the dependency graphs are extracted using Massey Architecture Explorer [126]. Then, the class, top-level class, weak package and strong package dependency graphs are generated. We encoded the raw result data in JavaScript Object Notation (JSON) format files, which can be accessed from `http://tanglez.googlecode.com/svn/data/raw/`. Each record contains information about the vertices and edges of each tangle, its type, topology and size, and the values for the various metrics that were used to identify their topologies. Listing 4.8 shows the data structure of an example system composed of two packages and three classes (Figure 4.11).

```
{
  "system": "An example system composed of two packages and three classes",
  "dependencyGraph": {
    "vertices": [
        { "name": "A",  "parent": "p1.A", "namespace": "p1" },
        { "name": "B",  "parent": "p1.B", "namespace": "p1" },
        { "name": "C",  "parent": "p2.C", "namespace": "p2" }
    ],
    "edges": [
        { "src": "p1.A",  "tar": "p1.B" },
        { "src": "p1.B",  "tar": "p2.C" },
        { "src": "p2.C",  "tar": "p1.A" }
    ]
  },
  "ctangles": [
      {
        "id": 1000,
        "topology": "circle",
        "vertices": [ "p1.A", "p2.C", "p1.B" ],
        "edges": [
          { "src": "p1.A",  "tar": "p1.B" },
          { "src": "p1.B",  "tar": "p2.C" },
          { "src": "p2.C",  "tar": "p1.A" }
        ],
        "metrics": {
            "V": 3.0,    "E": 3.0,  "RATIO": 1.0,   "DENSE": 0.0,  "TRANS": 0.0,
          "HUB": 0.0, "STAR": 0.67, "CHAIN": 0.0, "BACKREF": 0.0
        }
      }
  ],
  "tlctangles": [ ... ],
  "wptangles": [ ... ],
  "sptangles": [ ... ]
}
```

Listing 4.8 *Dependency graph, tangles and their topologies of a system stored in JSON data format.*

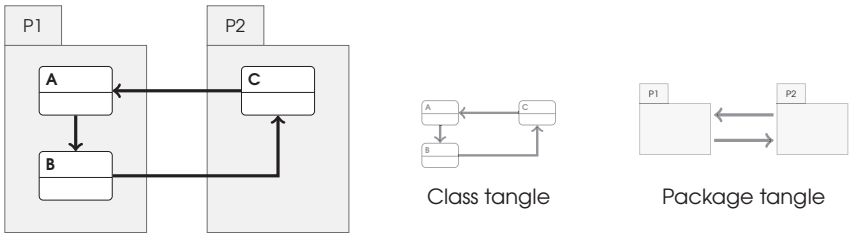


Figure 4.11 *An example dependency graph of the system shown in Listing 4.8.*

4.7.2 The Number of Tangles

The total number of tangles found in each type of graph is shown in Table 4.8 and Figure 4.12. It can be seen that there are significantly more tangles in the class graph than in the top level class graph. This means that most class tangles are caused by inner classes. Surprisingly, there are moderately more strong package tangles (361) than weak package tangles (305), which implies that a package can be a member of

more than one strong package tangle while a package is unique in weak package tangles (Figure 4.13). This indicates that many package tangles may be the result of incorrect package boundaries and therefore non-cohesive packages (Figure 4.14). In many cases, these tangles could be removed by splitting packages or merging the classes from which the strong package tangle was built into one package. There are well-known examples for this scenario. For instance, the `java.util` package contains several clusters of unrelated classes, in particular the collection types, and classes that are part of the date and time API.

shape	class	top level class	weak package	strong package
tiny	2376	715	106	198
circle	723	198	29	34
chain	226	44	11	10
star	927	63	10	11
clique	8	4	7	2
multi-hub	687	366	138	96
semi-clique	10	5	1	1
unknown	47	36	3	9
total	5004	1431	305	361

Table 4.8 *Breakdown of tangles and their topologies.*

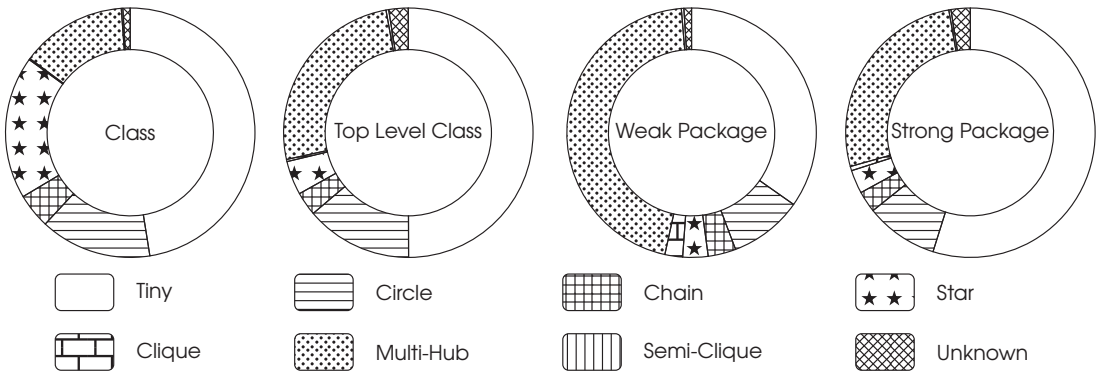


Figure 4.12 *Distribution of tangle topologies on the corpus.*

Figure 4.14 shows the relationship between package boundaries and strong package tangles. The presence of large weak, but not strong, tangles indicates that classes are not placed within correct packages, or perhaps classes needed to be merged into a single package. The dashed line suggests splitting packages to overcome tangles between packages.

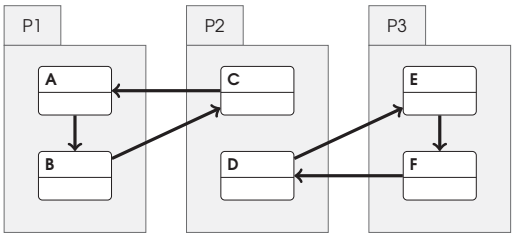


Figure 4.13 *A package can be a member of more than one strong package tangle. In this UML diagram, two strong package tangles:* $\{P_1, P_2\}$, $\{P_2, P_3\}$ *and one weak package tangle* $\{P_1, P_2, P_3\}$ *are extracted from the same dependency graph.*

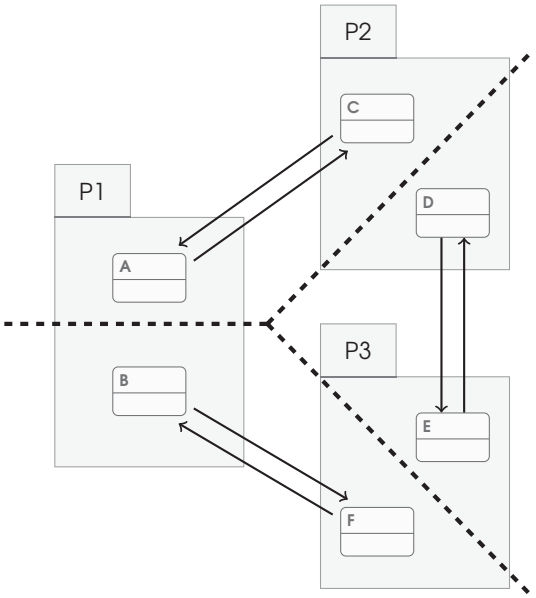


Figure 4.14 *Package boundaries and strong package tangles.*

4.7.3 The Size of Tangles

Tiny tangles represent about half of the class and top level class tangles and about one-third of package tangles. This is good news: these tangles are easy to understand and therefore easy to refactor. In many cases, this could be done by merging the respective artefacts. The average size of tangles in each graph is shown in Tables 4.9 and 4.10. The results summarised in Tables 4.9 and 4.10 indicate that tangles lose symmetry as they get larger. Therefore, large tangles have more irregular topologies.

shape	class			top level class		
	avg	max	sdv	avg	max	sdv
chain	3.26	5.00	0.45	3.09	4.00	0.29
circle	3.03	6.00	0.21	3.12	6.00	0.39
star	7.50	97.00	6.39	7.71	88.00	12.75
clique	4.00	8.00	1.58	4.50	8.00	2.06
multi-hub	50.29	2698.00	174.11	54.33	1196.00	128.87
semi-clique	4.60	6.00	0.80	5.20	6.00	0.75
unknown	5.06	9.00	1.28	6.50	22.00	3.64

Table 4.9 *The size of class tangles.*

shape	weak package			strong package		
	avg	max	sdv	avg	max	sdv
circle	3.03	4.00	0.18	3.09	5.00	0.37
chain	3.00	3.00	0.00	3.10	4.00	0.30
star	5.20	9.00	1.78	4.64	8.00	1.23
clique	3.71	4.00	0.45	3.00	3.00	0.00
multi-hub	19.69	373.00	34.26	16.88	249.00	26.83
semi-clique	14.00	14.00	0.00	4.00	4.00	0.00
unknown	8.33	11.00	3.09	5.56	8.00	1.34

Table 4.10 *The size of package tangles.*

4.7.4 The Occurrence of Topologies

Simple cycles and cliques are rare in all graphs. The few cliques that exist in the package graph are very small. The only common symmetric topology is *chain*, (Figure 4.12). Not surprisingly, there are large number of stars in the class graph that disappear in the top level class graph. These are the stars created by outer classes in the centre, with multiple inner classes attached to them. Outer classes with a single inner class are classified as tiny tangles.

In the four different levels of classification of tangle topologies, the tiny and multi–hub topologies tend to be the most dominant. In addition, the topologies of tangles are not uniformly distributed. Indeed, the distribution of the number of detected tangle topology per system tends to be right skewed, which implies that few projects contain most of the tangles (Figures 4.15).

A significant number of tangles in both package graphs have either a star or multi–hub topology. This could be an indicator that tangles are formed around central core packages. One explanation for this case is that packages tend to grow too large, and sub-packages are created to keep large numbers of classes manageable.

An example where this occurs is the `javax.swing` package in the core Java API (Figure 4.16). Swing contains several rather sophisticated components such as trees (`JTree`) and tables (`JTable`). The APIs of these components contain numerous types

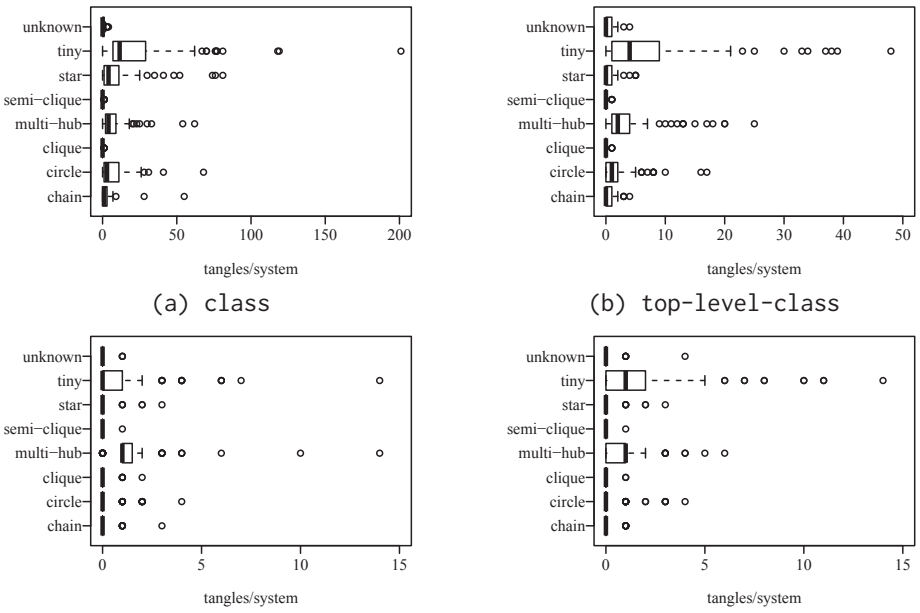


Figure 4.15 *The distribution of tangles topologies on the corpus. Note that the scale of each figure is different. The x-axis show the number of tangles detected per system.*

representing selection models, renderers, data models etc. These types are "outsourced" into sub-packages. However, these sub-packages are closely integrated with the parent package, and therefore form a strong package tangle.

This is also an example of a multi–hub graph. The core AWT package `java.awt` forms a similar star with its sub-packages, but there is a cyclic reference between `java.awt` and `javax.swing` that is caused by a reference from `javax.swing.JComponent` to `java.awt.Component`. Therefore Swing and AWT packages are part of one larger tangle.

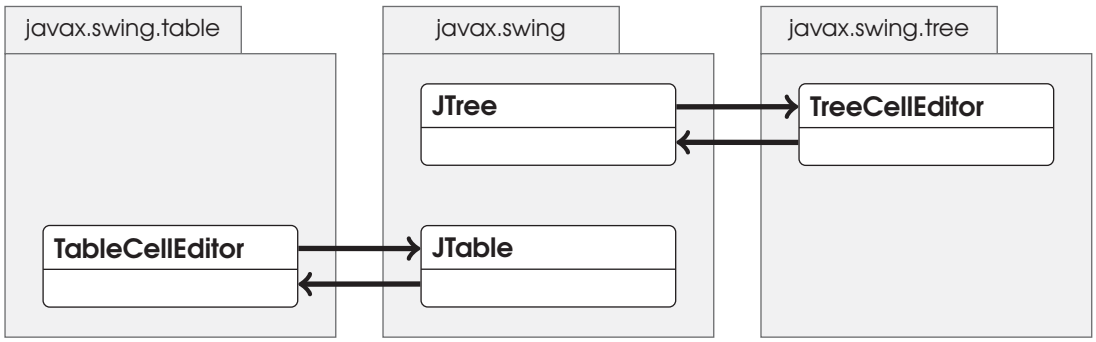


Figure 4.16 *Star-like package dependencies in Swing.*

4.7.5 Tangles Morphology

In order to verify the correctness of the figures presented about the number of tangles in each level, we performed an experiment that counts the number of tangles by shape when extracted from class to top-level-class, from top-level-class to strong package, and from weak package to strong package tangles. Our other interest is to check as to whether weak package tangles and strong package tangles are tightly related. Indeed, the fact that the number of strong package tangles is higher than weak package tangles has some significance in terms of the way classes are placed within packages. In addition, we are interested in verifying the assumption that every strong package tangle is contained within a weak one.

The results of tangles morphology are shown in Tables 4.11 and 4.12 for the change in tangle topology from class to top-level-class tangles and from top-level-class to strong package tangles respectively.

		top-level-class								
	shape	untangled	tiny	circle	chain	star	clique	MH	SC	unknown
class	tiny	1891	485							
	circle	7	4	44						
	chain	531	56		63					
	star	600	89		5	28		1		1
	clique	3					4			
	multi-hub	436	59	1	21	4		170	1	56
	semi-clique	8							10	
	unknown	114	13	3	2			18	1	275

Table 4.11 *Class to top-level-class tangles topologies change in morphology. MH: multi–hub, SC: semi–clique.*

		strong package								
	shape	untangled	tiny	circle	chain	star	clique	MH	SC	unknown
top-level-class	tiny	629	86							
	circle	34	12	1	1					
	chain	79	9		7					
	star	27	5			2				
	clique	4								
	multi-hub	139	32		6	1		8		3
	semi-clique	11	1							
	unknown	146	53	3	13	2	2	46	10	59

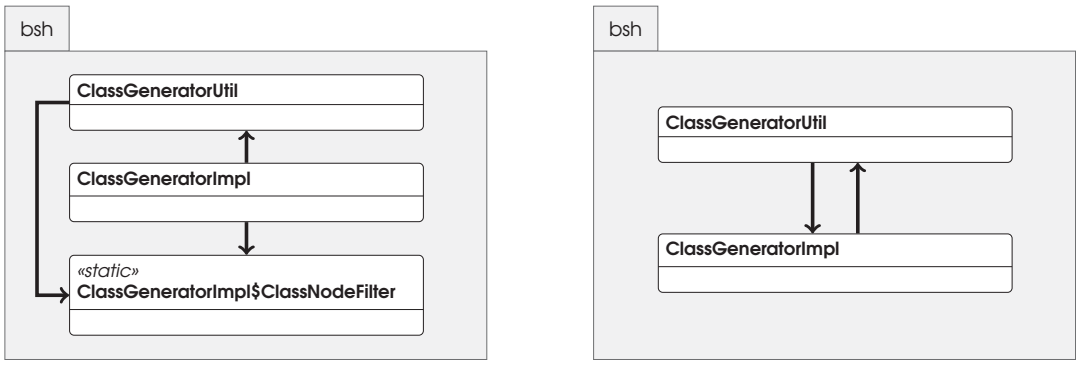
Table 4.12 *Top-level-class to strong package tangles topologies change in morphology. MH: multi–hub, SC: semi–clique.*

The number of untangled class tangles after removing inner classes is 1414, which is less than the number of top-level-class tangles shown in Table 4.8 with 17 tangles. This indicates that not every class tangle can be reduced to top-level-class tangle. For

		strong package								
	shape	untangled	tiny	circle	chain	star	clique	MH	SC	unknown
weak package	tiny	76	32							
	circle	1								
	chain	8	5		5					
	star	1				1				
	clique	1	2							2
	multi-hub	18	35	1	9	1	2	24	6	2
	semi-clique	1	13			1		1	2	3
	unknown	9	111	3	12	2		23	2	55

Table 4.13 *Weak package to strong package tangles topologies change in morphology. MH: multi–hub, SC: semi–clique.*

instance, the class diagram shown in Figure 4.17, which is extracted from the Art of Illusion (AOI-2.8.1), is not strongly connected. However, collapsing inner classes results in a top-level-class tangle. This is due to the fact that the inner class is declared static, and therefore there is a reference from the inner class to its outer class. However, there is not an explicit reference from the inner class to its outer class because the inner class is nested static one; see Section 3.1.3. Top-level-class tangles are formed by collapsing all inner classes in a class graph. More than two thirds of the tangles disappear when moving to a higher level (Figure 4.18). It is not surprising to see tangles change from complex to simpler topologies when moving to a higher level. In an analogy, a microscope can show millions of particles that are not visible to the naked eye. Therefore, the majority of class level tangles collapse to tiny when moving to package level tangles.



(a) Class graph is not tangled. (b) Top-level-class graph is tangled.

Figure 4.17 *Top-level-class tangle derived from non tangled class graph.*

Note that the packages involved in one weak package tangle can make multiple strong package tangles (Figures 4.13 and 4.16). Unlike the other morphological change tables from class to top-level-class and from top-level-class to strong package, which are

one-to-one, the change in morphology from weak package tangles to strong package tangles (Table 4.13) can be one-to-many. The total number of strong package tangles generated from weak package tangles is 361, which is equal to the number counted directly from the dependency graph. Therefore, the untangled number in this table refers to weak package tangles that contains no strong package tangles in them. The summary of morphology results can be seen in Figure 4.18.

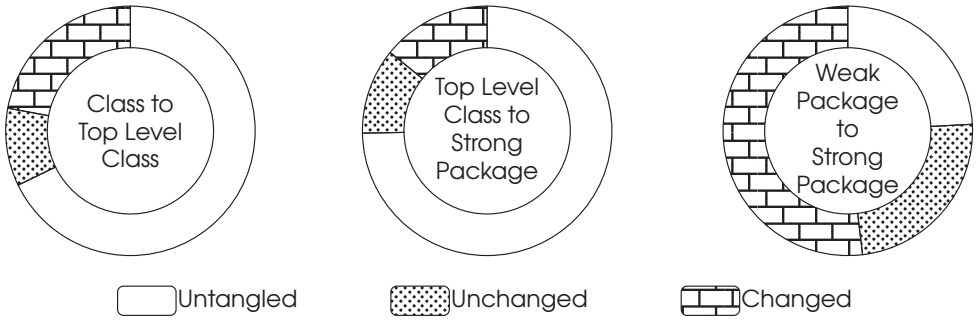


Figure 4.18 *Summary of tangles change in morphology. The ratio of changed tangles topologies in the weak package to strong package is questionable because it is not one-to-one.*

A total of 3590 class tangles disappeared when moving from class level to top-level-class tangles, which suggests that the majority of class tangles are formed by their inner classes. In addition, 1069 top-level-class tangles disappeared when moving to strong package tangles level. In both levels of tangle morphology, about one-tenth of the tangles remain in the same topology.

4.8 Conclusion

In conclusion, this chapter is an attempt to answer the first two research questions which are related to the shapes of tangles and the ability to construct an efficient algorithm to classify tangles based on their shapes. In particular, we presented the reference topologies we used and the criteria followed to pick tangle shapes which are divided based on the symmetry of tangles. The method to generate each tangle has been presented. We also listed our contribution with some computationally efficient metrics. We use these metrics in addition to other metrics defined in the background chapter to build the classifier. Some results have been shown by using tabulated data and statistical figures related to the distribution of tangles on different levels of dependencies.

In the next chapter, we discuss the methods we followed to find the relationship between the shape of tangles and the package containment trees.

CHAPTER 5

TANGLES AND THE PACKAGE CONTAINMENT TREE

“ If a cluttered desk is a sign of a cluttered mind, of what, then, is an empty desk a sign?”
[Albert Einstein]

Contents

Summary – In this chapter, we attempt to answer the third research question, which is about finding the relationship between the shape of tangles and the package containment trees. We devise some metrics that allow us to check whether tangles are confined to branches of the package containment tree, and analyse the shapes of these tangles. We introduce a new measure to verify the assumption made by Laval et al. [25] that the majority of cycles in package dependency graphs form in branches of the package containment tree, and are therefore not critical.

5.1 Introduction

Software systems have different levels of packaging that are used to organise and sort modules that interact together to build a larger system. In Java programs, multiple classes can be bundled into what is called JARs. A single JAR can contain classes and packages of classes. Packages can be used to group classes in order to improve access time and avoid naming conflicts. Furthermore, two or more classes cannot have the same name if placed under the same directory. In order to overcome this problem, classes can be organised into namespaces or packages. Gosling et al. [80] suggest that packages should be organised into a hierarchically structure. A package `S` is considered as a sub-package of another package `P` if its fully qualified name is `P.S`. We assume that systems in the Qualitas Corpus [135] follow the common convention for naming packages.

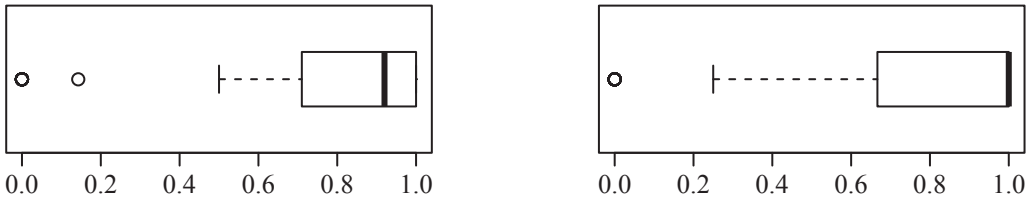
5.2 Parent Centrality

We used the parent centrality measure to validate the correctness of the assumption made by Laval et al. [25], which states that the majority of cyclic dependencies occur within branches of the package containment tree. *Parent centrality* is a centrality measure that is expressed as a ratio between the number of edges that link child packages and their parents to the number of edges that move from a child with lower betweenness to a parent with a higher betweenness measure. To illustrate, let $T_p = (V_p, E_p)$ be a package tangle. *Child to parent* is a set of edges $E_{cp} \subseteq E_p$ given that c is a sub-package of p and $(c, p) \in E_p$. *Child to parent of higher betweenness centrality* (B_C) is a set of edges defined as $\mathcal{E}^+_{cp} = \{(c', p') \in E_p | B_C(c') < B_C(p')\}$. The parent centrality can be measured using Equation 5.1. The result of calculating the parent centrality on weak and strong package tangles on the corpus is shown in Figure 5.1. In both tangle types, it can be seen that P_C is close to 1.0, which suggests that hubs usually form between child and parent packages. Note that tiny tangles are not included because they only contain two vertices of the same betweenness. Moreover, tangles that include no relationships from child to parent packages are not considered.

$$P_C(T_p) = \frac{|\mathcal{E}^+_{cp}|}{|E_{cp}|} \tag{5.1}$$

Out of the 305 weak package tangles extracted from the programs in the Qualitas Corpus, we found that P_C is undefined for 52 tangles (17%), and 1 for 73 tiny tangles. The distribution of values for the remainder is shown in Figure 5.1.a. Out of the 361 strong package tangles, we found that P_C is undefined for 92 tangles (25%) and 1 for

124 tiny tangles. The distribution of values for the remainder is shown in Figure 5.1.b.



(a) weak package (b) strong package

Figure 5.1 *Parent centrality results on the corpus.*

5.3 The Shape of Tangles and the Package Containment Tree

We investigated the question of whether there is evidence in the data set that tangles often consist of closely related packages and can therefore be regarded as less critical. While there is no objective measure to quantify the (semantic) relatedness of packages, the hierarchical structure of package names can be used to approximate this relationship, assuming that packages with similar names are more closely related than packages with very different names. It has been argued [25] that tangles consisting of more closely related packages are often the result of splitting large packages for better readability and are thus not critical. Packages with common parents are thought to be semantically related and relatively coherent. For instance, packages such as `javax.swing` and `javax.swing.tree` depend on each other. However, this relationship is certainly less critical than the mutual dependency between `javax.swing` and `java.awt`. This example also shows the weakness of this approach: the two (user interface) packages `javax.swing` and `java.awt` are certainly "closer" than `java.awt` and `java.sql`.

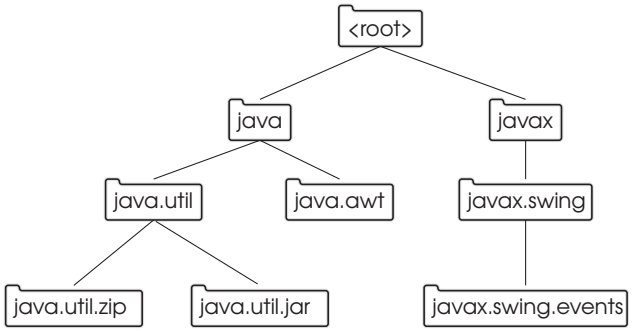


Figure 5.2 *The package containment tree for some selected core Java packages.*

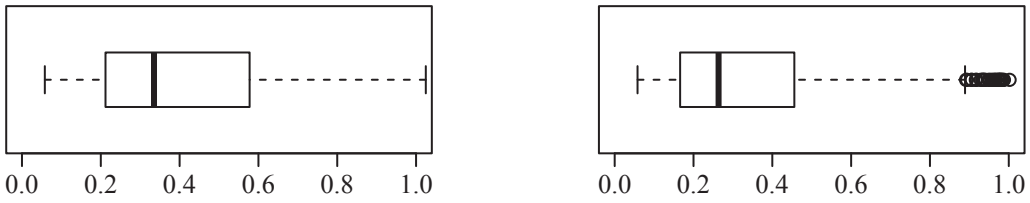
The shortest path length can be used to measure closeness within the package

containment tree. In particular, two metrics are of interest - the diameter (DIAM, longest shortest path) indicating the maximum distance between packages within the PCT, and the Average Shortest Path Length (ASPL). For instance, the diameter of the PCT shown in Figure 5.2 is 6 (the distance between `java.util.zip` and `javax.swing.events`), while the ASPL is 2.82. Note that ASPL works better for projects where all packages share a long common prefix. We apply both metrics to each tangle and measure the ratios between the respective metrics for the tangle and the respective metrics for the entire graph.

This gives us a notion of a relative closeness of the packages within tangles. For a given tangle t within a graph g, we call $DIAM(t)/DIAM(g)$ diameter closeness (DCLOSE(t)), and $ASPL(t)/ASPL(g)$ average closeness (ACLOSE(t)).

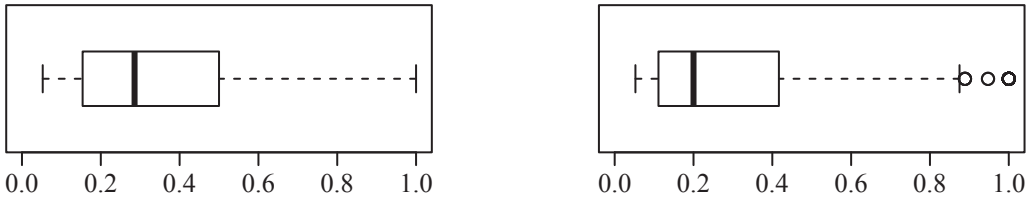
Tangle type	average DCLOSE	average ACLOSE
weak	0.37	0.42
strong	0.29	0.34

Table 5.1 *Package containment tree metrics.*



(a) weak package (b) strong package

Figure 5.3 *ACLOSE distribution in package tangles.*



(a) weak package (b) strong package

Figure 5.4 *DCLOSE distribution in package tangles.*

The results shown in Table 5.1 are computed by taking the average of the respective values for all tangles in all graphs within the data set. The results show that packages within a tangle are more closely related than randomly chosen packages within the same graph. In other terms, tangles tend to form in branches of the PCT. This effect is slightly stronger for strong package tangles. This reflects the fact that these tangles

are more coherent as they are created by class tangles. If we accept the assumption by Laval et. al. [25] that tangles from packages within branches of the PCT are less critical, then we can conclude that many tangles are not critical.

5.4 Conclusion

In summary, we have seen in Chapter 4 that multi–hubs and star tangles shapes are the most common shapes if tiny tangles are excluded. The next thing that comes to mind is to verify if there are any semantics in the presence of such hubs in these tangles. The reason behind this was to find out if the cyclic references are made mainly between parent packages and their children and grandchildren. If this is the case, then these tangles are not critical according to Laval et al. [25]. We introduced a new measure that we called the *"parent centrality"* which tests the flow of dependencies from child to parent packages using betweenness centrality. We also used and applied the concept of the package containment tree on the package tangles extracted from the Qualitas corpus and measured the diameter and average shortest path. These two measures tend to be small, which verifies the assumption that package tangles tend to form in branches of the PCT.

In the next chapter, we summarise the key findings of our research and present some conclusions and possible future directions of our research.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

❝ As long as there were no machines, programming was no problem at all; when we had weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem.❞
[E. W. Dijkstra]

Contents

Summary – In this chapter, we conclude our research and present a summary of the key points of this research. We show the methods and experiments conducted to answer the research questions and point out the areas that might be considered as a threat to validity. Finally, we provide some future directions for this research.

6.1 Conclusions

Software systems grow in size and complexity over time. According to some researchers, complexity may not be easily separated from the software system as it evolves. Some of the new trends in software engineering are concerned with the aspect of modularity as an approach to taming the complexity of software systems. Some of the methods followed to control the complexity of software are the use of layered design and avoiding cyclic references between compilation units. Parnas's advice of "avoiding cycles", which dates back to early 1970s [15], seems to be ignored. Cyclic dependencies have been considered a violation to proper design due to their effects on testability, deployment and maintainability. Indeed, applying changes becomes an issue with software systems that contain such a violation. This can be explained by the ripple effect of change from one module to another. Specifically, the change spreads everywhere in the presence of cyclic references and worse, the change might be applied more than once to a single module. Another performance–related issue raised about cyclic dependencies is the question of which module gets loaded before the other if they depend on each other. Therefore, the shape of tangles can play an important role in determining the degree to which a tangle is worse than the other. Nearly none of the software systems used in this study are cycle–free. If cyclic dependencies were detrimental to software quality and suppress software evolution, then such systems would have never been considered modular and successful. Therefore, it might be logical to rank such cyclic dependencies based on desirability.

We have studied the topology of tangles formed by cyclic dependencies in different types of dependency graphs extracted from Java programs by deriving a set of metrics that classifies different types of tangles and demonstrating this approach on a large, representative data set that has been widely used in empirical studies. The results of this analysis show that regular topologies are rare, that a significant number of tangles are trivial (that is, they are tiny), and that many tangles have a structure that features one or multiple hub nodes.

We have also found evidence that tangles tend to form in branches of the package containment tree, which seems to indicate that there are many cases where tangles are formed from parent packages together with their sub–packages. This raises the question of which packages are critical and which should be removed through refactoring. The prime candidates seem to be multi–hub tangles. An example for this is the tangle in the strong package dependency graph extracted from the JRE version 1.6 that contains two hubs: `java.awt` and `javax.swing`.

The distribution of tangles topologies shows that more than half of them are tiny.

This raises the question of whether ignoring multiple edges was a good decision or not. In addition, we do not differentiate between `uses` and `extends` as the relationship between modules.

In Section 3.3 (p.40), we presented a case where the Java compiler copies constants values to referencing classes instead of making a dynamic reference. This limitation of the compiler may affect the shape of dependency graphs and also the tangles extracted from those dependency graphs. Furthermore, if such references were dynamic and detectable on the byte-code level, then we might have more cross-tree references than what has been reported.

The results of our study have been taken by analysing open–source programs. Our assumption is that the design and the architecture of the used programs in the corpus simulate real world business or commercial applications.

6.2 Future Work

In our analysis we have used only unweighted graphs. In particular, when investigating package tangles, it could be beneficial to add weights to edges to represent the number of inter-class relationships that cause package dependencies. For the example discussed in Section 5.3 (p.71), this would show a significant asymmetry: the dependency of the Swing package on the Abstract Window Toolkit (AWT) package is much stronger than the dependency of the AWT package on the Swing package (Figure 1.9 (p.8)). Another area of interest is to investigate how change propagates through different topologies.

The types of dependency graphs used in this research are only class and package levels. Class level graphs can be very large and contain too much detail. On the other hand, package graphs hide a large bulk of information. Although the top-level-class dependency graphs are relatively smaller than class graphs, the number of classes in one package is not controlled. We would consider using another type of dependency graph that we would call *"Boundary Dependency Graph"*, which would show the relationships between anchor classes. A class is considered boundary if it has the greatest number of incoming or outgoing references from outside the package within which it resides. Therefore, a package may have at most two or three boundary classes. This dependency graph can identify the entry points between packages. Cyclic dependencies within packages are less problematic than those spanned over different packages. We plan to see the distribution of topologies formed by this kind of graph on the Qualitas Corpus and find out its significance.

Apart from linking cyclic dependencies to the package containment tree, this research is mainly focused on the shape of cyclic dependencies. Therefore, the next

logical step is to study the semantics of such dependencies. We expect that in the future our study will lead to better tools for distinguishing critical cyclic dependencies from cyclic dependencies that are less problematic. It provides a different, and complementary, view of software tangles to that of Laval et al. In the future, we plan to combine these views in order to provide a deeper understanding of the underlying structure of detrimental cyclic dependencies and how they can be avoided.

Another future direction of this research can be to determine the fault tolerance of each of the tangle topologies that have been considered in the classification process. A common argument against cyclic dependencies is that change causes ripple effects. Clearly a very dense structure is worse than a sparse one. However, there is still not enough information about what lies in between these extremes, such as chains and stars. In medical fields, the rate of propagation is used to assess a certain infectious disease which can be applied in a similar fashion to the way bugs spread between software modules. Disease propagation models could be used to simulate these ripple effects for different topologies.

BIBLIOGRAPHY

[1] 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 *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 50–.

[2] R. S. Pressman, *Software Engineering: A Practitioner's Approach (McGraw-Hill Series in Computer Science)*. McGraw Hill College Division, 2004.

[3] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1, Mar. 2007.

[4] F. P. Brooks, Jr., *The Mythical Man-Month*. New York, NY, USA: ACM, Apr. 1975, vol. 10.

[5] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[6] M. M. Lehman, "Laws of software evolution revisited," in *Proceedings of the 5th European Workshop on Software Process Technology*, ser. EWSPT '96. London, UK, UK: Springer-Verlag, 1996, pp. 108–124.

[7] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," in *Proceedings of the 4th International Symposium on Software Metrics*, ser. METRICS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 20–, http://dl.acm.org/citation.cfm?id=823454.823901.

[8] C. Lilienthal, "Architectural complexity of large-scale software systems," in *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, ser. CSMR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 17–26, http://dx.doi.org/10.1109/CSMR.2009.16.

[9] T. Zimmermann and C. Bird, "Collaborative software development in ten years: Diversity, tools, and remix culture," in *Proceedings of the Workshop on the Future of Collaborative Software Development*, 2012.

[10] L. O'Brien. (2005, Dec) Knowing. net – how many lines of code in windows? [Online]. Available: http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/

[11] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 449–451.

[12] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*. Elsevier Science Ltd/North-Holland, Jun 1976.

[13] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[14] E. W. Dijkstra, "The structure of the-multiprogramming system," *Communications of the ACM*, vol. 26, no. 1, pp. 49–52, 1983.

[15] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[16] D. L. Parnas, "On a 'buzzword': Hierarchical structure," in *Software Pioneers*, M. Broy and E. Denert, Eds. New York, NY, USA: Springer-Verlag New York, Inc., 2002, pp. 429–440.

[17] W. Stevens, G. Myers, and L. Constantine, *Structured Design*. Upper Saddle River, NJ, USA: Yourdon Press, 1979.

[18] H. Zimmermann, "OSI reference model - the ISO model of architecture for open systems interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, 1980.

[19] N. Briscoe, "Understanding the OSI 7-layer model," *PC Network Advisor*, vol. 120, no. 2, 2000.

[20] D. L. Parnas, "Designing software for ease of extension and contraction," in *Proceedings ICSE '78*. Piscataway, NJ, USA: IEEE Press, 1978, pp. 264–277.

[21] R. C. Martin, "Design principles and design patterns," *Object Mentor*, pp. 1–34, 2000.

[22] H. Melton and E. Tempero, "An empirical study of cycles among classes in Java," *Empirical Software Engineering*, vol. 12, no. 4, pp. 389–415, Aug 2007.

[23] J. Dietrich, C. McCartin, E. Tempero, and S. A. M. Shah, "Barriers to modularity - an empirical study to assess the potential for modularisation of Java program," in *Proceedings Qosa'10*. Springer-verlag Berlin Heidelberg, 2010, to Appear.

[24] H. Melton and E. Tempero, "Static members and cycles in Java software," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 2007, pp. 136–145.

[25] J. Laval, J. Falleri, P. Vismara, and S. Ducasse, "Efficient retrieval and ranking of undesired package cycles in large software systems," *Journal of Object Technology*, vol. 11, no. 1, pp. 4:1–24, Apr 2012.

[26] J. Dietrich. (2011) Dependency analysis and the modularisation of Java programs. [Online]. Available: http://java.dzone.com/articles/dependency-analysis-and-1

[27] M. Newman, *Networks: An Introduction*. New York, NY, USA: Oxford University Press, Inc., 2010.

[28] A. Xie and P. A. Beerel, "Implicit enumeration of strongly connected components," in *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design*. IEEE Press, 1999, pp. 37–40.

[29] M. Dellnitz and O. Junge, "Set oriented numerical methods for dynamical systems," *Handbook of Dynamical Systems*, vol. 2, pp. 221–264, 2002.

[30] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers & Mathematics with Applications*, vol. 7, no. 1, pp. 67–72, 1981.

[31] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM SIGPLAN-sigact Symposium on Principles of Programming Languages*. ACM, 1981, pp. 207–218.

[32] J. Purdom, Paul, "A transitive closure algorithm," *BIT Numerical Mathematics*, vol. 10, no. 1, pp. 76–94, 1970.

[33] E. W. Dijkstra, E. W. Dijkstra, E. W. Dijkstra, and E. W. Dijkstra, *A Discipline of Programming*. Prentice-hall Englewood Cliffs, 1976, vol. 1.

[34] I. Munro, "Efficient determination of the transitive closure of a directed graph," *Information Processing Letters*, vol. 1, no. 2, pp. 56–58, 1971.

[35] J. Cheriyan and K. Mehlhorn, "Algorithms for dense graphs and networks on the random access computer," *Algorithmica*, vol. 15, no. 6, pp. 521–549, 1996.

[36] H. N. Gabow, "Path-based depth-first search for strong and biconnected components," *Information Processing Letters*, vol. 74, no. 3-4, pp. 107–114, May 2000.

[37] R. Tarjan, "Depth-first search and linear graph algorithm," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[38] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding strongly connected components in distributed graphs," *J. Parallel Distrib. Comput.*, vol. 65, no. 8, pp. 901–910, aug 2005.

[39] J. Barnat, J. Chaloupka, and J. Van De Pol, "Distributed algorithms for scc decomposition," *J. Log. and Comput.*, vol. 21, no. 1, pp. 23–44, Feb. 2011.

[40] L. Fleischer, B. Hendrickson, and A. Pinar, "On identifying strongly connected components in parallel," in *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, ser. IPDPS '00. London, UK, UK: Springer-Verlag, 2000, pp. 505–511.

[41] S. D. Nikolopoulos and L. Palios, "On the parallel computation of the biconnected and strongly connected co-components of graphs," *Discrete Appl. Math.*, vol. 155, no. 14, pp. 1858–1877, Sep. 2007.

[42] J. Barnat and P. Moravec, "Parallel algorithms for finding sccs in implicitly given graphs," in *Proceedings of the 11th International Workshop, FMICS 2006 and 5th International Workshop, PDMC Conference on Formal Methods: Applications and Technology*, ser. FMICS'06/PDMC'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 316–330.

[43] J. Barnat, J. Chaloupka, and J. van de Pol, "Improved distributed algorithms for scc decomposition," *Electron. Notes Theor. Comput. Sci.*, vol. 198, no. 1, pp. 63–77, Feb. 2008.

[44] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Computing strongly connected components in parallel on cuda," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 544–555.

[45] E. Nuutila and E. Soisalon-Soininen, "On finding the strongly connected components in a directed graph," *Inf. Process. Lett.*, vol. 49, no. 1, pp. 9–14, Jan. 1994.

[46] R. Bloem, H. N. Gabow, and F. Somenzi, "An algorithm for strongly connected component analysis in n log n symbolic steps," in *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '00. London, UK, UK: Springer-Verlag, 2000, pp. 37–54.

[47] R. Gentilini, C. Piazza, and A. Policriti, "Computing strongly connected components in a linear number of symbolic steps," in *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '03. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 573–582.

[48] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, Jun. 2005.

[49] M. Fowler, "Reducing coupling," *IEEE Softw.*, vol. 18, no. 4, pp. 102–104, Jul. 2001.

[50] M. Fowler, "Refactoring: Improving the design of existing code," in *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*. London, UK, UK: Springer-Verlag, 2002, pp. 256–.

[51] S. W. Ambler, A. Vermeulen, and G. Bumgardner, *The Elements of Java Style*. New York, NY, USA: Cambridge University Press, 1999.

[52] S. M. A. Shah, J. Dietrich, and C. McCartin, "Making smart moves to untangle programs," in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, ser. CSMR '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 359–364.

[53] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[54] V. Ramachandran, "Finding a minimum feedback arc set in reducible flow graphs," *J. Algorithms*, vol. 9, no. 3, pp. 299–313, Sep. 1988.

[55] P. Eades, X. Lin, and W. F. Smyth, "A fast and effective heuristic for the feedback arc set problem," *Inf. Process. Lett.*, vol. 47, no. 6, pp. 319–323, Oct. 1993.

[56] P. Charbit, S. Thomassé, and A. Yeo, "The minimum feedback arc set problem is np-hard for tournaments," *Comb. Probab. Comput.*, vol. 16, no. 1, pp. 1–4, Jan. 2007.

[57] G. Even, J. S. Naor, B. Schieber, and M. Sudan, "Approximating minimum feedback sets and multicuts in directed graphs," *Algorithmica*, vol. 20, no. 2, pp. 151–174, 1998.

[58] D. Younger, "Minimum feedback arc sets for a directed graph," *Circuit Theory, IEEE Transactions on*, vol. 10, no. 2, pp. 238–245, 1963.

[59] C. Demetrescu and I. Finocchi, "Combinatorial algorithms for feedback problems in directed graphs," *Inf. Process. Lett.*, vol. 86, no. 3, pp. 129–136, may 2003.

[60] L. Roditty and V. Vassilevska Williams, "Fast approximation algorithms for the diameter and radius of sparse graphs," in *Proceedings of the 45th Annual ACM Symposium on Symposium on Theory of Computing*, ser. STOC '13. New York, NY, USA: ACM, 2013, pp. 515–524.

[61] D. B. West, *Introduction to Graph Theory*, 2nd ed. Pearson, 2001.

[62] A. Ganesh and F. Xue, "On the connectivity and diameter of small-world networks," *Advances in Applied Probability*, vol. 39, no. 4, pp. 853–863, 2007.

[63] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[64] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *J. ACM*, vol. 24, no. 1, pp. 1–13, Jan. 1977.

[65] D. Karger, R. Motwani, and G. Ramkumar, "On approximating the longest path in a graph," *Algorithmica*, vol. 18, no. 1, pp. 82–98, 1997.

[66] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, Mar. 1977.

[67] M. E. Newman, "A measure of betweenness centrality based on random walks," *Social Networks*, vol. 27, no. 1, pp. 39–54, 2005.

[68] D. R. White and S. P. Borgatti, "Betweenness centrality measures for directed graphs," *Social Networks*, vol. 16, no. 4, pp. 335–346, 1994.

[69] F. Cowell, *Measuring Inequality (LSE Perspectives in Economic Analysis)*. Oxford University Press, USA, 2011.

[70] D. A. Verstegen, "Concepts and measures of fiscal inequality: A new approach and effects for five states." *Journal of Education Finance*, no. 2, p. 145, 1996.

[71] R. Bendel, S. Higgins, J. Teberg, and D. Pyke, "Comparison of skewness coefficient, coefficient of variation, and gini coefficient as inequality measures within populations," *Oecologia*, vol. 78, no. 3, pp. 394–400, 1989.

[72] R. Dorfman, "A formula for the gini coefficient," *The Review of Economics and Statistics*, vol. 61, no. 1, pp. 146–149, 1979.

[73] S. Yitzhaki, "Relative deprivation and the gini coefficient," *The Quarterly Journal of Economics*, pp. 321–324, 1979.

[74] H. Theil, *Economics and information theory.*, ser. Studies in mathematical and managerial economics: v. 7. Amsterdam, North-Holland Pub. Co.; Chicago, Rand McNally, 1967., 1967.

[75] J. G. Palma, "Globalizing inequality: 'Centrifugal' and 'centripetal' forces at work," *Revue Tiers Monde*, no. 35, Sep 2006.

[76] E. M. Hoover, "The measurement of industrial localization," *The Review of Economics and Statistics*, vol. 18, no. 4, pp. 162–171, 1936.

[77] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345–, Jun. 1962.

[78] T. Schank and D. Wagner, "Approximating clustering coefficient and transitivity," *Journal of Graph Algorithms and Applications*, vol. 9, p. 2005, 2005.

[79] A. Lubiw, "Some np-complete problems similar to graph isomorphism." *SIAM J. Comput.*, vol. 10, no. 1, pp. 11–21, 1981.

[80] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison-Wesley, 2005.

[81] E. Hautus, "Improving Java software through package structure analysis," in *6th IASTED International Conference on Software Engineering and Applications*, 2002.

[82] O. Dictionaries, *Oxford Dictionary of English*. New York, NY , U.S.A.: Oxford: Oxford University Press, 2010.

[83] N. Betzler, M. R. Fellows, C. Komusiewicz, and R. Niedermeier, "Parameterized algorithms and hardness results for some graph motif problems," in *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching*, ser. CPM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 31–43.

[84] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE, 2001, pp. 313–320.

[85] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: Simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, October 2002.

[86] J. Dietrich and C. McCartin, "Scalable motif detection and aggregation," in *Proceedings of the Twenty-third Australasian Database Conference-volume 124*. Australian Computer Society, Inc., 2012, pp. 31–40.

[87] J. Dietrich, C. McCartin, E. Tempero, and S. M. A. Shah, "On the existence of high-impact refactoring opportunities in programs," in *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*, ser. ACSC '12. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2012, pp. 37–48.

[88] H. Bunke and K. Riesen, "A family of novel graph kernels for structural pattern recognition," in *Progress in Pattern Recognition, Image Analysis and Applications*. Springer, 2007, pp. 20–31.

[89] H. Kashima and A. Inokuchi, "Kernels for graph classification," in *ICDM Workshop on Active Mining*, vol. 2002. Citeseer, 2002.

[90] G. Li, M. Semerci, B. Yener, and M. J. Zaki, "Graph classification via topological and label attributes," in *9th Workshop on Mining and Learning with Graphs (with SIGKDD)*, Aug. 2011.

[91] G. Li, M. Semerci, B. Yener, and M. J. Zaki, "Effective graph classification based on topological and label attributes," *Stat. Anal. Data Min.*, vol. 5, no. 4, pp. 265–283, Aug. 2012.

[92] T. Kudo, E. Maeda, and Y. Matsumoto, "An application of boosting to graph classification," in *Advances in Neural Information Processing Systems*, 2004, pp. 729–736.

[93] L. Zhu, W. K. Ng, and S. Han, "Classifying graphs using theoretical metrics: A study of feasibility," in *Proceedings of the 16th International Conference on Database Systems for Advanced Applications*, ser. DASFAA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 53–64.

[94] H. Melton and E. Tempero, "The crss metric for package design quality," in *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*,

ser. ACSC '07. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2007, pp. 201–210.

[95] H. Melton and E. Tempero, "Towards assessing modularity," in *Assessment of Contemporary Modularization Techniques, 2007. ICSE Workshops Acom'07. First International Workshop on*. IEEE, 2007, pp. 3–3.

[96] H. Melton, "On the usage and usefulness of oo design principles," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. ACM, 2006, pp. 770–771.

[97] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, Jul. 1995.

[98] C. Bauer and G. King, *Hibernate in Action (In Action Series)*. Greenwich, CT, USA: Manning Publications Co., 2004.

[99] F. Bourqun and R. K. Keller, "High-impact refactoring based on architecture violations," in *Proceedings CSMR '07*. Washington, Dc, USA: IEEE Computer Society, 2007, pp. 149–158.

[100] F. Simon, F. Steinbrueckner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings CSMR'01*. IEEE Computer Society, 2001, p. 30.

[101] H. Melton and E. Tempero, "Identifying refactoring opportunities by identifying dependency cycles," in *Proceedings of the 29th Australasian Computer Science Conference-volume 48*. Australian Computer Society, Inc., 2006, pp. 35–41.

[102] N. 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*. New York, NY, USA: ACM, 2007, pp. 949–950.

[103] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui, "Automatic package coupling and cycle minimization," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 103–112.

[104] J. Dietrich, C. McCartin, E. Tempero, and S. Shah, "on the detection of high-impact refactoring opportunities in program," *Arxiv Preprint Arxiv:1006.1747*, 2010.

[105] S. M. A. Shah, J. Dietrich, and C. McCartin, "On the automated modularisation of Java programs using service locators," in *Software Composition*, ser. Lecture

Notes in Computer Science, T. Gschwind, F. Paoli, V. Gruhn, and M. Book, Eds. Springer Berlin Heidelberg, 2012, vol. 7306, pp. 132–147.

[106] S. Jungmayr, "Testability measurement and software dependencies," in *Proceedings of the 12th International Workshop on Software Measurement*, 2002, pp. 179–202.

[107] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, 1987.

[108] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *ACM SIGPLAN Notices*, vol. 19. ACM, 1984, pp. 177–184.

[109] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proceedings of the 14th International Conference on Software Engineering*. ACM, 1992, pp. 392–411.

[110] F. Balmas, "Displaying dependence graphs: A hierarchical approach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 3, pp. 151–185, 2004.

[111] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[112] B. G. Ryder, "Constructing the call graph of a program," *Software Engineering, IEEE Transactions on*, no. 3, pp. 216–226, 1979.

[113] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[114] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java(TM) Language Specification – Java SE 7 Edition (Java Series)*. Addison-Wesley Professional, 2013.

[115] J. Tessier. Dependency Finder. [Online]. Available: http://depfind.sourceforge.net/

[116] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[117] D. E. Knuth, "A generalization of dijkstra's algorithm," *Information Processing Letters*, vol. 6, no. 1, pp. 1–5, 1977.

[118] H. Melton and E. Tempero, "Jooj: Real-time support for avoiding cyclic dependencies," in *Proceedings of the Thirtieth Australasian Conference on Computer Science - Volume 62*, ser. ACSC '07. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2007, pp. 87–95.

[119] F.-J. E. Elmer. (2012) Classycle: Analysing tools for Java class and package dependencies. [Online]. Available: http://classycle.sourceforge.net/

[120] G. Inc. (2013, sep) Java developer tools: CodePro Analytix – dependency analysis. [Online]. Available: https://developers.google.com/java-dev-tools/codepro/doc/

[121] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 602–.

[122] M. Clark. (2009, sep) JDepend: Dependency analyser. [Online]. Available: http://clarkware.com/software/JDepend.html

[123] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage software architecture," in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 164–165.

[124] J. Laval and J. Falleri. (2012) Popsycle. [Online]. Available: https://popsycle.googlecode.com/

[125] S. Sarkar, G. M. Rama, and R. Shubha, "A method for detecting and measuring architectural layering violations in source code," in *Software Engineering Conference, 2006. Apsec 2006. 13th Asia Pacific*. IEEE, 2006, pp. 165–172.

[126] J. Dietrich. (2012, sep) Massey architecture explorer. [Online]. Available: http://xplrarc.massey.ac.nz/

[127] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of Java code for empirical studies," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, ser. APSEC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 336–345.

[128] D. B. Johnson, "Finding all the elementary circuits of a directed graph." *SIAM Journal of Computing*, no. 1, p. 77, 1975.

[129] R. Tarjan, "Enumeration of the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 2, no. 3, pp. 211–216, 1973.

[130] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "Comparative analysis of evolving software systems using the gini coefficient," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 179–188.

[131] B. Vasilescu, A. Serebrenik, and M. van den Brand, "By no means: A study on aggregating software metrics," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, ser. WETSoM '11. ACM, 2011, pp. 23–26.

[132] P. D. Allison, "Measures of inequality," *American Sociological Review*, vol. 43, no. 6, pp. 865–880, 1978.

[133] R. I. Lerman and S. Yitzhaki, "A note on the calculation and interpretation of the gini index," *Economics Letters*, vol. 15, no. 3, pp. 363–368, 1984.

[134] S. V. Stehman, "Selecting and interpreting measures of thematic classification accuracy," *Remote sensing of Environment*, vol. 62, no. 1, pp. 77–89, 1997.

[135] Q. R. Group, "Qualitas corpus," The University of Auckland, 2013. [Online]. Available: http://www.qualitascorpus.com/

APPENDICES

APPENDIX A

GLOSSARIES

Directed Graph

In graph theory, a directed graph a.k.a DiGraph $G = (V, E)$ is a graph composed of finite set of vertices V and edges E. Each edge is a binary nonsymmetric relation between two distinct vertices. Therefore, the set of edges can be mathematically noted as $E = \{(u, v)|u, v \in V\} \subset V \times V$. In this study, graphs are *simplicial*, i.e., loops (reflexive relations) and multiple edges are not considered.

Shortest Path

A walk (sequence of vertices) from any vertex v to any other vertex u where $v, u \in V$ is called a path. It can be noted as $(v_1, v_2, v_3, \cdots, v_k)$ where v_1 and v_k are the source and target nodes respectively and $(v_i, v_{i+1}) \in E$. The path is simple if it has no duplicate vertices. The shortest path between v_i and v_j is the path that links them and has the lowest number of in-between vertices. Since the algorithm used to calculate the shortest path in this study is Dijkstra algorithm, the shortest path is noted as $\delta((s, t), G)$ where s, t are the source and the target vertices respectively. If there is no possible directed path between v_i and v_j, then $\delta((v_i, v_j), G) = \infty$.

Degree

The degree of a vertex noted as $d(v)$ is the number of incident edges on that vertex. In a directed graph, the indegree $d^-(v)$ is the number of edges pointing to v and the outdegree $d^+(v)$ is the number of edges pointing out of v. The degree of a vertex is the sum of its indegree and outdegree $d(v) = d^-(v) + d^+(v)$. The degree of a graph $d(G)$ is the maximum of degree of all vertices.

Circuit or cycle

A circuit is a path that starts and ends at the same node is called a circuit or cycle. A circuit is elementary if it has no duplicate nodes. A circuit is also called a closed path. A graph is cyclic if the number of vertices is equal to the number of edges.

Strongly connected component

A digraph $G = (V, E)$ is strongly connected if G is cyclic and every two distinct vertices are linked by a path, i.e. $\delta((v_i, v_j), G) \neq \infty \wedge \delta((v_j, v_i), G) \neq \infty \; \forall \; v_i, v_j \in V \wedge i \neq j$.

Strongly connectedness is a feature that can be represented in cyclic directed graph when a path exists between any two vertices. Figure A.1 shows a directed graph with three strongly connected components namely $\{\{a, b, e\}, \{f, g\}, \{c, d, h\}\}$. Dotted edges indicate a path that has no back reference. We use the term tangle to denote a directed graph in which every edge has a back reference.

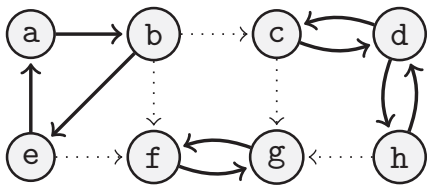


Figure A.1 *Strongly connected components extracted from dependency graph.*

Back Reference Path

In a strongly connected digraph $G = (V, E)$, the back reference path between two vertices v and u where $\{v, u\} \subset V$ such that $(v, u) \in E$ and $v \neq u$ is the shortest path from u to v. It is noted as $backref(v, u) = \delta(u, v)$. The back reference path is immediate if the $(u, v) \in E$. In Figure A.1, the back reference path of (a, b) is $\{(b, e), (e, a)\}$.

Topology

The study of qualitative properties of certain objects (called topological spaces) that are invariant under a certain kind of transformation, especially those properties that are invariant under a certain kind of equivalence (called homeomorphism).

Software Dependency Graph

The dependency graph is a directed graph containing a finite set of nodes (software units) and finite set of edges (dependency relationships) between nodes. Multiple and self dependencies are disregarded in this study.

Software Tangle

A tangle is a collection of software units/components which are linked to each other in a cyclic dependency, i.e. strongly connected.

Minimum Feedback Edge Set (MFES)

MFES is the minimal set of edges needed to be removed from a directed cyclic graph to break all its cycles. In other words, MFES is the minimal edge cut that makes a directed cyclic graph acyclic.

APPENDIX B

ANTLR AND HIBERNATE VERSION DATA

B.1 ANTLR

Date	Version	#P	#D	#C	CR	#SCCs	LSCCS	LSCCR
09-1998	2.4.0	5	7	4	57%	3	3	60%
03-1999	2.6.0	4	6	4	66%	2	3	75%
01-2003	2.7.2	8	19	17	89%	2	7	87%
09-2006	2.7.7	12	29	26	89%	3	10	83%
08-2007	3.0	9	31	31	100%	2	7	77%
05-2008	3.1	27	94	68	72%	12	10	37%
09-2009	3.2	31	116	97	83%	12	10	32%
11-2010	3.3	31	116	96	82%	12	10	32%

Table B.1 *Some measures among different versions of ANTLR system.*

B.2 Hibernate

Date	Version	#P	#D	#C	CR	#SCCs	LSCCS	LSCCR
12-2001	0.8.1	8	25	10	40%	5	4	50%
09-2002	1.1.0	23	87	58	66%	9	15	65%
06-2003	2.0.0	31	184	151	82%	9	23	74%
12-2003	2.1.0	28	201	181	90%	5	24	85%
08-2004	2.1.5	29	205	182	88%	6	24	82%
01-2005	2.1.8	29	215	189	87%	6	24	82%
03-2005	3.0.0	51	442	427	96%	6	44	86%
10-2006	3.2.0	76	684	659	96%	7	70	92%

Table B.2 *Some measures among different versions of Hibernate system.*

#P: number of packages

#D: number of dependencies

#C: number of cyclic references

CR: ratio of cyclic references to all dependencies

LSCCS: size of the largest SCC

LSCCR: ratio of packages in the largest SCC.

APPENDIX C

QUALITAS CORPUS DATASET

The dataset used in this research is a collection of open-source Java software systems that can be obtained from `http://www.qualitascorpus.com/`. The version of corpus used is 20120401. The following are some information about each system in the dataset.

Full Name	Recent version	Domain	Number of Classes	LOC	Release Date
Ant	1.8.2	parsers/generators/make	1286	109127	2010-12-27
ANTLR	3.4	parsers/generators/make	339	27112	2011-07-18
AOI	2.8.1	3D/graphics/media	862	111725	2010-01-03
ArgoUML	0.34	diagram generator/data visualisation	2560	192410	2011-12-15
AspectJ	1.6.9	programming language	2624	412394	2010-07-05
Axion	1.0-M2	database	261	23744	2003-07-11
Vuze	4.7.0.2	database	7249	457152	2011-12-02
Batik SVG Toolkit	1.7	3D/graphics/media	2599	17848	2008-01-09
C-JDBC	2.0.2	database	586	81306	2005-09-16
Castor	1.3.1	middleware	1327	115543	2010-01-03
Apache Cayenne	3.0.1	database	2171	127529	2010-08-26
Checkstyle	5.1	IDE	349	23316	2010-02-16
Cobertura	1.9.4.1	testing	122	51860	2010-03-03
Apache Commons Collections	3.2.1	tool	458	27635	2008-04-15
Colt	1.2.0	SDK	593	38625	2004-09-10
Columba	1.0	tool	1190	71680	2005-09-18
Compiere	330	tool	2534	400257	2010-06-01

Full Name	Recent version	Domain	Number of Classes	LOC	Release Date
Derby	10.6.1.0	database	2995	592817	2010-05-17
Display Tag Library	1.2	diagram generator/data visualisation	131	11832	2008-12-27
drawswf	1.2.9	3D/graphics/media	319	27008	2004-06-08
DrJava	20100913-r5387	IDE	1866	62380	2010-09-13
eclipse_SDK	3.7.1	IDE	32615	2330479	2011-09-10
EMMA: a free Java code coverage tool	2.0.5312	testing	330	25806	2005-06-12
Exo Platform	v1.0.2	diagram generator/data visualisation	1225	56805	2006-05-16
Find Bugs	1.3.9	testing	1715	109096	2009-08-21
fitjava	1.1	testing	61	2240	2004-04-07
FitLibrary	20100806	testing	892	27539	2010-08-06
freecol	0.10.3	games	1225	93239	2011-09-27
FreeCS	1.3.2010040	tool	147	23012	2010-04-06
FreeMind	0.9.0	diagram generator/data visualisation	912	50198	2011-02-19
Galleon	2.3.0	3D/graphics/media	790	52653	2006-04-15
Gantt Project	2.0.9	tool	1058	47051	2009-03-31
GeoTools	2.7-M3	SDK	5593	446863	2010-09-02
Hadoop Common	1.0.0	middleware	2069	142790	2011-12-27
Heritrix	1.14.4	tool	703	61681	2010-05-10
Hibernate	4.1.0	database	3242	196995	2012-02-08
HyperSQL	2.0.0	database	535	123268	2010-06-07
HtmlUnit	2.8	testing	556	40004	2010-08-05
Informa	0.7.0-alpha2	middleware	170	9722	2007-01-06
IReport	3.7.5	diagram generator/data visualisation	3381	221490	2010-09-22

Full Name	Recent version	Domain	Number of Classes	LOC	Release Date
iText PDF	5.0.3	diagram generator/data visualisation	544	76369	2010-07-22
ivata op	0.11.3	middleware	222	23786	2005-10-10
jFin Date Math	R1.0.1	SDK	62	4807	2010-02-19
JAG	6.1	tool	255	15733	2006-05-25
James	2.2.0	tool	340	27003	2004-06-15
Java Assembling Language	0.10	tool	49	5732	2006-05-23
JasperReports	3.7.3	diagram generator/data visualisation	1844	170064	2010-07-20
javacc	5.0	parsers/generators/make	100	13772	2009-10-20
jboss	5.1.0	middleware	3612	281643	2009-05-23
jchempaint	3.0.1	SDK	1045	90831	2010-02-13
jEdit	4.3.2	tool	1128	107469	2010-05-09
Jena	2.6.3	middleware	1392	70948	2010-06-01
jext	5.0	diagram generator/data visualisation	504	26565	2004-07-07
JFreeChart	1.0.13	tool	649	98078	2009-04-20
jgraph	5.13.0.0	tool	187	22758	2009-09-28
jgraphpad	5.10.0.2	tool	431	23750	2006-11-09
JGraphT	0.8.1	tool	255	11931	2009-07-04
JGroups	2.10.0	tool	1211	96325	2010-07-12
jhotdraw	7.5.1	3D/graphics/media	1070	75958	2010-08-01
JMeter	2.5.1	testing	1003	79917	2011-09-29
jmoney	0.4.4	tool	193	8197	2003-09-29
joggplayer	1.1.4s	3D/graphics/media	130	14936	2002-04-26
jparse	0.96	parsers/generators/make	69	12559	2004-07-17
JPF	1.5.1	SDK	189	13246	2007-05-19
Java Runtime Analysis Toolkit	0.6	testing	250	14146	2003-09-14
JRE	1.6.0	programming language	10714	922958	2010-06-22

Full Name	Recent version	Domain	Number of Classes	LOC	Release Date
jrefactory	2.9.19	tool	1553	113427	2004-05-09
Java powered Ruby implementation	1.5.2	programming language	4664	160360	2010-08-20
jsXe	04_beta	tool	107	8829	2006-04-25
JSP Wiki	2.8.4	middleware	455	43326	2010-05-08
JTOpen	7.1	middleware	2054	397220	2010-08-25
JUNG	2.0.1	diagram generator/data visualisation	858	37989	2010-01-25
JUnit	4.10	testing	219	6568	2011-09-29
Log4j	1.2.16	testing	308	20637	2010-03-30
Lucene	3.5.0	tool	2309	172338	2011-11-20
Marauroa	3.8.1	games	204	13823	2010-07-25
maven	3.0	parsers/generators/make	697	54336	2010-10-04
MegaMek	0.35.18	games	2185	258957	2010-08-31
mvnForum	1.2.2-ga	tool	273	51034	2010-08-17
MyFaces Core	2.0.2	middleware	1365	119529	2010-09-25
Naked Objects	4.0.0	IDE	3002	110378	2009-08-11
NekoHTML	1.9.14	parsers/generators/make	55	6625	2010-02-02
Netbeans	6.9.1	IDE	31023	1890536	2010-08-23
OpenJMS	0.7.7-beta-1	middleware	560	33905	2007-03-14
OSCache	2.4.1	middleware	75	6198	2007-07-07
PicoContainer	2.10.2	middleware	242	9259	2010-02-25
PMD	4.2.5	testing	926	60875	2009-02-08
POI	3.6	tool	1785	143507	2009-12-15
Pooka	3.0-080505	tool	849	44474	2008-05-05
ProGuard	4.5.1	tool	658	55567	2010-07-08
Quartz	1.8.3	middleware	286	26819	2010-06-22
QuickServer	1.4.7	middleware	111	10885	2006-03-01

Full Name	Recent version	Domain	Number of Classes	LOC	Release Date
Quilt	0.6-a-5	testing	77	5683	2003-10-20
The Roller Weblogger	4.0.1	tool	587	50980	2009-01-13
RSSOwl	2.0.5	tool	1682	73230	2010-06-01
SableCC	3.2	parsers/generators/make	285	28394	2005-12-24
sandmark	3.4	tool	1088	90121	2004-08-11
Spring Framework	3.0.5	middleware	3089	160302	2010-10-20
SQuirrel SQL	3.1.2	database	169	6944	2010-06-15
Struts	2.2.1	middleware	1074	74670	2010-08-16
sunflow	0.07.2	3D/graphics/media	221	21648	2007-02-08
Tapestry	5.1.0.5	middleware	1502	53367	2009-05-06
Tomcat	7.0.2	middleware	1739	166478	2010-08-11
Trove	2.1.0	SDK	34	2196	2009-08-14
Velocity Engine	1.6.4	diagram generator/data visualisation	261	26854	2010-05-10
Web Curator Tool	1.5.2	tool	724	49933	2011-08-22
WebMail	0.7.10	tool	104	8212	2002-10-07
Weka	3.6.6	tool	2122	256454	2011-10-28
Xalan	2.7.1	parsers/generators/make	1238	189462	2007-11-27
Xerces	2.10.0	parsers/generators/make	948	129164	2010-06-18
XMOJO	5.0.0	middleware	135	17669	2003-07-17