MINING FOR PATTERNS IN

PROGRAM DEPENDENCE GRAPHS

By

IMRAN AFZAL

Bachelor of Science in Electrical Engineering

University of Engineering and Technology

Lahore, Punjab, Pakistan

1995

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 2010

MINING FOR PATTERNS IN

PROGRAM DEPENDENCE GRAPHS

Thesis Approved:

Dr. M. H. Samadzadeh

Thesis Adviser

Dr. N. Park

Dr. Xiaolin Li

Dr. Mark E. Payton

Dean of the Graduate College

PREFACE

Program graphs display programs from different perspectives. Some patterns repeat themselves in program graphs. By looking at these frequent patterns, one can make informed inferences about the underlying characteristics of programs. Searching for frequent patterns can be a challenging task due to the complexity and size of these graphs. An effort has been underway to apply data mining techniques to unsupervised discovery of these frequent patterns in graphs and then analyzing the output to deduce rules that can provide useful information about programs.

This thesis work concerned the discovery of patterns in program dependence graphs. Program dependence graphs of different versions of open source Java programs were extracted and mined for patterns. Analysis of the discovered patterns pointed out the existence of relationships between the discovered patterns and the changes made in the program code. It was found that the patterns can be grouped into at least six different classes based on the code changes they represent. These patterns can prove to be useful in program maintenance.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my graduate advisor Dr. M. H. Samadzadeh for his continued guidance, support, and encouragement throughout my thesis work. This thesis work would not have been possible without his valuable insight and direction. I would also like to thank Dr. Jonyer for introducing me to the field of data mining.

My sincere appreciation extends to Dr. Nophill Park and Dr. Xiaolin Li for their advice and for serving on my graduate committee.

Finally, I thank my parents, my wife Ayesha, and my two sons Mustafa and Ibraheem for their encouragement and patience throughout my studies.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

II. BACKGROUND 7

2.1 Knowledge Discovery from Data (KDD) 7
2.2 Association Analysis 7
2.3 Itemsets 7
2.4 Association Rules 8
2.5 Performance Evaluation 9

III. GRAPH MINING 10

3.1 Frequent Patterns 11
3.2 Program Dependence Graphs (PDG) 11
3.2.1 System Dependence Graph (SDG) 12
3.2.2 Abstract System Dependence Graph (ASDG) 12
3.3 Graph Mining Algorithms 14
3.3.1 Apriori Based Algorithms 14
3.3.1.1 AGM 15
3.3.1.2 FSG 15
3.3.1.3 FAT Miner 15
3.3.1.4 PATH 16
3.3.1.5 FFSM 16
3.3.2 Pattern Growth Algorithms 16
3.3.2.1 gSpan 16
3.3.2.2 MoFa 17
3.3.2.3 GASTON 17
3.3.3 Un-Categorized Algorithms 17
3.3.3.1 Subdue 18
3.3.3.2 Warmr 18

Chapter Page

IV. DESIGN AND IMPLEMENTATION 19

4.1 Overview 19
4.2 Abstract System Dependence Graph (ASDG) Extraction 21
4.3 Transformation 21
4.4 Pattern Generation 22
4.5 Tools 24
4.5.1 Structure 101 24
4.5.2 WEKA 24
4.5.3 Graphviz 25

V. ANALYSIS OF DISCOVERED PATTERNS 26

5.1 Overview 26
5.2 Pattern Classification 33
5.3 Class Description and Function 34
5.3.1 C1 – Newly Formed Patterns 34
5.3.2 C2 – Patterns Disappearing Right After Their Creation 35
5.3.3 C3 – Impermanent Patterns 37
5.3.4 C4 – Omnipresent Patterns 38
5.3.5 C5 – Disappearing Omnipresent Patterns 38
5.3.6 C6 – Reincarnated Patterns 42

VI. SUMMARY AND FUTURE WORK 43

REFERENCES 45

APPENDICES 49

APPENDIX A - GLOSSARY 50
APPENDIX B - PSEUDO-CODE 53
APPENDIX C - TRANSACTIONAL REPRESENTATION OF GRAPHS 55
APPENDIX D - RESULTS 57
EXPERIMENT 1 58
EXPERIMENT 2 79
EXPERIMENT 3 98
EXPERIMENT 4 103
APPENDIX E - PROGRAM CODE 145

LIST OF TABLES

Table Page

1. JDOM Package Description....4
2. Sample Table Representing PDGs in Transactional Form 21
3. Sample Summary Table of KnowledgeFlowApp.java for Pattern Discovery....23
4. Classes of Patterns 33

LIST OF FIGURES

Figure Page

1. Sample Package Level Abstract System Dependence Graph 3
2. Sample Class Level Dependence Graph 5
3. Sample Member Level Dependence Graph 5
4. A Sample Program and its ASDG 13
5. Graph G_1 and Its Corresponding Transaction T_1 20
6. A Pattern Discovered in KnowledgeFlowApp.java (Version 4804) 27
7. A Pattern Discovered in KnowledgeFlowApp.java (Version 4966) 31
8. A Class C2 Pattern Discovered in KnowledgeFlowApp.java (Version 4786) ...36
9. A Class C3 Pattern Discovered in Evaluation.java (Version 5197) 37
10. A Class C4 Pattern Discovered in KnowledgeFlowApp.java (Version 6140) ...38
11. A Class C5 Pattern Discovered in GridSearch.java (Version 6263) 39
12. A Class C1 Pattern Discovered in GridSearch.java (Version 6263) 41
13. A Class C6 Pattern Discovered in GridSearch.java (Version 4829) 42

CHAPTER I

INTRODUCTION

The past decade has seen a tremendous growth in the size and complexity of computer programs, which in turn has made it impractical to analyze program graphs through traditional techniques. This has created a need for study and unsupervised discovery of interesting structures in program graphs [Fischer and Meinl 04].

Data mining techniques have been successfully employed in different disciplines to mine graph databases. For example, different chemical structures have been mined to discover new antibiotics and graphs representing social networks have been mined to discover communities [Du et al. 07].

This thesis concerns the discovery of frequent patterns in program graphs of various open source Java programs. Java is an open source programming language which is widely used in different kinds of applications ranging from servers to clients running on mobile devices. It is fast becoming the lingua franca of software. Mining graphs of Java programs can prove to be useful for program comprehension, for aiding program maintenance, and in general for tackling the problem of software complexity.

A central issue to any data mining experiment is data. In a graph mining experiment, the first step is the extraction and processing of program graphs from source code. Unfortunately, a publicly available database of program graphs of Java programs does not seem to exist.

The first part of the research included in this thesis report addresses the issue of extracting program graphs from Java source code. A major chunk of time in data mining is typically spent on what can be generally classified as massaging the data, e.g., accounting for missing values and normalizing imprecise or inaccurate values. This is a critical and time consuming part of the knowledge discovery process. In the overall process, the time spent on careful inspection of data pays off many folds down the road [Witten and Frank 05].

For this thesis work, program graphs were extracted from two open source projects: WEKA [WEKA 08] and Derby [Derby 04]. Due to the design of Java, which is a strict object oriented language, abstract system dependence graphs (ASDG) were used in this thesis. ASDGs abstract away statement level dependencies and provide a higher level of dependency view of a program that makes them better suited for the purpose of this thesis. Dependencies exist at different levels, e.g., in a Java program there can be dependencies at package, class, and member levels. The following three sample dependence graphs, one at package level, one at class level, and one at member level, were generated for JDOM [JDOM 09], which is an open source Java based library for manipulating XML data. These graphs were generated using structure101 [Headway Software 07]. Structure101 is a tool created by Headway Software which is used to generate dependence graphs of Java class files.

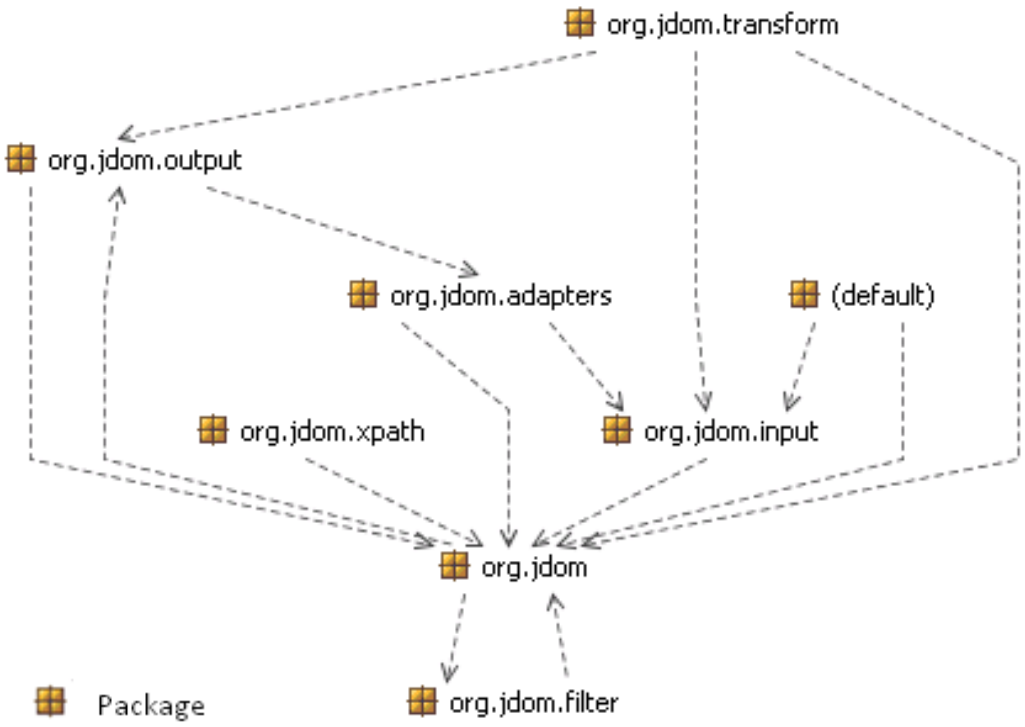


Figure 1. Sample Package Level Abstract System Dependence Graph

Figure 1 depicts a sample package level dependence graph of open source project JDOM [JDOM 09]. It is a directed graph in which the edges point from a dependent to a predicate package. For example, the code in org.jdom.transform uses code in org.jdom.input which creates a dependency of org.jdom.transform on org.jdom.input. This dependency is represented by an edge pointing from dependent package org.jdom.transform to predicate package org.jdom.input.

JDOM code is divided into eight packages [JDOM 09]. A brief description of each package appears in the table below.

Package	Description
org.jdom	This package contains classes that represent the components of an XML document, e.g., nodes and attributes.
org.jdom.adapters	This package contains classes to interface with various DOM implementations.
org.jdom.filter	This package contains classes to filter elements of a document based on type, name, and value.
org.jdom.input	This package contains classes to build JDOM documents from various inputs.
org.jdom.output	This package contains classes to output JDOM documents to various destinations such as text files.
org.jdom.transform	This package contains classes to help with transformations, based on the JAXP TrAX classes.
org.jdom.xpath	This package contains classes that provide support for XPath.
Default	This package includes classes that do not belong to any package. This package is comprised of only one class JDOMAbout.

Table 1. JDOM Package Description [JDOM 09]

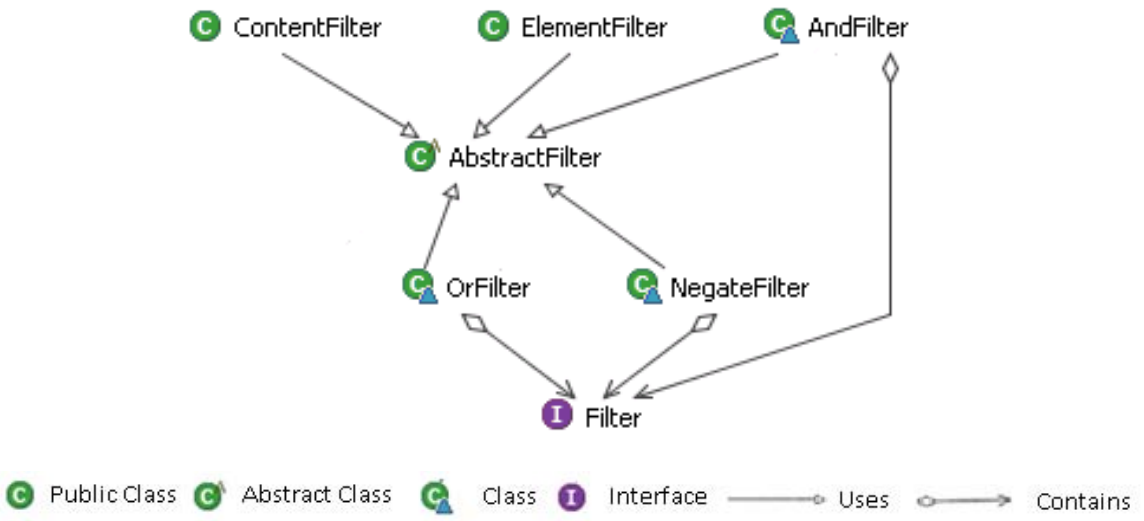


Figure 2. Sample Class Level Dependence Graph

Figure 2 depicts a sample class level dependence graph of package org.jdom.filter. This package contains six classes and one interface. These classes and interface are used to filter the elements of a document based on type, name, and value. The nodes and edges in the graph represent classes and dependencies, respectively. For example, class ElementFilter extends class AbstractFilter which creates a dependency from class ElementFilter to class AbstractFilter.

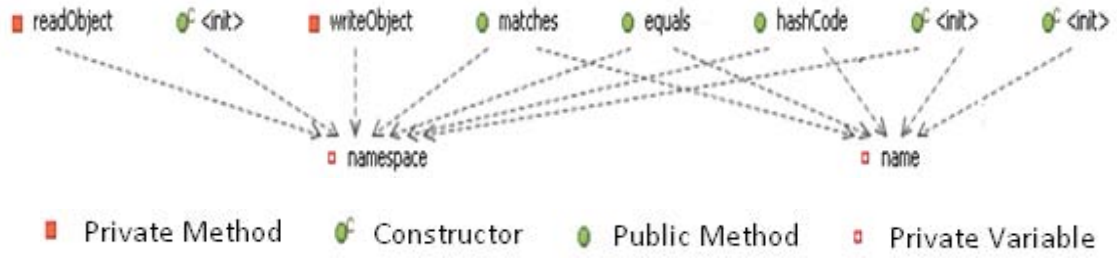


Figure 3. Sample Member Level Dependence Graph

Figure 3 depicts a sample member level dependence graph of class org.jdom.filter.ElementFilter. This class is used to filter elements in an XML document.

It contains 190 lines of code. In the above figure, each node represents a member method or a variable of class ElementFilter.java. An edge is directed from a dependent member to a predicate member. For example, method hashCode contains a reference to variable name which is represented by an edge from hashCode to name in Figure 3.

CHAPTER II

BACKGROUND

2.1 Knowledge Discovery from Data (KDD)

KDD is the process of extracting knowledge from data. It consists of data cleaning, data integration, data selection, data mining, pattern evaluation, and knowledge representation [Han and Kamber 06].

2.2 Association Analysis

Market basket is the collection of items purchased by a customer in a trip to a store. One of the initial applications of association analysis was in market basket analysis in studying the relationships among items purchased by customers in retail stores. Simply put, it is the methodology used to discover the rules that represent the probability of appearance of different items together in an itemset [Tan et al. 06].

2.3 Itemsets

Let the set of all transactions be $T = \{t_1, t_2, t_3,..., t_N\}$ and the set of all items be $I = \{i_1, i_2, i_3,..., i_M\}$, $t_j \subseteq I$ for $1 \leq j \leq N$ [Tan et al. 06]. A collection of zero or more items is called an itemset, and a collection of k-items is called a k-itemset, e.g., $\{i_1, i_3\}$ is a 2-itemset. In a market basket, an itemset can consist of the items purchased by a customer during a visit to the retail store [Tan et al. 06].

2.4 Association Rules

Tan, Steinbech, and Kumar define association rules as follows [Tan et al. 06].

An association rule is an implication expression of the form $X \rightarrow Y$, where X and Y are disjoint itemsets, i.e., $X \cap Y = \emptyset$. The strength of an association rule can be measured in terms of its support and confidence, as defined below.

If we represent *support count* by σ, then the support count of an itemset X can be expressed as:

$$\sigma(X) = |\{ t_N \mid X \subseteq t_N,\ t_N \in T\}|$$

So, the support count of an itemset is the number of transactions in T that contain all of its items.

Support is the measure of occurrence of a rule in a given set of transactions T, where N is the total number of elements in T. Specifically, for an association rule $X \rightarrow Y$, the support s can be defined as [Tan et al. 06]:

$$s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N}$$

Confidence is the measure of frequency of occurrence of all items in itemset Y, whenever all items of X occur in a transaction t_n. The confidence c of an association rule $X \rightarrow Y$ can be defined as [Tan et al. 06]:

$$c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}$$

Support is used to separate rules that might occur only by chance from the ones that actually represent some underlying phenomenon. The minimum support of a rule is determined through feedback from domain knowledge experts in several iterations of rule generation.

Confidence is used as a measure of how reliable a rule is. As stated by Tan et al., the confidence of the association rule X→Y "represents the conditional probability of Y occurring every time X does" [Tan et al. 06].

The frequent itemsets are the itemsets whose frequency of occurrence in the data set is greater than a minimum support specified by a domain expert.

2.5 Performance Evaluation

Evaluating the results of data mining can be tricky. One notable method, which has started to emerge as the standard in many data mining experiments, is cross-validation [Witten and Frank 05]. In cross-validation, the data is divided into n equal parts. One part is put away as the testing data set and the rest of the data is used to generate rules. Then the generated rules are tested against the test data that was initially held out from the training dataset. This process is repeated n times so that every part is used as the test data. This method of evaluating the test results is called n-fold cross-validation [Witten and Frank 05].

CHAPTER III

GRAPH MINING

In the real world, not everything can be represented by simple transactions. A large number of applications require more complex data structures [Han and Kamber 06] and graphs can be used to model these structures.

Graphs have become increasingly popular in modeling complicated structures such as circuits, images, chemical compounds, protein structures, biological networks, social networks, the Web, workflows, and XML documents [Han and Kamber 06].

These graph models are well studied and a lot of effort has been put into their analysis. By using data mining techniques, the hidden patterns in the data collected as graphs can be discovered and analyzed.

In the past few years, much attention has been paid to mining graphs and the techniques learned in mining transactional data have been modified to mine graph data [Han and Kamber 06].

3.1 Frequent Patterns

Frequent patterns are the graph counterpart of transactional itemsets (for a definition, see Section 2.3) : a set of nodes and edges appearing multiple times in a graph dataset with their support being greater than or equal to a minimum threshold specified by the user.

Similar to frequent itemsets, frequent patterns represent some underlying association rule hidden in the structure of the data.

3.2 Program Dependence Graphs (PDG)

A PDG is a directed graph whose nodes represent the statements of a program, and each edge connecting two nodes represents the dependence of one statement on another [Ferrante et al. 87].

A program dependence graph represents two types of dependence: data dependence and control dependence, as explained below.

a. Data Dependence: If the order of execution of two statements s1 and s2 is inverted in a program and this affects the values of their variables, then the two statements have a data dependency on each other.

b. Control Dependence: If the execution of one statement depends on the values of variables of the other (predicate) statement, then a control dependency exists between the two statements.

3.2.1 System Dependence Graph (SDG)

The program dependence graphs represent dependencies at the statement level. These graphs were originally used in compiler optimization but were later adopted by researchers to study other software engineering problems such as program slicing and maintenance. Horwitz et al. [Horwitz et al. 88] extended PDGs by including procedures and procedure calls. They called these graphs System Dependence Graphs (SDG) and used them in program slicing.

3.2.2 Abstract System Dependence Graphs (ASDG)

Chen and Rajlich proposed abstract system dependence graphs to study the feature location in program code [Chen and Rajlich 00]. Feature location is the process of identifying which components of a software system will implement a certain feature. For example, a feature request by a product team can be the addition of a new parameter to the routing logic of a transaction processing system, the feature location for such a request will be the identification of parts of the system where this feature will be implemented [Chen and Rajlich 00]. They argued that by abstracting away the statement level dependence, the resulting higher level of abstraction is more useful in feature location and statement level dependency is not required. What follows is a sample C program and its corresponding ASDG.

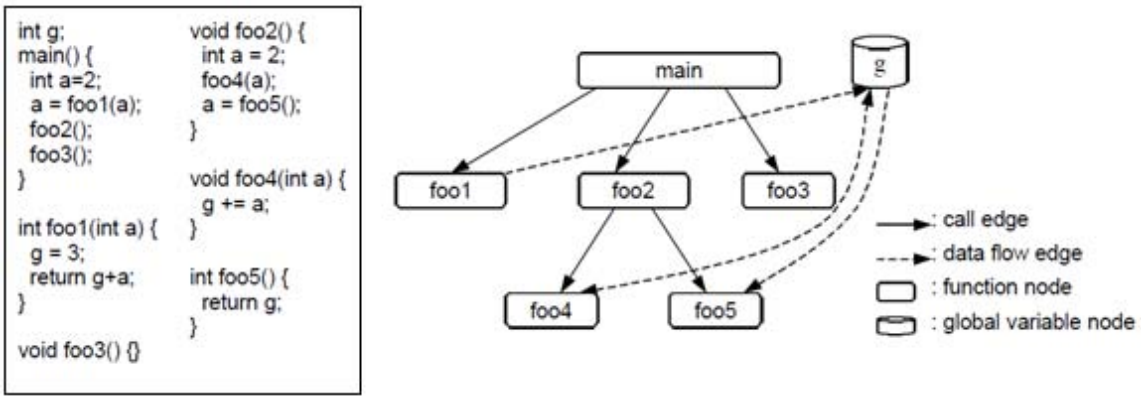


Figure 4. A Sample Program and Its ASDG [Chen and Rajlich 00]

In the above figure, the nodes of the ASDG represent functions or global variables in the code. A continuous edge represents a function call and a dotted line represents a data flow edge [Chen and Rajlich 00].

Software systems are becoming increasingly large and the amount of information provided by PDGs (Program dependence Graphs) can be overwhelming. Therefore, ASDGs (Abstract System Dependence Graphs) were considered more suitable for modeling the dependencies of Java programs, and hence they were used in this study. The tool used for extracting graphs from programs is called Structure101 [Headway Software 07] (for details, please refer to Section 4.5.1). It generates added annotation for an edge including the number of edges and type of dependency, e.g., “uses”, “implements”, and “extends”. However, for the purpose of this research, the types of the dependencies were ignored since they did not contribute to mining for patterns.

3.3 Graph Mining Algorithms

Graph mining algorithms can be divided into two groups: Apriori Based Algorithms and Pattern Growth Algorithms [Han and Kamber.06]. These two groups are briefly explained in the following subsections.

3.3.1Apriori Based Algorithms

Apriori based algorithms extend the Apriori algorithm [Agrawal and Srikant 94] for discovering association rules in transactional data.

Agrawal and Srikant made an important discovery that became the foundation of all Apriori Based algorithms [Agrawal and Srikant 94]. They called this property *Apriori* and described it as follows: A k-itemset is frequent only if all of its sub-itemsets are frequent.

Utilizing this property, they proposed an algorithm that discovered all the 1-itemsets and used these 1-itemsets to create all 2-itemsets, eliminating the itemsets whose support and confidence were less than the established threshold. These steps were repeated till all the frequent k-itemsets were discovered [Agrawal and Srikant 94].

In a similar manner, the discovery of frequent graphs uses frequent patterns of size 1 and then builds on them, where the size of a pattern is defined by the number of its nodes. After each repetition, the size of frequent patterns is incremented by one and patterns with support less than the threshold are eliminated [Tan et al. 06]. The generation of patterns at the beginning of each repetition is called *candidate generation* .

The following five subsections briefly describe different Apriori-Based algorithms.

3.3.1.1 AGM

AGM (**A**priori Based **G**raph **M**ining Algorithm) discovers the association rules hidden in the frequent patterns in a graph. It uses an adjacency matrix to model the graph and then mines it for frequent patterns by using a typical extension of the algorithm proposed by Agrawal [Agrawal et al. 93] [Inokuchi et al. 00].

3.3.1.2 FSG

FSG (**F**requent **S**ub**g**raph Discovery) discovers frequent patterns in a graph. It has a linear execution time. It uses adjacency lists to store input graphs, intermediate candidates, and frequent subgraphs. This representation improves the algorithm's efficiency when input graphs are sparse. The candidate generation process has been improved by using different optimization techniques [Kuramochi and Karypis 04].

3.3.1.3 FAT Miner

The FAT (**F**requent **A**ttribute **T**ree) Miner algorithm is used to find frequent patterns in attribute trees. Knijf defines an attribute tree as “a labeled rooted ordered tree, with a non-empty attribute node” [Knijf 07]. In certain real life applications such as XML databases and HTML pages, a lot of information can lay hidden in the attributes of nodes, unless these attributes are taken into account while discovering frequent subgraphs. FAT Miner does not ignore node attributes and mines nodes as well as their attributes to generate frequent subgraphs [Knijf 07].

3.3.1.4 PATH

PATH (**Path**-join edge-disjoint) is another variation of the Apriori Graph algorithm. It considers only edge-disjoint paths in candidate generation and then joins the candidates to create bigger patterns. This method also uses a novel definition of support to mine graph patterns [Vanetik et al. 02].

3.3.1.5 FFSM

FFSM (**F**ast **F**requent **S**ubgraph **M**ining) reduces the number of candidate patterns generated during the candidate generation process. It uses a novel framework that restricts the generation of useless candidates [Huan et al. 03].

3.3.2 Pattern-Growth Algorithms

A pattern-growth mining algorithm starts with an already discovered frequent graph 'g'. It keeps on extending g by appending one edge at a time in all possible positions. If the resultant graph does not exist in the dataset or its support is less than the threshold, it is discarded. The edge is added to the frequent candidates and the whole process is repeated recursively until all of the frequent patterns containing g are discovered [Han and Kamber 06].

3.3.2.1 gSpan

gSpan (**G**raph Based **S**ubstructure **Pa**ttern Mi**n**ing) is the first algorithm that uses DFS (Depth First Search) to discover frequent patterns. It also safeguards against the possible rediscovering of the same graph multiple times. gSpan employs a new lexicographic

ordering technique for DFS trees in graphs and using this order performs depth first search to mine frequent subgraphs [Yan and Han 02].

3.3.2.2 MoFa

MoFa (**M**ining **Mo**lecular **Fra**gments) was developed to discover frequent molecular fragments in different chemical compounds. This algorithm employs an inductive technique to prune the search tree by eliminating the molecular substructures that violate the natural laws of chemistry. Compared to the brute force method, when provided with a set of base rules, the efficiency of the algorithm is increased substantially [Borgelt and Berthold 02].

3.3.2.3 GASTON

GASTON (**G**raph **S**equence **T**ree extracti**on**) divides the frequent pattern discovery process into phases to speed up frequent pattern mining. It is based on the assumption that frequent patterns discovered in any graph dataset are mostly free trees, where a free tree is an undirected graph that is both connected and acyclic. It first searches for frequent free trees in the dataset and then, using these discovered frequent free trees, it discovers other frequent patterns by employing the pattern growth technique [Nijssen and Kok 04].

3.3.3 Un-Categorized Algorithms

Besides the above Apriori Based and Pattern Growth algorithms, there are two noteworthy algorithms that do not fall into any category.

3.3.3.1 Subdue

This algorithm was one of the initial attempts to discover substructures. It removes previously discovered patterns from the graph dataset and mines the graphs again. It keeps on repeating this process till it is unable to find any more subgraphs [Holder et al. 94].

3.3.3.2 Warmr

Warmr (a data mining tool for chemical data) was the first inductive logic programming algorithm that was used for data mining [King et al. 01]. The value of Warmr lies in its ability to find all frequent substructures in the database and the high accuracy of the rules discovered by it. As reported by King et al., the results of the application of this algorithm to a well-studied database of chemical compounds, put a lower bound on the complexity of the relationship between chemical structure and carcinogenicity.

CHAPTER IV

DESIGN AND IMPLEMENTATION

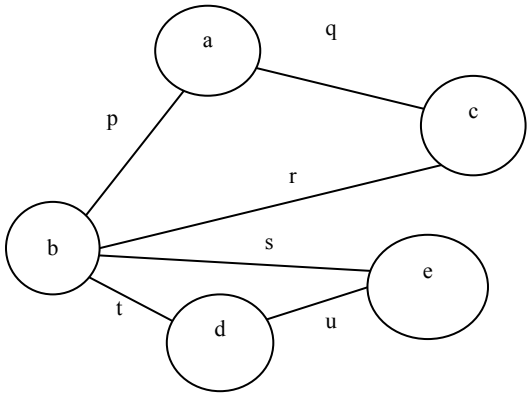
This chapter starts with an overview of the approach taken to discover frequent patterns from programs. Then the entire process is described by using a sample Java class and mining it for patterns.

4.1 Overview

The earliest techniques to discover association rules were developed by Agrawal et al. in the first half of the 90s [Agrawal et al. 93]. These techniques were employed to mine the data gathered by retailers to study the purchasing behavior of their customers. Later on, having seen the benefits of the knowledge discovered, new algorithms were invented to mine more complex data structures such as graphs.

This thesis work concerned the use of techniques developed for mining transactional data to mine program graphs by first transforming these graphs to a representation which mimics transactions.

This approach was initially proposed by Inokuchi [Inokuchi et al. 99]. In this approach, for each 3-tuple of an edge label *l(e)* and its two corresponding vertex *l(v)* labels, there exists an item I_i in Transaction T_i . So, for example, graph G_1 depicted below can be represented by a corresponding transaction T_1.



T_1 = {(a,b,p), (a,c,q), (c,b,r), (b,e,s), (b,d,t), (e,d,u)}

Figure 5. Graph G_1 and Its Corresponding Transaction T_1

The size of transaction T will be equal to the number of edges in the corresponding graph G. This approach can only work if every 3-tuple is unique. The program graphs used for this study fulfill this requirement (for more details, please see Appendix C).

Once these graphs are modeled accurately, the existing techniques of transactional data mining can be employed to find the itemsets that represent frequent graph patterns.

A Java class (KnowledgeFlowApp.java) was used as an example. This class belongs to the WEKA (Waikato Environment for Knowledge Analysis) project which is an open-source data mining platform. The class had 3169 lines of code in its latest version and it was changed 19 times as of 2008 [WEKA 08].

4.2 ASDG (Abstract System Dependence Graph) Extraction

All 19 versions (4698, 4762, 4786, 4796, 4804, 4806, 4829, 4899, 4953, 4966, 4984, 5134, 5226, 5244, 5396, 5611, 6013, 6015, 6140) of the Java class KnowledgeFlowApp.java were downloaded from their subversion source code repository. Using structure101 (for a description, see Section 4.5.1), an abstract system dependence graph was extracted for each version. We shall refer to each ASDG (Abstract System Dependence Graph) by the notation ASDG (N), where N is the subversion revision number associated with a specific version of the class.

4.3 Transformation of Graphs to Transactions

Each ASDG (Abstract System Dependence Graph) was transformed from a graph to a transaction by using the method mentioned in Section. 4.1 Overview.

After the transformation, all ASDGs with nodes A_1,A_2,A_3, …, A_n appeared as rows in a table with their edge label *l(e)* as the columns.

VERSION ID	REVISION NUMBER	$A_1 \rightarrow A_2$	$A_2 \rightarrow A_3$	$A_3 \rightarrow A_4$	…..	$A_{n-1} \rightarrow A_n$
4698	0	1	0	0	0	1
……	……	……	……	……	……	……
6140	18	1	1	1	1	1

Table 2. Sample Table Representing PDGs in Transactional Form

Table 2 shows the transactional representation of PDGs. A value of 1 represents the presence, and 0 the absence of a particular edge in the graph corresponding to a specific version of a Java class.

4.4 Pattern Generation

Pattern generation took place every time a program was modified and committed to the code repository. Pattern mining algorithms were used to mine for patterns incrementally by starting from the oldest version and working to the newest version. At each step, only the current and previous versions were considered for mining and all the subsequent versions were ignored. This stepwise approach allowed us to study the changes in the discovered patterns from one version to another, e.g., patterns discovered in version n of a class were compared with the patterns discovered in version n-1 of the same class.

To speed up the analysis, a summary table was maintained which was updated with the support count (for the definition and calculation, see Section 2.4) and value of each edge for the respective version. This way, instead of calculating the support count of each edge from the scratch for every new version of the class, the support count of the previous version was used. For example, the support count of an edge A→B for version n, was obtained by adding one to its support count in version n-1.

VERSION ID	REVESION NUMBER	EDGE	SUPPORT COUNT	VALUE
4698	0	$A_1 \rightarrow A_2$	1	1
4698	0	$A_2 \rightarrow A_3$	1	1
4698	0	$A_3 \rightarrow A_4$	1	1
……	……	……	……	……
4698	0	$A_{n-1} \rightarrow A_n$	1	1
4762	1	$A_1 \rightarrow A_2$	2	1
4762	1	$A_2 \rightarrow A_3$	2	1
4762	1	$A_3 \rightarrow A_4$	2	1
……	……	……	……	……
4762	1	$A_{n-1} \rightarrow A_n$	1	0
……	……	……	……	……
6140	18	$A_1 \rightarrow A_2$	16	1
6140	18	$A_2 \rightarrow A_3$	19	1
6140	18	$A_3 \rightarrow A_4$	19	1
……	……	……	……	……
6140	18	$A_{n-1} \rightarrow A_n$	18	1

Table 3. Sample Summary Table of KnowledgeFlowApp.java for Pattern Discovery

Table 3 shows a sample summary table of Java class KnowledgeFlowApp. Each row of the table shows the support count and value of edge label *l(e)* for each version of the program. Appendix B lists the pseudo-code of the algorithms used to discover the patterns.

4.5 Tools

The following subsections contain a brief description of the tools that were used to generate and analyze program graphs from source code. There are mainly three tools that were used to gather and analyze data: Structure101, WEKA, and Graphviz.

4.5.1 Structure 101

Structure101 is a product of Headway Software [Headway Software 07]. Its focus is allowing the teams developing applications in Java to pin-point the dependencies that might exist among different pieces of code. One feature of this software, which was of interest to this research, was its ability to generate ASDGs (Abstract System Dependence Graphs) of programs at different levels (package, class, etc.) and to export them to xml files.

As part of this thesis work, Java programs were written to transform graphs generated by Structure 101 to ARFF (Attribute-Relation File Format) files that can be loaded to WEKA, the data mining tool, and to mySQL database.

4.5.2 WEKA

WEKA (Waikato Environment for Knowledge Analysis) is a data mining platform [WEKA 08]. It was developed at the University of Waikato, New Zealand. It is written in Java and is available under GNU GPL (General Public License). It has tools for each phase of the knowledge discovery process. It supports data pre-processing, data visualization, and result evaluation. It also contains a library of association, classification, and clustering algorithms [Witten and Frank 05].

4.5.3 Graphviz

Graphviz (Graph Visualization) is an open-source collection of graph drawing tools [Ellson et al. 02]. It is distributed as an add-on package for SUSE Linux, Debian, Mandrake, SourceForge, and Open BSD. The input can be provided in the form of XML or text files, and graphs can be generated and exported in different formats, e.g., jpeg, gif, and png.

CHAPTER V

ANALYSIS OF DISCOVERED PATTERNS

In the previous chapter we discussed different steps involved in discovering patterns in the abstract system dependence graphs (ASDG) of Java classes. This chapter contains the results and analysis of the discovered patterns.

While studying the discovered patterns, it was observed that they can be grouped into classes to better represent the underlying program. The patterns were classified into six groups, C1 through C6, based on the support count and value of the edges in the graph. These six groups are: newly formed patterns (C1), patterns disappearing right after their creation (C2), impermanent patterns (C3), omnipresent patterns (C4), disappearing omnipresent patterns (C5) and reincarnated patterns (C6).

5.1 Overview

Each pattern maps to some characteristic related with the program. This can be better explained by taking a few examples from the patterns generated from a sample class.

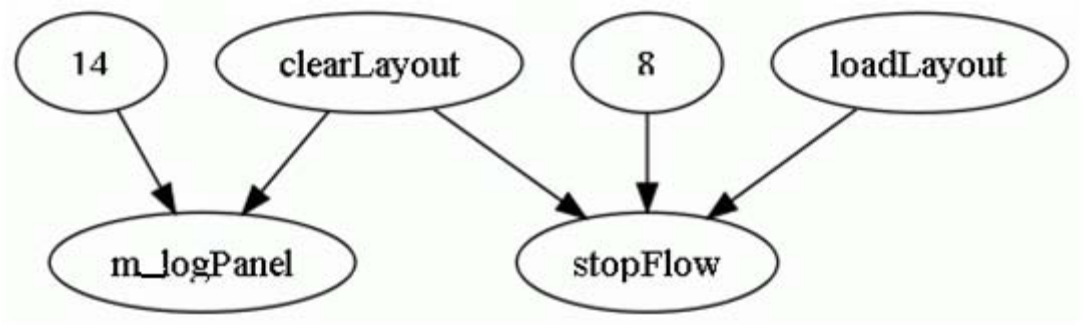


Figure 6. A Pattern Discovered in KnowledgeFlowApp.java (Version 4804)

The above figure shows a pattern that was discovered in version 4804 of the class KnowledgeFlowApp.java. This pattern belongs to the C1 class of newly-formed dependencies in the program. An examination of the pattern reveals that dependencies were created from various nodes to stopFlow. A close scrutiny of the previous versions of the class and comparing them with the new version indicates that the code in the anonymous class represented by node 8 in the pattern had a code segment which was removed and moved to a new function called stopFlow. This explains why there was a new node in the pattern and also that there were multiple references to that node from already existing nodes.

The code in version 4796 and what it got replaced by in version 4804 are given below.

The code in the following figure is part of method setUpToolbars. In this code fragment, an action named listener is defined. When this listener action is called, it stops all the running bean instances.

Code in version 4796

```
m_stopB.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
 m_logPanel.statusMessage("[KnowledgeFlow]|Attempting to stop all components...");

Vector components = BeanInstance.getBeanInstances();

for (int i = 0; i < components.size(); i++) {
      Object temp = ((BeanInstance)    components.elementAt(i)).getBean();

          if (temp instanceof BeanCommon) {
            ((BeanCommon) temp).stop();
          }
        }
        m_logPanel.statusMessage("[KnowledgeFlow]|OK.");
      }
    });
```

The code segment in the following figure shows the modified listener action. In this version of the class, the code dealing with stopping of the beans is replaced by the call to a new method stopFlow. This method now contains the code used to stop all the objects of type BeanInstance.

Code in version 4804

```
m_stopB.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         m_logPanel.statusMessage("[KnowledgeFlow]|Attempting to stop all
components...");
         stopFlow();
         m_logPanel.statusMessage("[KnowledgeFlow]|OK.");
       }
     });

private void stopFlow() {
   Vector components = BeanInstance.getBeanInstances();

   for (int i = 0; i < components.size(); i++) {
     Object temp = ((BeanInstance) components.elementAt(i)).getBean();

     if (temp instanceof BeanCommon) {
       ((BeanCommon) temp).stop();
     }
   }
 }
```

The following code segments represent the newly formed edge from clearLayout to stopFlow. In version 4796, method clearLayout did not have the code to stop the objects of type BeanInstance. In version 4804 a call to method stopFlow was added in method clearLayout.

Code in version 4796

```
public void clearLayout() {
   BeanInstance.reset(m_beanLayout);
   BeanConnection.reset();
   m_beanLayout.revalidate();
   m_beanLayout.repaint();
 }
```

Code in version 4804

```
public void clearLayout() {
   stopFlow(); // try and stop any running components
   BeanInstance.reset(m_beanLayout);
   BeanConnection.reset();
   m_beanLayout.revalidate();
   m_beanLayout.repaint();
   m_logPanel.clearStatus();
   m_logPanel.statusMessage("[KnowledgeFlow]|Welcome to the Weka Knowledge
Flow");
  }
```

The three newly formed edges 8→stopFlow, clearLayout→stopFlow, and loadLayout→stopFlow appear to point towards a scenario where a programmer wrote the code to accomplish a certain task in one method and then realized that the same task needed to be done in other methods in the program. This resulted in the creation of a new method stopFlow and three calls to this method.

The rest of the two edges 14→m_LogPanel and clearLayout→m_LogPanel represent statements to display messages in m_LogPanel which is a Java swing GUI control JScrollPane.

The following code shows the statements which are represented by the edges 14→m_LogPanel and clearLayout→m_LogPanel in the pattern in Figure 6.

Code in version 4804

```
deleteItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          BeanConnection.removeConnections(bi);
          bi.removeBean(m_beanLayout);
          if (bc instanceof BeanCommon) {
            String key = ((BeanCommon)bc).getCustomName()
              + "$" + bc.hashCode();
            m_logPanel.statusMessage(key + "|remove");
          }
          revalidate();
          notifyIsDirty();
        } });

public void clearLayout() {
   stopFlow(); // try and stop any running components
   BeanInstance.reset(m_beanLayout);
   BeanConnection.reset();
   m_beanLayout.revalidate();
   m_beanLayout.repaint();
   m_logPanel.clearStatus();
   m_logPanel.statusMessage("[KnowledgeFlow]|Welcome to the Weka
Knowledge Flow");
  }
```

The figure below represents a pattern that was discovered in version 4966 of the sample class KnowledgeFlowApp.java. This pattern represents three newly formed dependencies.

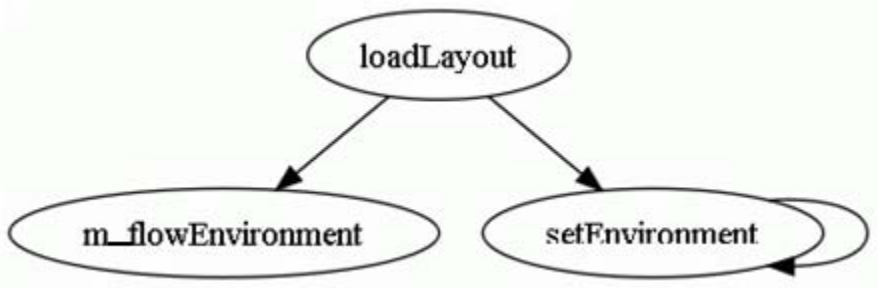


Figure 7. A Pattern Discovered in KnowledgeFlowApp.java (Version 4966)

In Figure 7, node m_flowEnvironment represents a member variable of type Environment, and node setEnvironment represents a member method of the class KnowledgeFlowApp.java. The signature of method loadLayout was changed to take an input parameter of type Environment. This change is represented by edges loadLayout→m_flowEnvironment and loadLayout→setEnvironment.

Code in version 4966

```
/** Environment variables for the current flow */
  protected Environment m_flowEnvironment;

  /**
   * Set the environment variables to use. NOTE: loading a new layout
   * resets back to the default set of variables
   *
   * @param env
   */
  public void setEnvironment(Environment env) {
    m_flowEnvironment = env;
    // pass m_flowEnvironment to all components
    // that implement EnvironmentHandler
    Vector beans = BeanInstance.getBeanInstances();
    for (int i = 0; i < beans.size(); i++) {
      Object temp = ((BeanInstance) beans.elementAt(i)).getBean();
      if (temp instanceof EnvironmentHandler) {
        ((EnvironmentHandler) temp).setEnvironment(m_flowEnvironment);
      }
    }
  }
```

In the above code segment, m_flowEnvironment is a new member variable introduced in version 4966 and a new function setEnvironment is used to initialize it. An interesting point to note is that setEnvironment node is referencing itself. A pattern like this represents a recursive call.

5.2 Pattern Classification

If we represent support count by σ (for a definition of and examples for support count, please see Section 2.4), the item by *i*, and the version index as *n*, where version index start from 0 and increments by 1 every time a new version of class is committed in the repository, we can list the different types of patterns, i.e., C1, C2, C3, C4, C5, and C6, in a table as follows:

Class	**σ(i_n)**	**σ(i_{n-1})**	**value(i_n)**	**value(i_{n-1})**
C1	1	NA	1	NA
C2	1	NA	0	1
C3	≥ 1	NA	0	1
C4	n+1	NA	NA	NA
C5	n	N	NA	NA
C6	>1	NA	1	0

Table 4. Classes of Patterns

In this table, σ (i_n) is the support count of an item *i* in version index *n* of a class, e.g., the support count of an item $i = A_5 \rightarrow A_6$ in the third revision will be represented by

$\sigma([A_5 \rightarrow A_6]_2)$. An edge in a graph is represented by an item in the transaction, for reference, please see Section 4.1 Overview.

5.3 Class Description and Function

This classification of patterns into six groups was made as a result of observation and experience, and no claim is made as to the completeness or consistency of this classification. The existence of these patterns was manifested as a result of the generation of patterns of different levels of support and confidence over and over again. After doing and redoing the same experiments with different parameters, it became apparent that the patterns with minimum and maximum support count tended to bind closely to the changes made in the program code. Beyond the grouping of patterns into six classes, further classification of the patterns is possible, and some such categorizations are suggested as future work towards the end of this thesis report.

The following six subsections briefly describe the classes, i.e., newly formed patterns, patterns disappearing right after their creation, impermanent patterns, omnipresent patterns, disappearing omnipresent patterns, and reincarnated patterns.

5.3.1 C1 – Newly Formed Patterns

The patterns of this type represent newly-created dependencies. Both of the patterns discussed above in Section 5.1 belong to this class.

5.3.2 C2 – Patterns Disappearing Right After Their Creation

These patterns represent pieces of code which were incorrect and were removed based on changes in the design. For example, in our sample class (see Section 4.1) a lot of such patterns were formed in the second revision of the class in version 4786. This can be the result of a change in design approach by the author early in the development phase. An example of such a pattern appears below.

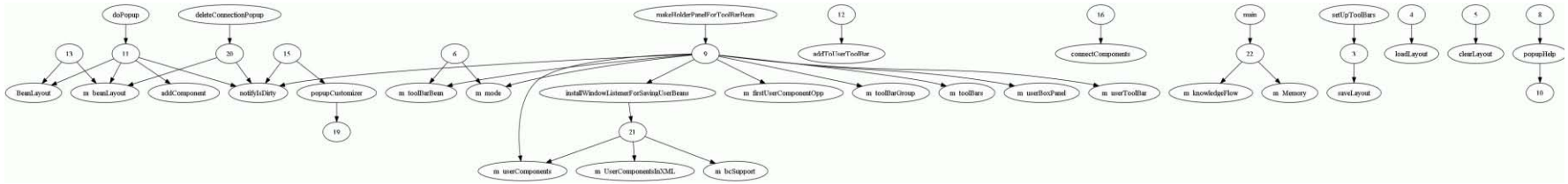


Figure 8. A Class C2 Pattern Discovered in KnowledgeFlowApp.java (Version 4786)

5.3.3 C3 – Impermanent Patterns

This class of patterns is similar to C2 because it also contains items that disappeared in the current version. The basic difference between the two patterns is that C2 includes only items that appeared for the first time in the previous version and disappeared in the current version, whereas C3 includes any disappeared items irrespective of which version they appeared in. The C3 group was created to capture the dependencies that disappeared after having been there for one or more versions. Patterns in this class could represent code written by one developer who is constantly wrong and is being asked to change his/her code, or it could represent some functionality that is constantly being second-guessed. Such patterns might also appear in projects where the customer requirements were either not completely understood or were constantly changing. In any case, such patterns represent a trouble spot that requires involvement by the senior team members. The following example is taken from the patterns discovered in version 5197 of Evalauation.java. Evaluation.java also belongs to WEKA [WEKA 08]. This class is revised 17 times and its latest version contains 3643 lines of code.

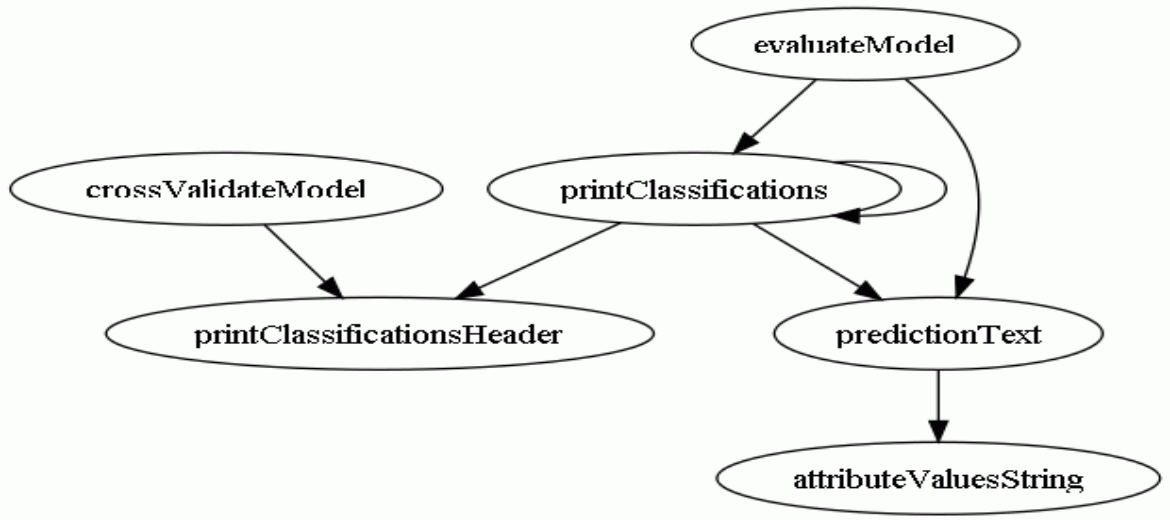


Figure 9. A Class C3 Pattern Discovered in Evaluation.java (Version 5197)

In this pattern we see all the dependencies that were removed from the program in version 5197.

5.3.4 C4 - Omnipresent Patterns

These patterns are always present, starting from the inception of the program. These patterns can represent code segments that describe the basic foundation of the code. These patterns represent dependencies that have always been present in the code. An example of omnipresent patterns from version 6140 in KnowledgeFlowApp.java appears below.

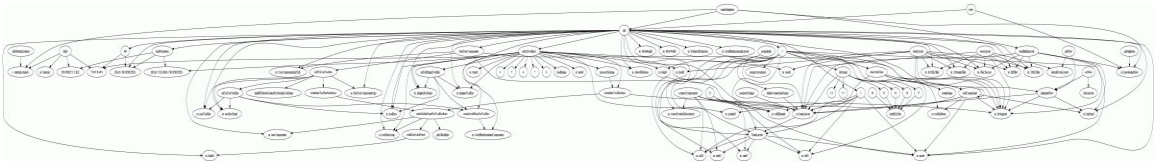


Figure 10. A Class C4 Pattern Discovered in KnowledgeFlowApp.java (Version 6140)

Admittedly, this figure is not readable, however it is indicative in that it shows that a comparatively large part of the code stays virtually intact. In this case out of 296 edges in the first revision (version 4762) of KnowledgeFlowApp.java 196 edges remained unchanged.

5.3.5 C5 – Disappearing Omnipresent Patterns

A pattern in this class represents a dependency which, after being present in every previous version, for some reason disappeared. Patterns with very high support count that appear very early in the life of a project, could disappear after a few versions. For example, in KnowledgeFlowApp such patterns were not encountered after revision 2. A

pattern must be included in this category only after careful consideration. Unexpected appearance of patterns belonging to this class can be used to alert managers and leads who can then talk to the developer to find out why this change was made and if there are any unforeseen consequences. It has also been observed that these patterns are generally accompanied by some new patterns of type C1 in the same or the next revision to replace the functionality that was provided by this dependency. Subsequent example from version 6263 from GridSearch.java appears below. This class also belongs to WEKA [WEKA 08]. It was revised 5 times. Version 6263 is the latest version of this class.

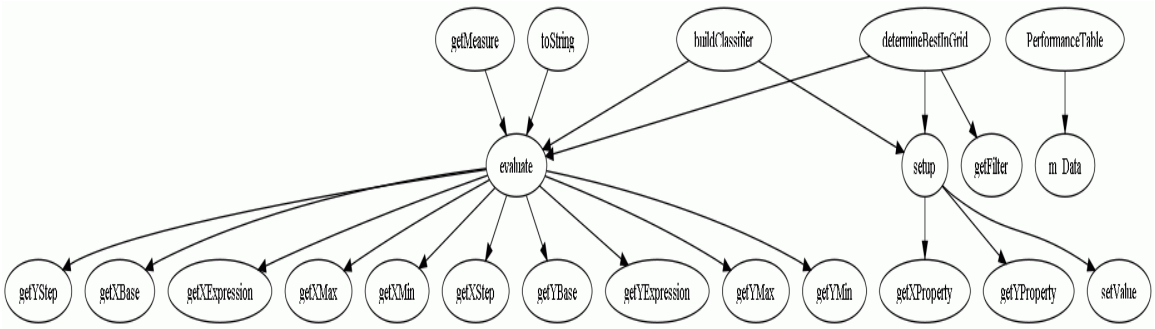


Figure 11. A Class C5 Pattern Discovered in GridSearch.java (Version 6263)

The above pattern represents dependencies that were present in all previous version of the class but disappeared in version 6263. The code was modified to take advantage of multi-core machines and during this change the dependencies in the above pattern were removed.

In the same version 6263 patterns belonging to C1 appeared which do appear to be replacing some of the dependencies that disappeared. For example, edges evaluate→getXStep, evaluate→getYBase, evaluate→getYExpression, evaluate→getYMax, evaluate→getYMin, evaluate→getYProperty in Figure 11 seem to

have been replaced by edges SetupGenerator→getXStep, SetupGenerator →getYBase, SetupGenerator→getYExpression, SetupGenerator→getYMax, SetupGenerator→getYMin, SetupGenerator →getYProperty in Figure 12, which contains patterns of group C1.

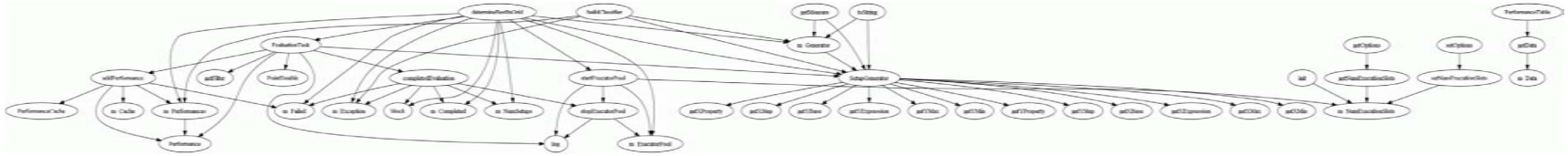


Figure 12. A Class C1 Pattern Discovered in GridSearch.java (Version 6263)

5.3.6 C6 - Reincarnated Patterns

Patterns in this class point to appearance, disappearance, and then re-appearance of dependencies. They are meant to represent interesting changes and can be used to ask the following questions about the code segments that they represent: *What function did they perform? Why did they disappear? What brought them back?* These patterns almost always represent contentious code and should be paid close attention to. In the experiments for this thesis, none of these types of patterns were found but, edge clearLayout->m_beanLayout of KnowledgeFlowApp.java class was manipulated to mimic conditions which caused it to appear as a reincarnated pattern. This edge existed in three consecutive versions 4804, 4806, and 4829. It was removed from version 4806 and it appeared in group C6 of version 4829.

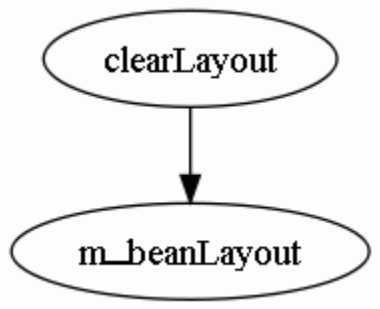


Figure 13. A Class C6 Pattern Discovered in KnowledgeFlowApp.java (Version 4829)

CHAPTER VI

SUMMARY AND FUTURE WORK

Chapter I discusses the need for unsupervised discovery of patterns in program graphs. Chapter II contains information about knowledge discovery from data (KDD). Chapter III describes program graphs and various graph mining algorithms. Chapter IV is about program graph extraction from Java source code and discovery of patterns. Chapter V includes the analysis of patterns discovered in the program graphs.

In this thesis work the potential that lies in the use of program graphs for software maintenance and comprehension was explored. Graph patterns can prove to be an invaluable tool for architects, designers, managers, and team leads in effectively monitoring the changes in the code.

The possibility of using the discovered patterns to control changes in code segments was discussed in Chapter V. For example, if it is not desirable to change a dependency, then one needs to make sure that it does not appear as a removed dependency in group C5. Similarly, if it is desirable for certain objects in a project not to have any dependency on each other, then one would need to do so by reviewing the C1 patterns. Any changes to the contrary (i.e., creation of a new dependency among the objects) will be immediately

reflected in the patterns generated after the code is committed to the repository by the development team.

Work needs to be done towards integrating the pattern discovery process with the nightly builds. The patterns can be generated as part of the daily run of the automated regression suite and reviewed in conjunction with the test results.

In this study, the focus was on abstract system dependence graphs (ASDG). There are other program graphs, e.g., control dependence graphs (CDG), control flow graphs (CFG), and data flow graphs (DFG) that can be used and analyzed in a similar fashion.

REFERENCES

[Agrawal and Srikant 94] Rakesh Agrawal and Ramakrishnan Srikant, "Fast Algorithms for Mining Association Rules", *Proceedings of the International Conference on Very Large Databases*, pp. 487-499, Santiago, Chile, September 1994.

[Agrawal et al. 93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami, "Mining Association Rules Between Sets of Items in Large Databases", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 207-287, Washington, District of Columbia, May 1993.

[Borgelt and Berthold 02] Christian Borgelt and Michael R. Berthold, "Mining Molecular Fragments: Finding Relevant Substructures of Molecules", *Proceedings of the International Conference on Data Mining*, pp. 211-218, Maebashi, Gumma, Japan, December 2002.

[Chang et al. 08] Ray-Yaung Chang, Andy Podgurski, and Jiong Yang, "Discovering Neglected Conditions in Software by Mining Dependence Graphs", *IEEE Transactions on Software Engineering*, Vol. 34, No. 5, pp. 579-596, September 2008.

[Chen and Rajlich 00] Kunrong Chen and Václav Rajlich, "Case Study of Feature Location Using Dependence Graph", *Proceedings of the 8th International Workshop on Program Comprehension (IWPC'00)*, pp. 241-249, Limerick, Ireland, June 2000.

[Dean et al. 01] Thomas R. Dean, Andrew J. Malton, and Ric Holt, "Union Schemas as a Basis for a C++ Extractor", *Proceedings of the Eighth Working Conference on Reverse Engineering*, pp. 59-67, Stuttgart, Germany, October 2001.

[Derby 04] Apache Derby Project, URL: http://db.apache.org/derby/, date created: August 2004, date accessed: December 2009.

[Doctor Garbage 09] Dr. Garbage Ltd. & Co., Control Flow Graph Factory 3.4, In 2008, Sergej Alekseev and Peter Palaga founded Dr. Garbage that specializes in debugging and development tools for Java. URL: http://www.drgarbage.com/control-flow-graph-factory-3-4.html, date created: unknown, date accessed: February 2009.

[Du et al. 07] Nan Du, Bin Wu, Xin Pei, Bai Wang, and Liutong Xu, "Community Detection in Large-Scale Social Networks", *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis,* pp. 16-25, San Jose, California, August 2007 .

[Ellson et al. 02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull, "Graphviz - Open Source Graph Drawing Tools", Lecture Notes in Computer Science, pp. 594-597, 2002.

[Ferrante et al. 87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, "The Program Dependence Graph and Its Use in Optimization", *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319-349, July 1987.

[Fischer and Meinl 04] Ingrid Fischer and Thorsten Meinl, "Graph Based Molecular Data Mining - An Overview", *Proceedings of the IEEE Conference on Systems, Man & Cybernetics*, Vol. 5, pp. 4578-4582, The Hague, South Holland, Netherlands, October 2004.

[Free Software Foundation 09] Free Software Foundation, Inc., Introduction to CFlow, Free Software Foundation, URL:http://www.gnu.org/software/cflow, date created: unknown, date accessed: July 2009.

[GrammaTech 05] GrammaTech Inc., CodeSufer Overview, GrammaTech Inc., URL: http://www.grammatech.com/products/codesurfer/overview.html, date created: unknown, date accessed: February 2009.

[Han and Kamber 06] Jiawei Han and Micheline Kamber, *Data Mining - Concepts and Techniques*, Second Edition, Morgan Kaufmann Publishers, San Francisco, California, 2006.

[Headway Software 07] Headway Software, "Controlling Architecture with Structure101", White Paper, Headway Software, Newton, Massachusetts, URL: http://www.headway-software.com/products/structure101/ ControllingArchitectur ewithStructure101.pdf, July 2007.

[Holder et al. 94] Lawrence B. Holder, Diane J. Cook, and Suranjani Djoko, "Substructure Discovery in the Subdue System", *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pp. 169-180, Seattle, Washington, July 1994.

[Horwitz et al. 88] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs", *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 35-46, Atlanta, Georgia, June 1988.

[Huan et al. 03] Jun Huan, Wei Wang, and Jan Prins, "Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism", *Proceedings of the Third IEEE International Conference on Data Mining*, pp. 549-552, Washington, District of Colombia, November 2003.

[Inokuchi et al. 99] Akihiro Inokuchi, Takashi Washio, Hiroshi Motoda1, Kouhei Kumasawa, and Naohide Arai, "Basket Analysis for Graph Structured Data", *Proceedings of the Third Pacific-Asia Conference on Methodologies for Knowledge Discovery and Data Mining*, pp. 421-433, Beijing, China, April 1999.

[Inokuchi et al. 00] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda, "An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data", *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pp. 13-23, Lyon, Rhône-Alpes Region, France, September 2000.

[JDOM 09] JDOM™ Project, An open source Java based library for manipulating XML data. URL: http://www.jdom.org, date created: unknown, date accessed: Novemeber 2009.

[King et al. 01] Ross D. King, Ashwin Srinivasan and Luc Dehaspe, "Warmr: A Data Mining Tool for Chemical Data", *Journal of Computer-Aided Molecular Design*, Vol. 15, No. 2, pp. 173-181, February 2001.

[Knijf 07] Jeroen De Knijf, "FAT-Miner: Mining Frequent Attribute Trees", *Proceedings of the 2007 ACM Symposium on Applied Computing*, pp. 417-422, Seoul, South Korea, March 2007.

[Kuramochi and Karypis 04] Michihiro Kuramochi and George Karypis, "An Efficient Algorithm for Discovering Frequent Subgraphs", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, No. 9, pp. 1038-1051, Piscataway, New Jersey, September 2004.

[Nijssen and Kok 04] Siegfried Nijssen and Joost N. Kok, "A Quickstart in Frequent Structure Mining Can Make a Difference", *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 647-652, Seattle, Washington, August 2004.

[Tan et al. 06] Pang-Ning Tan, Mihael Steinbach, and Vipin Kumar, *Introduction to Data Mining*, Addison-Wesley Publishers, Upper Saddle River, New Jersey, 2006.

[Vanetik et al. 02] N. Vanetik, E. Gudes, and S. E. Shimony, "Computing Frequent Graph Patterns from Semistructured Data", *Proceeding of the 2002 International Conference on Data Mining*, pp. 458-465, Maebashi City, Japan, December 2002.

[WEKA 08] Waikato Environment for Knowledge Analysis, URL: https://svn.scms.waikato.ac.nz/svn/weka, File Path: weka\src\main\java\weka\gui\ beans\KnowledgeFlowApp.java date created: November 2008, date accessed: December 2009.

[Witten and Frank 05] Ian H. Witten and Eibe Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, Second Edition, Morgan Kaufmann Publishers, San Francisco, California, 2005.

[Yan and Han 02] Xifeng Yan and Jiawei Han, “gSpan: Graph-Based Substructure Pattern Mining”, *Proceeding of the 2002 International Conference on Data Mining*, pp. 721-724, Maebashi, Japan, December 2002.

APPENDICES

APPENDIX A

GLOSSARY

AGM	**A**priori Based **G**raph **M**ining Algorithm – An extension of algorithm proposed by Agrawal [Agrawal et al. 93] [Inokuchi et al. 00] which discovers the association rules hidden in the frequent patterns that exist in a PDG (Program Dependence Graph).
ANT	**A**nother **N**eat **T**ool - A Java library that is used to build and configure both Java and non-Java based applications. ANT is the part of apache project and is distributed under apache open source license.
ASSOCIATION	A rule representing the co-occurrence of data items in the same instance. It is in the form of a predicate and an antecedent, e.g., if a customer buys beer then he/she is going to buy diapers as well.
ASSOCIATIVE ANALYSIS	The branch of data mining that deals with the analysis of association rules discovered from datasets.
CONFIDENCE	For an association rule $X \rightarrow Y$, confidence is the measure of frequency of occurrence of all items in itemset Y, whenever all items of X occur in a transaction.
COVERAGE	The measure of how many times an association rule's antecedent appears in the data transactions.
CROSS VALIDATION	A result validation method in which part of the dataset is held back and used to validate the results of the experiments. Sometimes the experiment is repeated several times while the part held out is rotated and used to validate the results. This variation of cross validation is called n-fold cross validation.
DATA MINING	The process of gleaning hidden knowledge nuggets from the cleansed data. It is the discovery part of the Knowledge Discovery Process.
DBMS	**D**ata**B**ase **M**anagement **S**ystem – software which implements a data store and provides data manipulation tools that helps in storing and manipulating data.
DOM	**D**ocument **O**bject **M**odel – A standard for modeling and manipulating document object in HTML, XHTML, XML and XSLT.

HTML	**H**yper **T**ext **M**arkup **L**anguage - A markup language that is used to define the structure of a document. It is widely used to publish documents on the internet.
INSTANCE	See TRANSACTION.
ITEMSET	A collection of one or more items in a transaction.
JDOM	**J**ava **D**ocument **O**bject **M**odel - An open source library that is used for modeling and manipulating XML DOM trees.
KDD	**K**nowledge **D**iscovery in **D**ata - KDD is the process of extracting knowledge from data. It consists of data cleaning, data integration, data selection, data mining, pattern evaluation, and knowledge representation.
MACHINE LEARNING	The branch of computer science that studies ways of enabling machines to learn from their previous experience, thus modifying their behavior by exhibiting good judgment in similar situations afterwards.
MAVEN	Maven (yi.=meyvn) is a Yiddish word which means accumulator of knowledge. It is a software project management tool. It provides facilities throughout the software project lifecycle from requirements definition to nightly builds, regression testing, and production deployment. Maven is a project of apache foundation and is distributed under apache open source software license.
PDG	**P**rogram **D**ependence **G**raph - A directed graph whose nodes represent the statements of the programs and each edge connecting two nodes represents the dependence of the respective statements on each other.
SAMPLING	The process of picking a small part of the data that accurately exhibits the properties of the data under investigation. Sampling is especially useful in studying large datasets whose size makes the brute force analysis prohibitively costly in terms of time and space.
SUPPORT	The percentage of the occurrence of the antecedent of an association rule (see Association).
TRANSACTION	A group of items that co-occur in a single instance. For example, in the case of market basket analysis, the items purchased during a single visit to the grocery store would represent a transaction.

WEKA	**W**aikato **E**nvironment for **K**nowledge **A**nalysis - A library developed at the University of Waikato which implements common data mining algorithms and also provides useful tools to assist various data mining tasks.
XHTML	E**x**tensible **H**yper **T**ext **M**arkup **L**anguage - It belongs to the family of extensible markup language (XML). Like hypertext markup language (HTML), it is used to format and structure documents. It provides a mean to extend HTML.
XML	E**x**tensible **M**arkup **L**anguage - A language used to describe the structure of data. Due to its ability to separate data from presentation, it is widely used by applications to present data as well as to communicate with other applications.
XSLT	**X**ML (Extensible Markup Language) **S**tylesheet **L**anguage for **T**ransformations - A language used to manipulate XML documents. It is mainly used to transform XML documents to HTML pages.

APPENDIX B

PSEUDO-CODE

This appendix contains the pseudo code of the program used to discover frequent items from a transaction dataset.

The following pseudo-code of function MAIN() depicts the steps used in the process of frequent pattern discovery.

```
MAIN ( )
{
        N = Total number of revisions;
        count = 0;
        sort all program dependence graphs in the ascending order of
version;
        while (count < N)
                do{
                Add PDG to dataset;
                UPDATE_SUMMARY_TABLE();
                GENERATE_PATTERNS ();
                count++;
                }
}
```

The following function is used to update the summary table.

```
UPDATE_SUMMARY_TABLE( )
{
     I = {Set of all columns in Table 1 MINUS version id, revision
number}
     N = Total number of revisions;
     count = 0;
    while (count < N){
        For each element of I
        do{
        SUM = SUM(SUPPORT COUNT) OF I for VERSION ID(count);
        VAL = VALUE OF I for VERSION ID(count);
        INSERT/UPDATE SUMMARY TABLE SET
        SUPPORT COUNT = SUM, SET VALUE = VAL,
        WHERE VERSION ID = COUNT AND ITEM = Ii ;
            }
         }
}
```

APPENDIX C

TRANSACTIONAL REPRESENTATION OF GRAPHS

Tan et al. explain the transactional representation of graphs as follows [Tan et al. 06]:

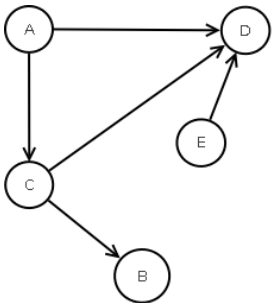
TID	Items
1	{Bread, Milk}
2	{Bread, Diapers, Beer, Eggs}
3	{Milk, Diapers, Beer, Cola}
4	{Bread, Milk, Diapers, Beer}
5	{Bread, Milk, Diapers, Cola}

The above table represents data collected in a supermarket. Each transaction shows the items purchased by customers. By analyzing the above data, different relationships can be discovered. For example, Diapers appear in four out of five transactions and in three out of those four transactions Beer also occurs. Therefore, looking at this data one can deduce that there is a high probability of a customer buying beer to buy diapers as well.

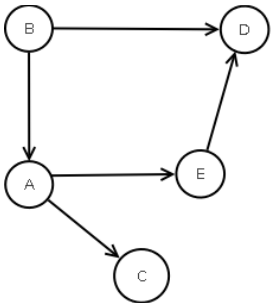
Market basket data can be represented in a binary format as shown in the following table, where each row corresponds to a transaction and each column corresponds to an item. An item can be treated as a binary variable whose value is one if the item is present in a transaction and zero otherwise.

TID	Bread	Milk	Diapers	Beer	Eggs	Cola
1	1	1	0	0	0	0
2	1	0	1	1	1	0
3	0	1	1	1	0	1
4	1	1	1	1	0	0
5	1	1	1	0	0	1

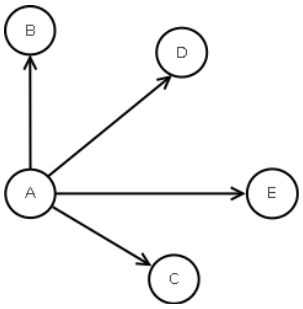
Now, let us consider an example of some graphs and their corresponding transaction-like representation.



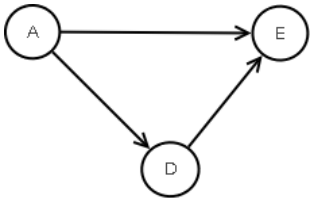
Graph G1



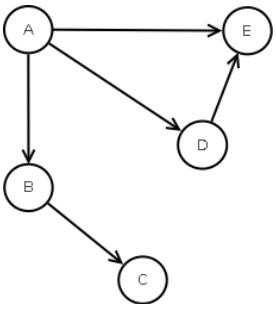
Graph G2



Graph G3



Graph G4



Graph G5

TID	A→B	A→C	A→D	A→E	B→A	B→C	B→D	C→B	C→D	D→E	E→D
1	0	1	1	0	0	0	0	1	1	0	1
2	0	1	0	1	1	0	1	0	0	0	1
3	1	1	1	1	0	0	0	0	0	0	0
4	0	0	1	1	0	0	0	0	0	1	0
5	1	0	1	1	0	1	0	0	0	1	0

Why does this approach work only when every edge is unique (every 3-tuple is unique) ?

Imagine that in Graph G2 we have two A→B nodes. Then either we should add two columns for A→B or put the value of 2 for A→B. Both of these approaches will interfere with the way we are calculating the support count and hence in frequent pattern generation.

APPENDIX D

RESULTS

This appendix contains the results of data mining experiments conducted to discover frequent patterns.

Abstract System Dependence Graphs (ASDG) of four classes from two open source projects WEKA [WEKA 08] and Derby [Derby 04] were mined for patterns. The name, number of revisions, and version numbers of these classes appear in the following table.

Class	Number of Revisions	Version Number
KnowledgeFlowApp.java	19	4698, 4762, 4786, 4796, 4804, 4806, 4829, 4899, 4953, 4966, 4984, 5134, 5226, 5244, 5396, 5611, 6013, 6015, 6140.
GridSearch.java	5	4698, 4828, 5803, 5928, 6263.
Evaluation.java	17	4698, 4838, 4842, 4899, 4997, 5072, 5093, 5162, 5197, 5678, 5685, 5688, 5714, 5928, 5987, 6041, 6344.
DRDAConnThread.java	34	497748, 515102, 515563, 515793, 517131, 534610, 534985, 542925, 545454, 556589, 557032, 562524, 570663, 574870, 581012, 611272, 613169, 614549, 617186, 631593, 631997, 632413, 632456, 633011, 642707, 666088, 674354, 700948, 701199, 703170, 734190, 899733, 901219, 924746.

EXPERIMENT 1 – KnowledgeFlowApp.java

In this experiment, different versions of class KnowledgeFlowApp.java were mined for patterns.

Details about the class and its versions appear below.

Name: KnowledgeFlowApp.java
Number of Revisions: 19
Location: /trunk/weka/src/main/java/weka/gui/beans/
Subversion URL: https://svn.scms.waikato.ac.nz/svn/weka

Version Number	Lines of Code	Date and Time	Author	Message
4698	2961	2:39:21 PM, Thursday, November 13, 2008	mhall	Move out of top level
4762	2984	4:41:48 PM, Sunday, November 23, 2008	mhall	clear all results/plots now needs confirmation from user
4786	3020	6:40:20 PM, Thursday, November 27, 2008	mhall	Updates for new log/status panel
4796	3037	11:31:23 PM, Friday, November 28, 2008	mhall	Added support for ConfigurationConstraints
4804	3061	12:49:38 AM, Sunday, November 30, 2008	mhall	More improvements to logging and status messages.
4806	3061	1:28:41 PM, Sunday, November 30, 2008	mhall	Updated for changes to BeanCommon.
4829	3061	2:18:21 PM, Thursday, December 04, 2008	mhall	Changed the fonts for component labels and connection labels from monospaced to default to improve readability under Windows
4899	3052	5:09:05 PM, Thursday, December 25, 2008	fracpete	fixed Javadoc documentation errors that showed up during Javadoc generation ("ant docs")
4953	3052	11:15:44 PM, Thursday, January 15, 2009	mhall	Fixed nasty bug that prevented a MetaBean that had been stored in the user tab from being binary serialized when the flow was saved.
4966	3081	4:57:53 PM, Monday, January 19, 2009	mhall	Altered to reflect changes to Environment.java
4984	3085	6:06:18 PM, Thursday, January 22, 2009	mhall	More changes to support local environment variables
5134	3085	7:27:41 PM, Tuesday, March 03, 2009	mhall	Removed a superfluous method from BeanCommon

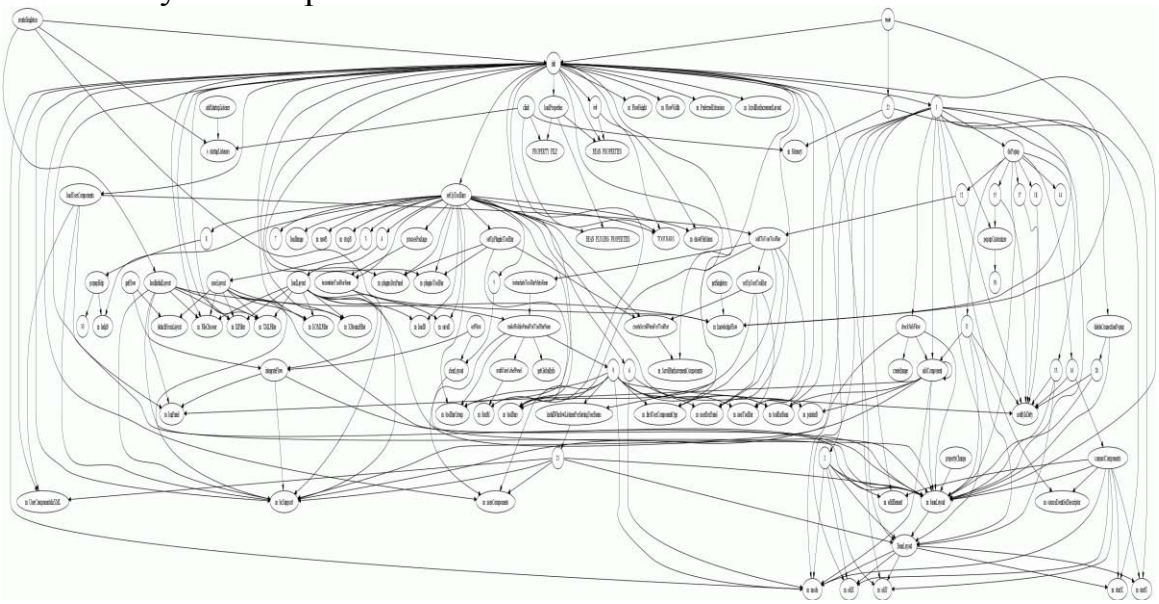
5226	3128	8:41:44 PM, Tuesday, April 07, 2009	mhall	Now checks to see if a component is Startable and adds it's startMessage to the contextual popup menu. Components that are Startable or implement userRequests get started or requests executed in a separate thread. This should remove the burden of launching computationally expensive tasks in separate threads from individual components (unless, like Classifier, they launch multiple tasks in parallel).
5244	3128	7:59:06 PM, Tuesday, April 14, 2009	mhall	Small fix for handling of Startable and UserRequestAcceptor.
5396	3136	9:54:56 PM, Sunday, May 24, 2009	mhall	Modifications to support GUI access to environment variables.
5611	3151	7:16:39 PM, Tuesday, June 16, 2009	mhall	Managed to break the linking in and out of MetaBeans - now fixed again. Plus some improvements to the labeling of events, actions etc. for MetaBeans.
6013	3155	6:36:55 PM, Thursday, October 01, 2009	mhall	Now allows outgoing connections to be made from all beans encapsulated in a MetaBean and not just those defined to be "output" beans.
6015	3164	12:24:33 AM, Friday, October 02, 2009	mhall	Added some code to ensure that an input in the subflow of a MetaBean is disconnected as a source for any target beans that are outside of the subflow (was causing serialization problems when adding to the user tab).
6140	3169	7:12:33 AM, Friday, December 11, 2009	mhall	Now creates the plugins directory (if not already existing) on exit.

Patterns discovered in each version of the class are grouped into classes (for a definition of the six classes, please see Section 5.2).

VERSION 4698 – Failed Build. There was some issues with the build of this version and the project did not compile.

VERSION 4762 (Revision 0)

C1 – Newly formed patterns



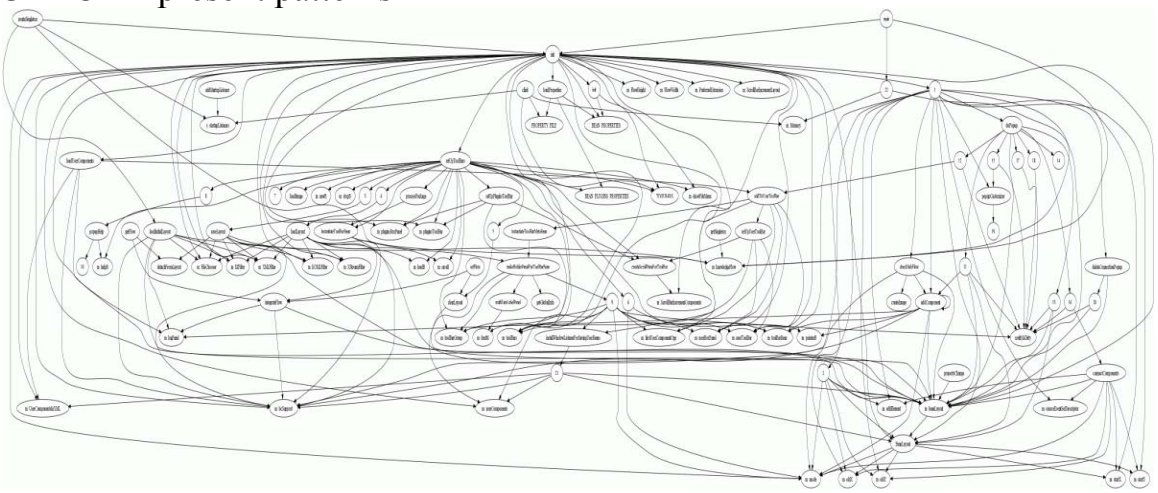
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 –Impermanent patterns

NONE FOUND

C4 - Omnipresent patterns



C5 – Disappearing omnipresent patterns

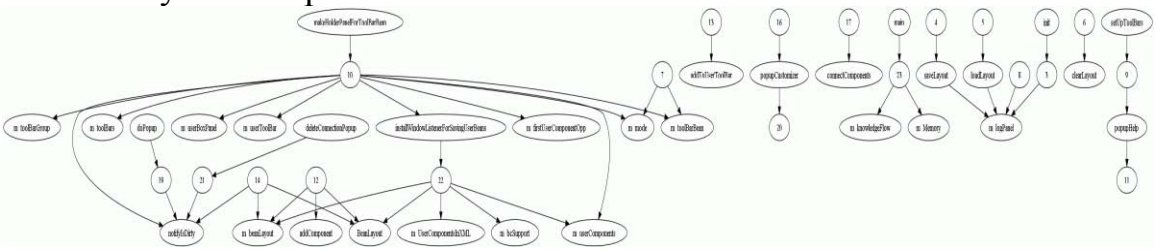
NONE FOUND

C6 - Reincarnated patterns

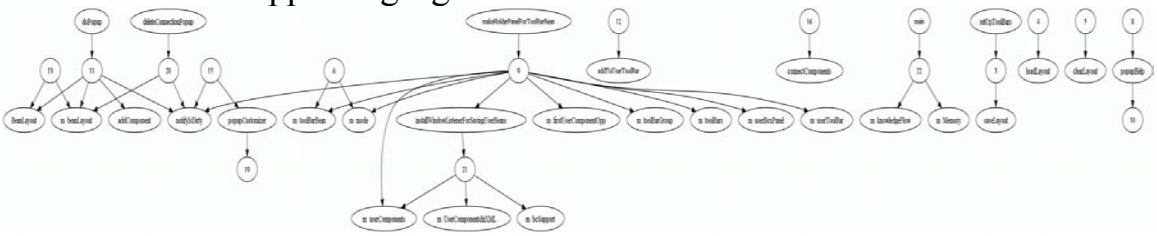
NONE FOUND

VERSION 4786 (Revision 1)

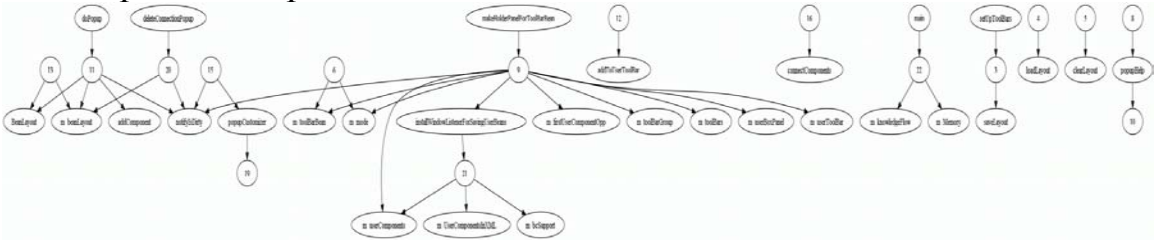
C1 – Newly formed patterns



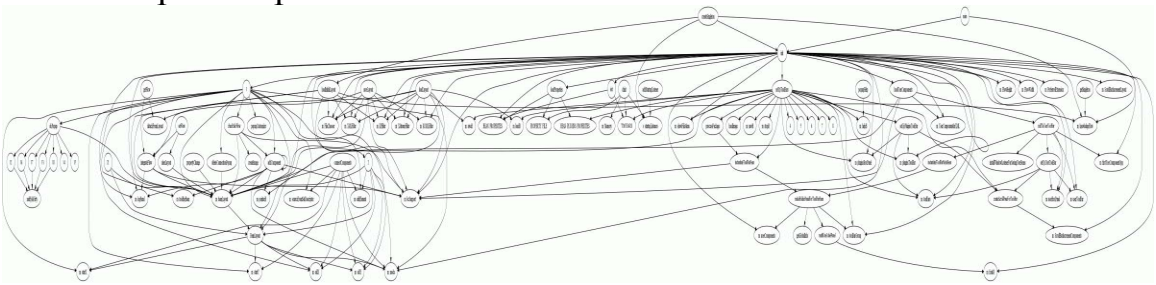
C2 – Patterns disappearing right after their creation



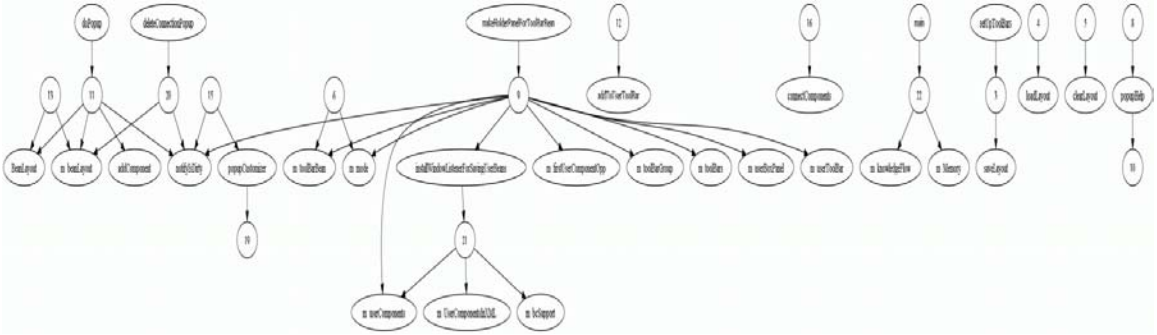
C3 – Impermanent patterns



C4 - Omnipresent patterns



C5 – Disappearing omnipresent patterns

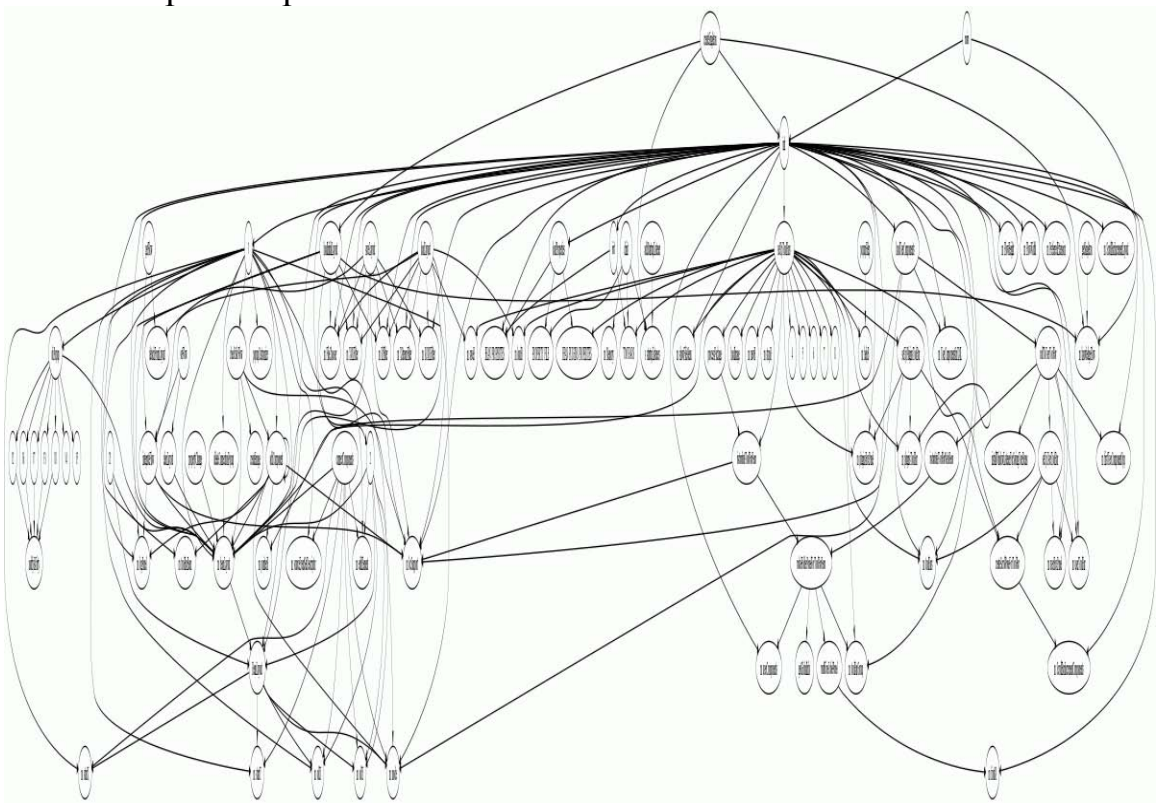


C6 - Reincarnated patterns

NONE FOUND

VERSION 4796 (Revision 2)

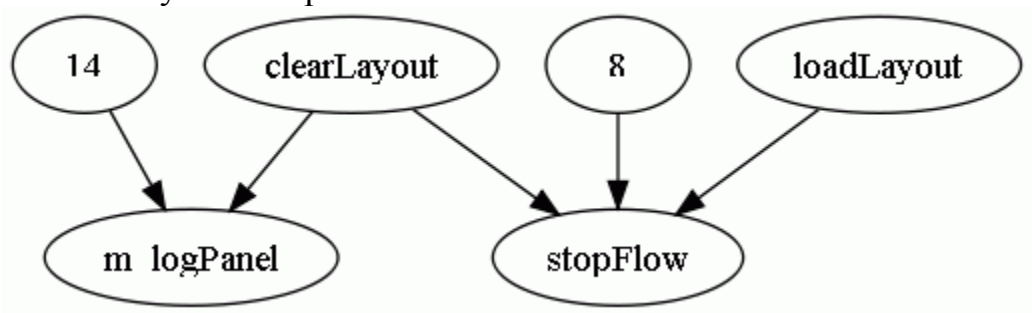
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 4804 (Revision 3)

C1 – Newly formed patterns



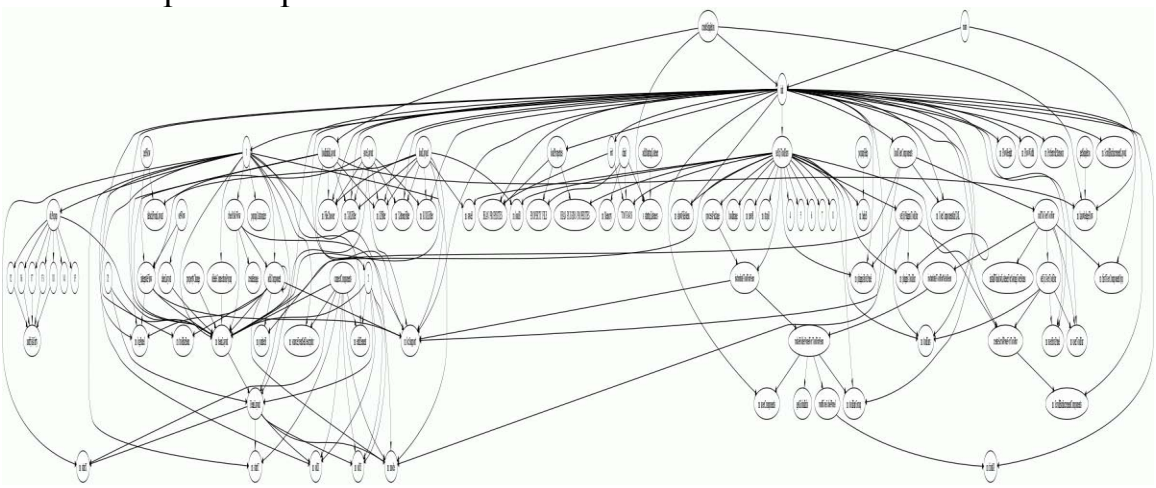
C2 – Patterns disappearing right after their creation

NONE FOUND

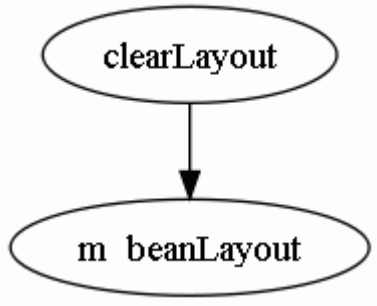
C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns



C6 – Reincarnated patterns

NONE FOUND

VERSION 4806 (Revision 4)

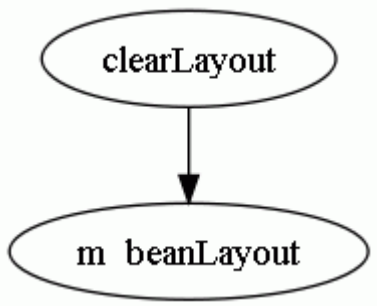
C1 – Newly formed patterns

NONE FOUND

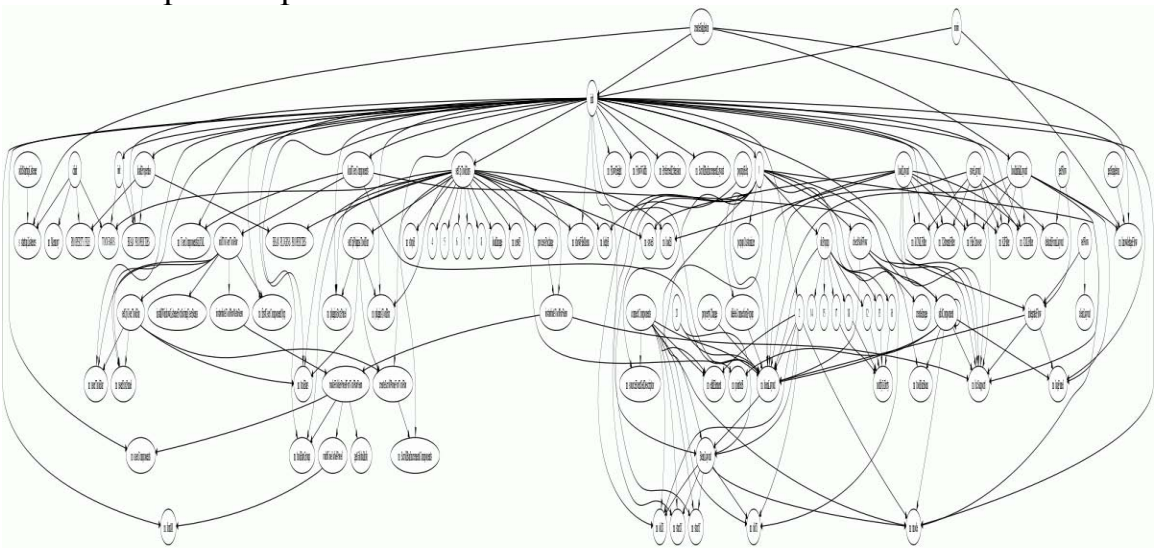
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns



C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

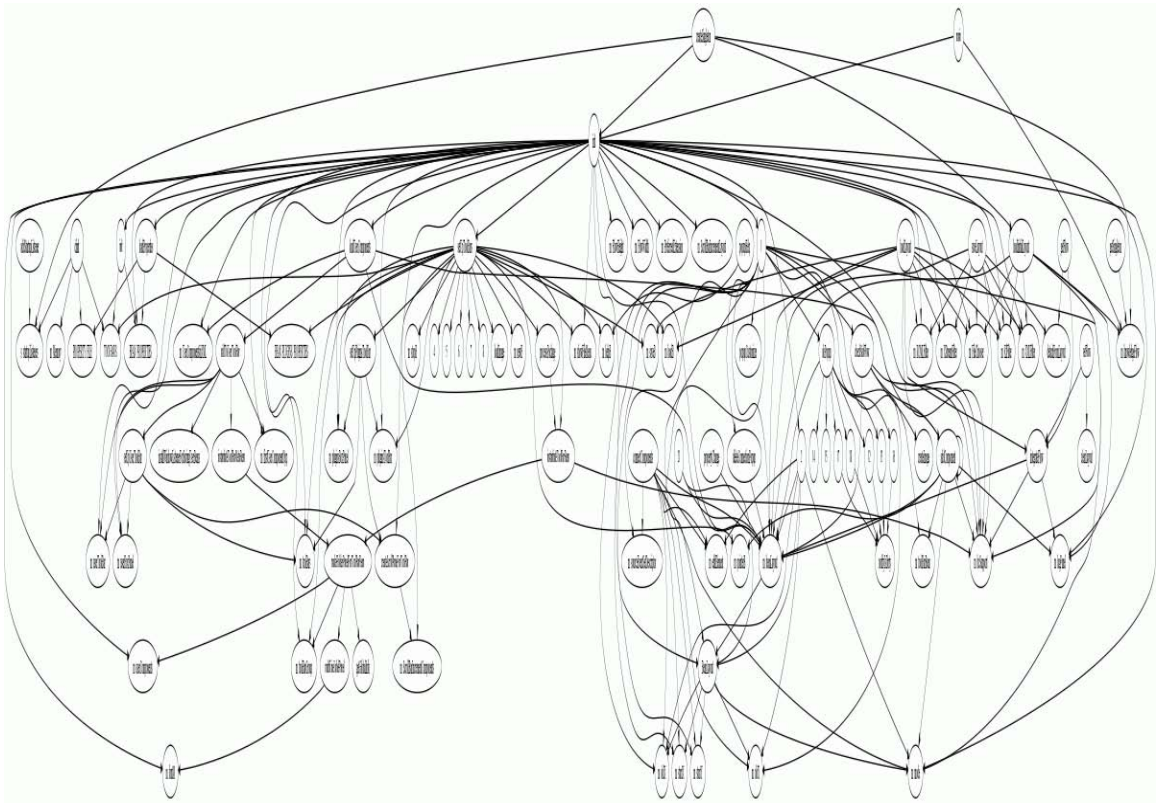
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 4829 (Revision 5)

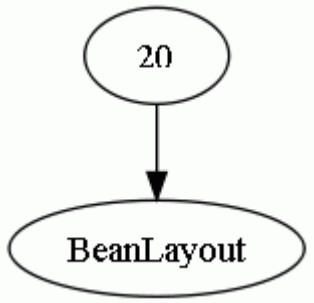
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 4899 (Revision 6)

C1 – Newly formed patterns



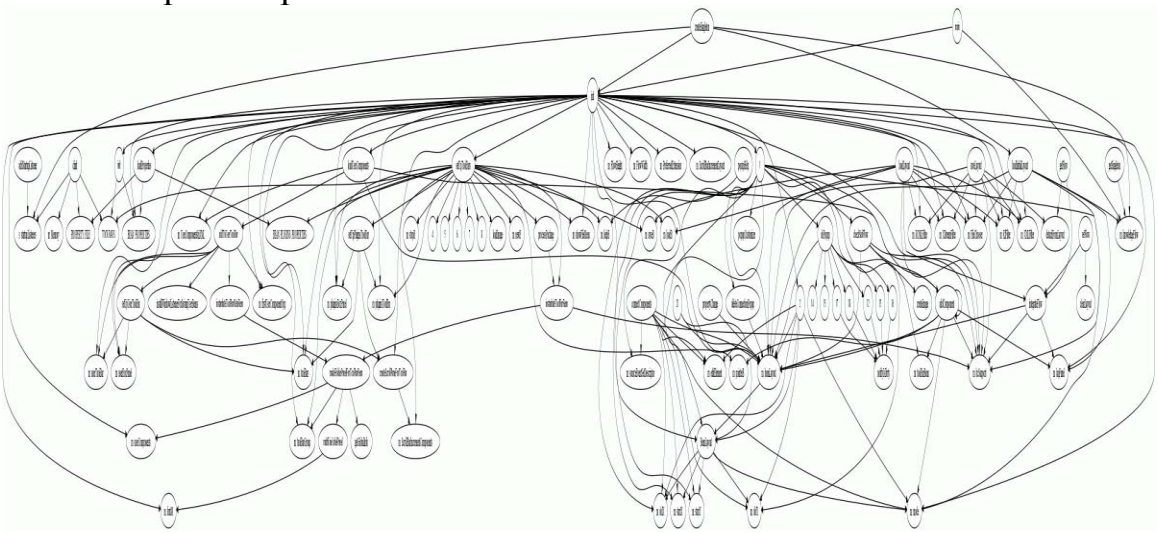
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

NONE FOUND

C6 – Reincarnated patterns

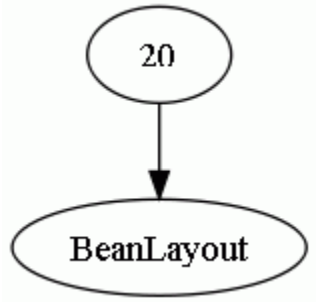
NONE FOUND

VERSION 4953 (Revision 7)

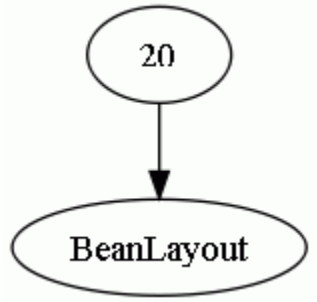
C1 – Newly formed patterns

NONE FOUND

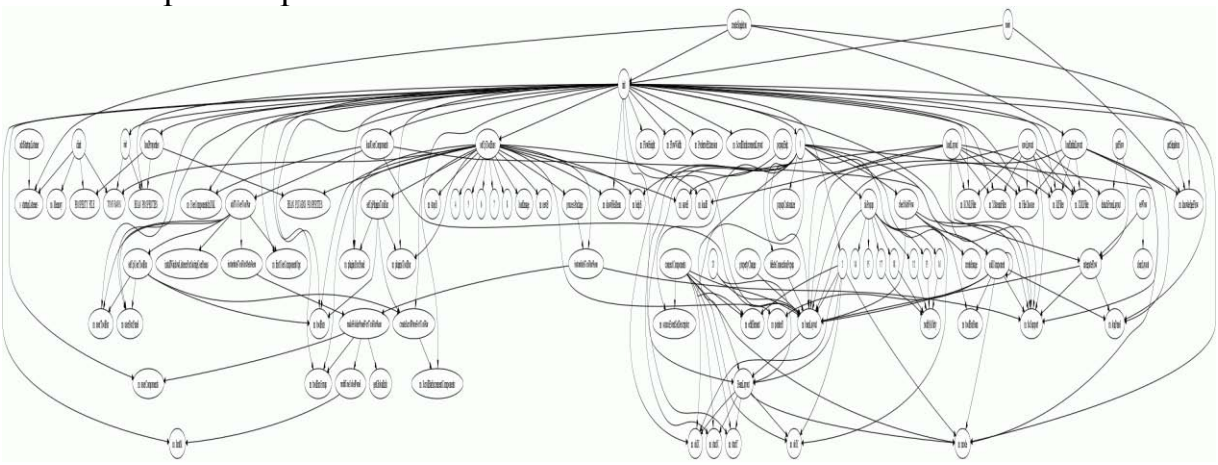
C2 – Patterns disappearing right after their creation



C3 – Impermanent patterns



C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

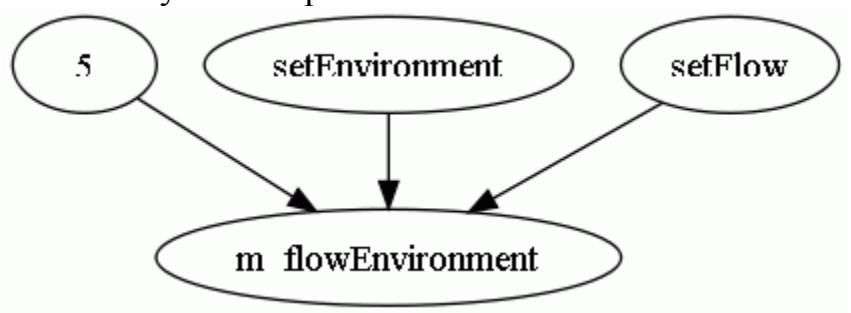
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 4966 (Revision 8)

C1 – Newly formed patterns



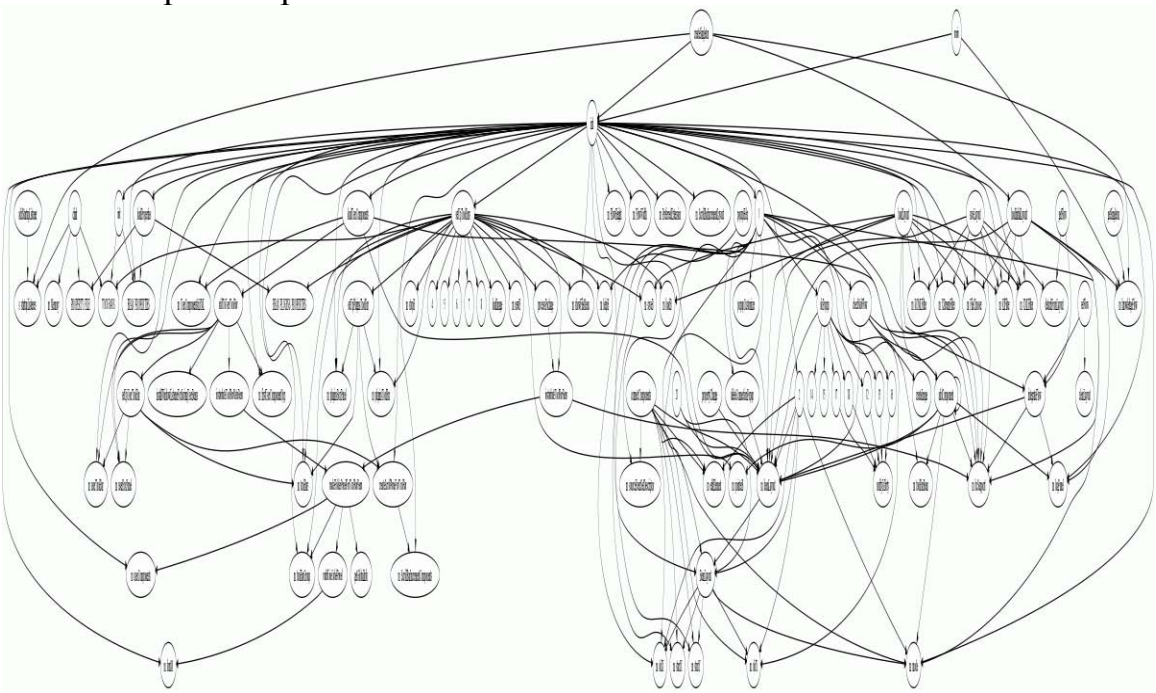
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

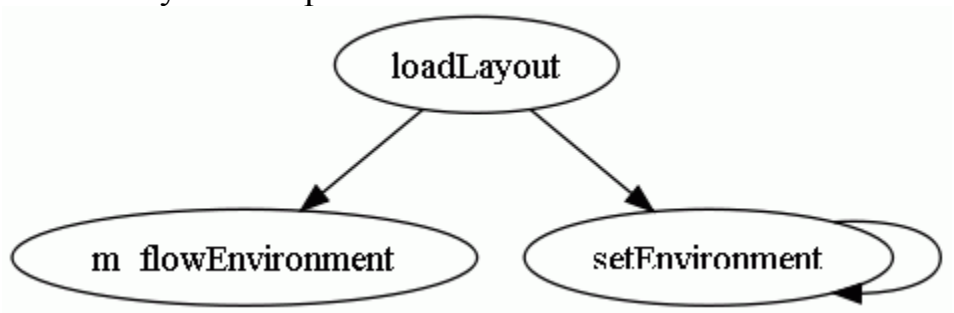
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 4984 (Revision 9)

C1 – Newly formed patterns



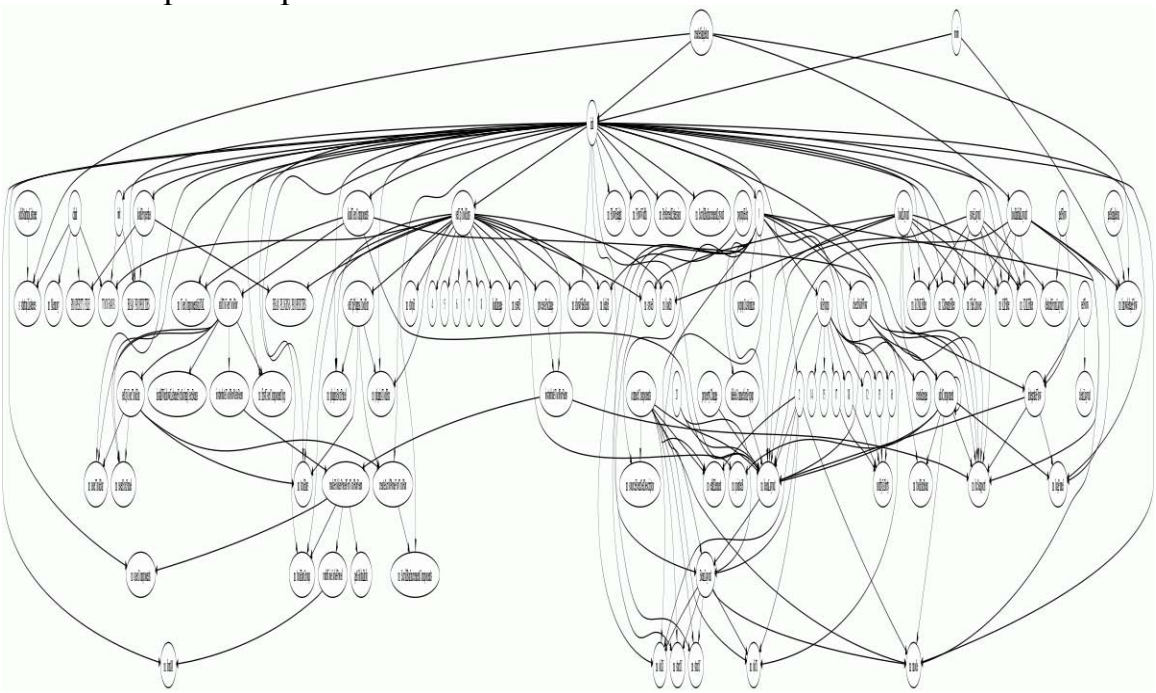
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

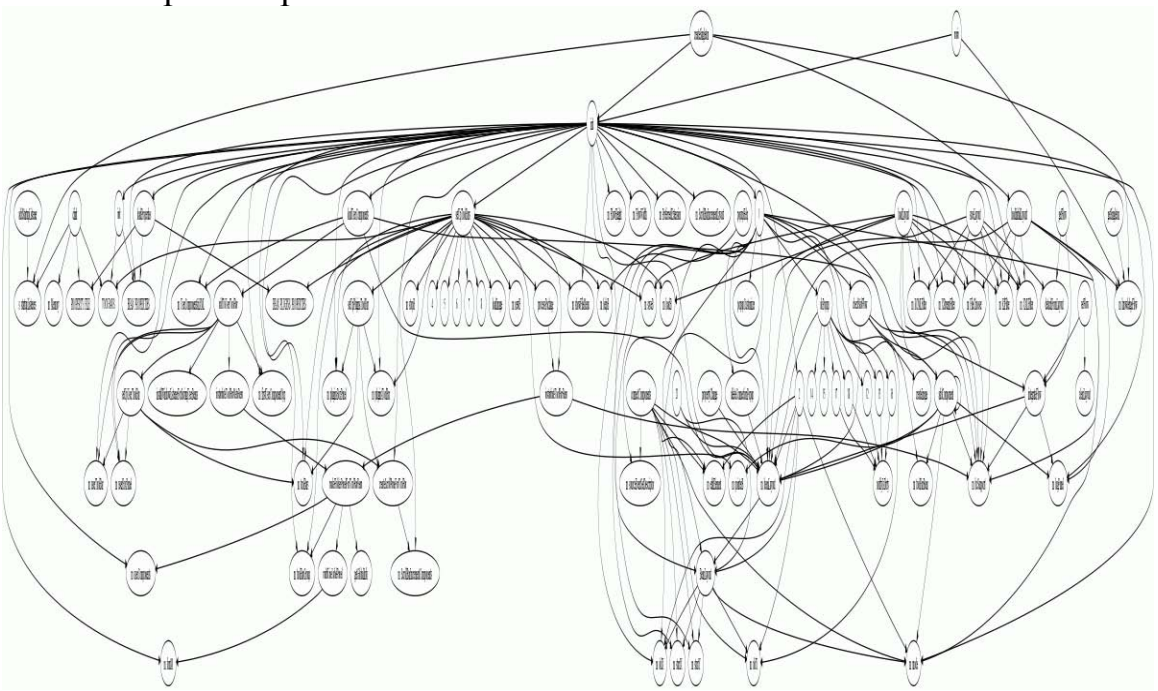
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 5134 (Revision 10)

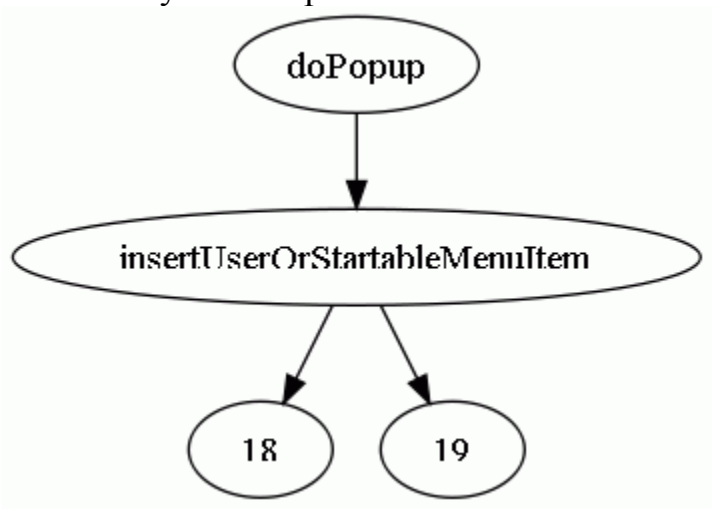
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 5226 (Revision 11)

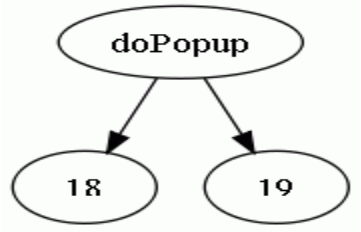
C1 – Newly formed patterns



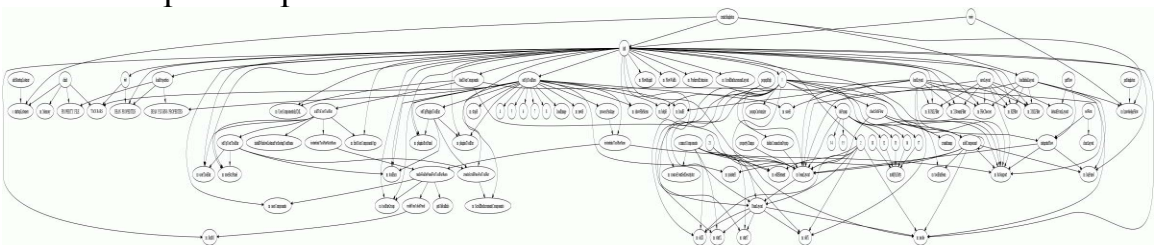
C2 – Patterns disappearing right after their creation

NONE FOUND

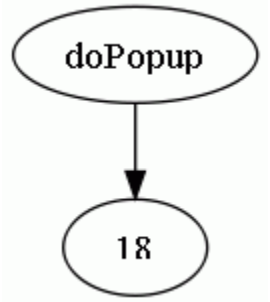
C3 – Impermanent patterns



C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

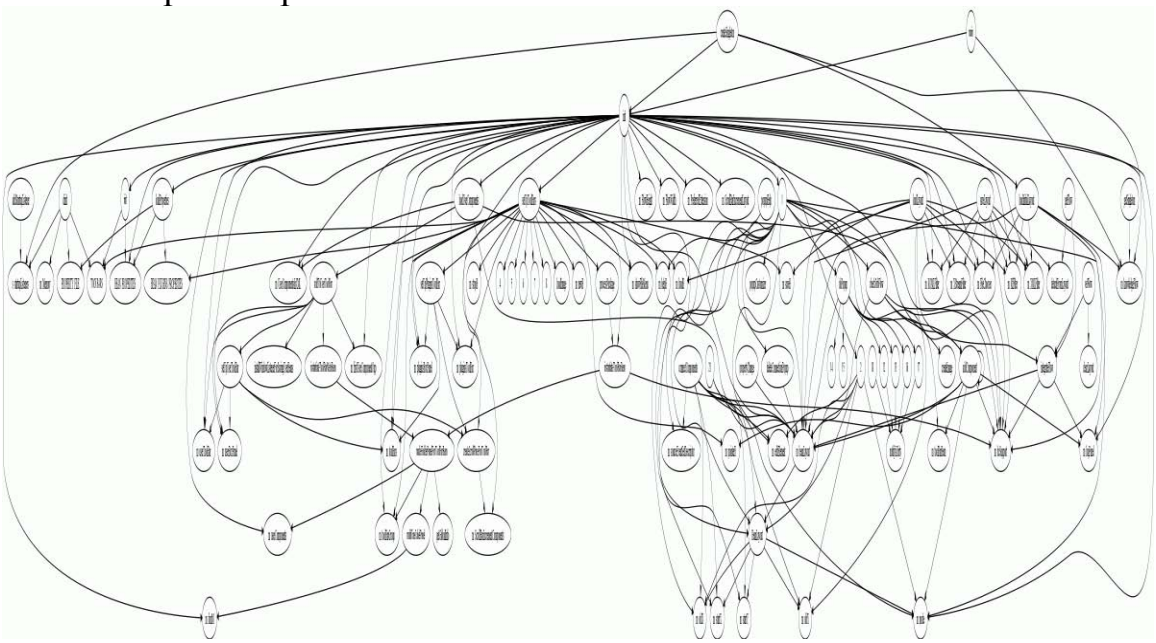


C6 – Reincarnated patterns

NONE FOUND

VERSION 5244 (Revision 12)

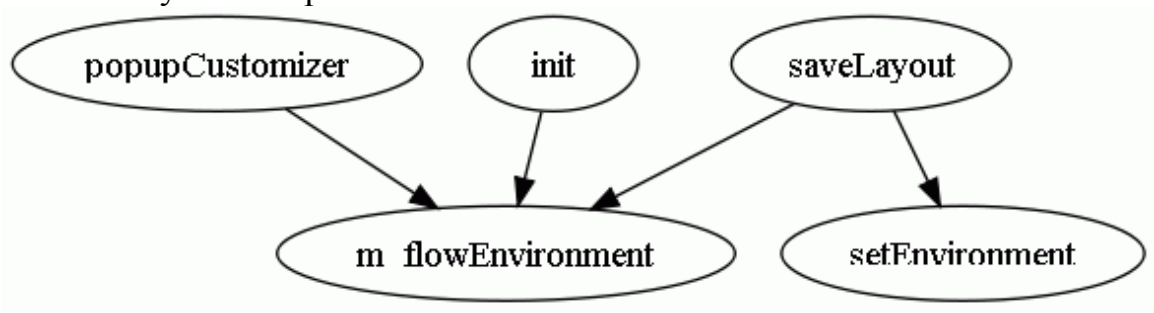
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 5396 (Revision 13)

C1 – Newly formed patterns



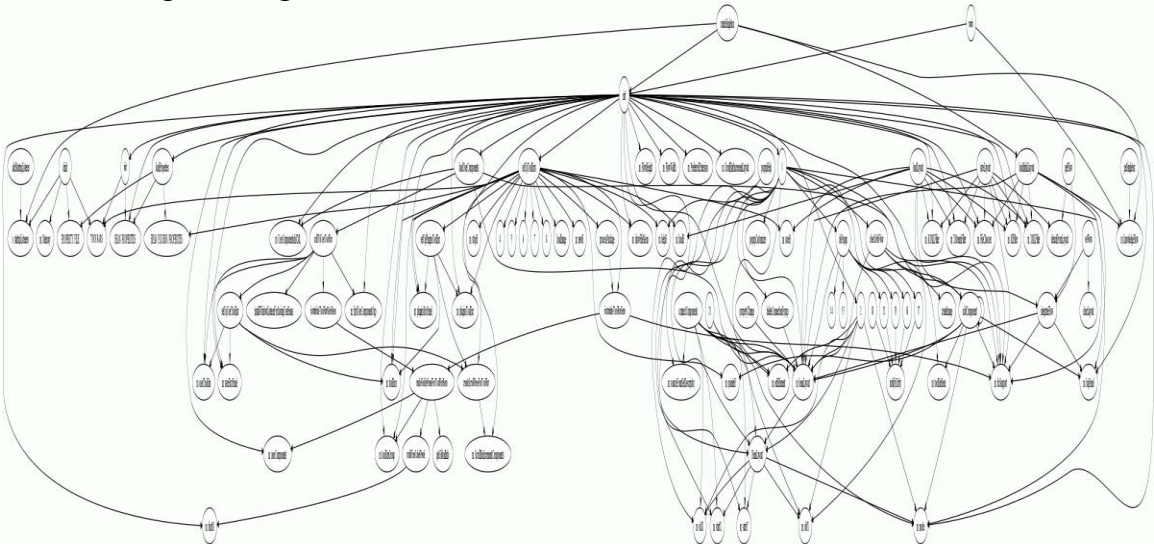
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

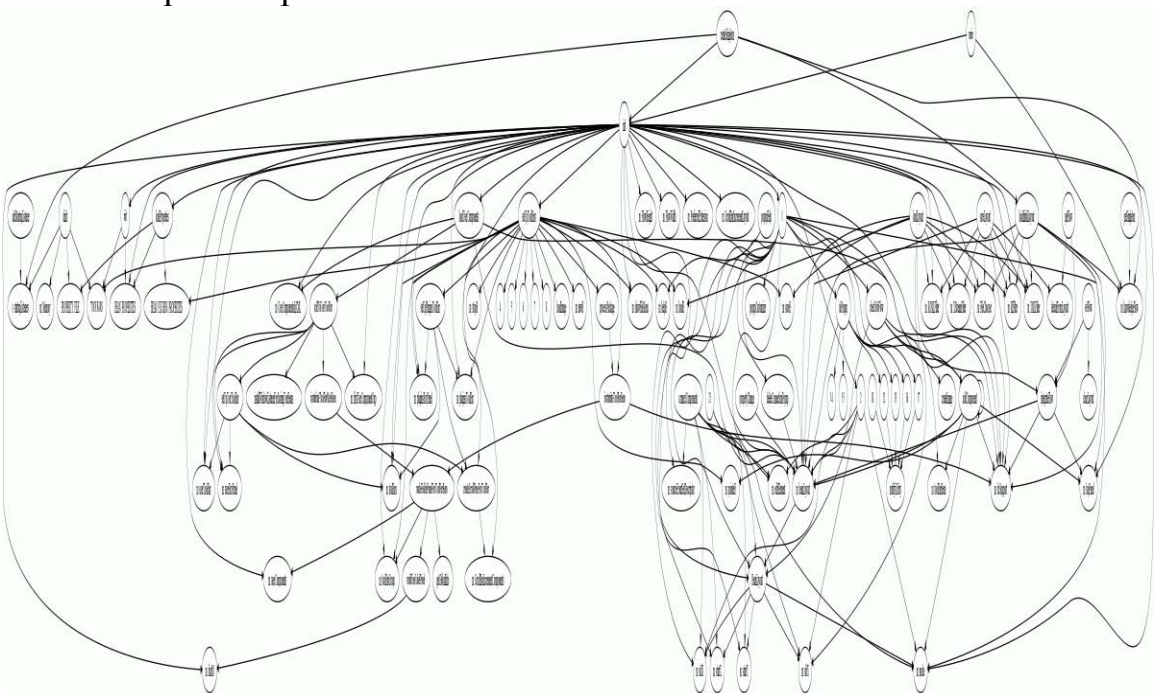
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 5611 (Revision 14)

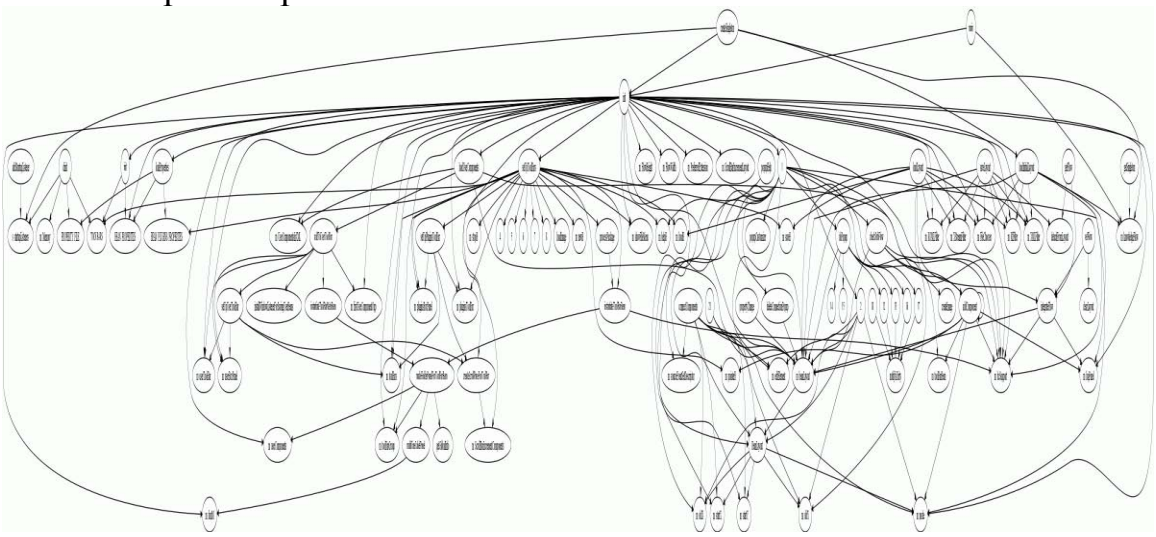
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 6013 (Revision 15)

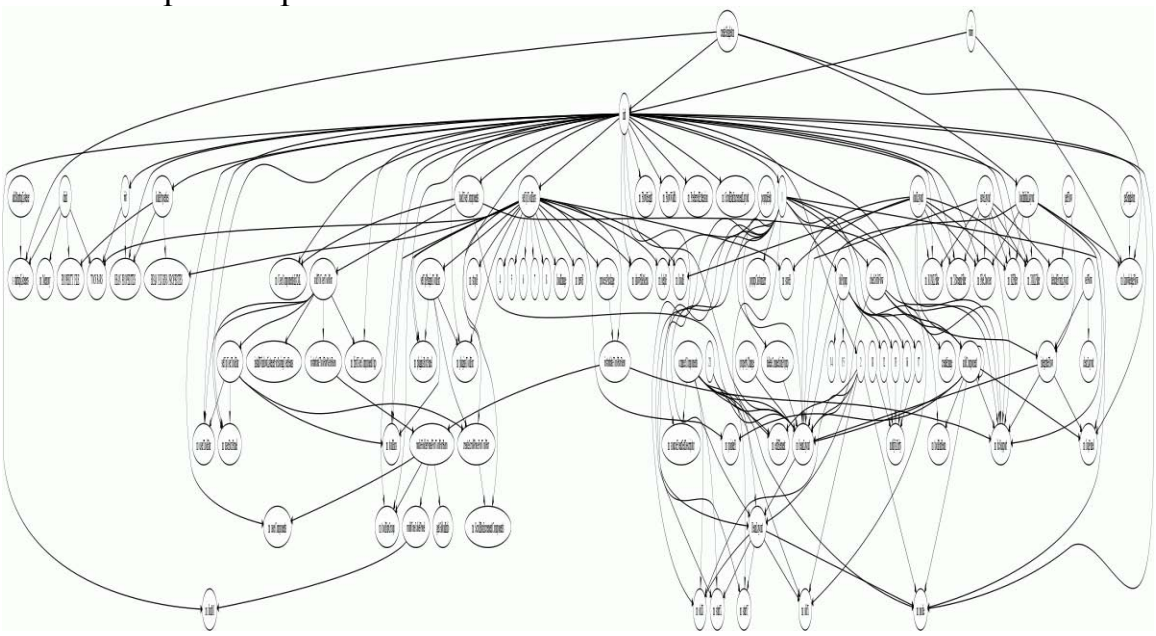
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 6015 (Revision 16)

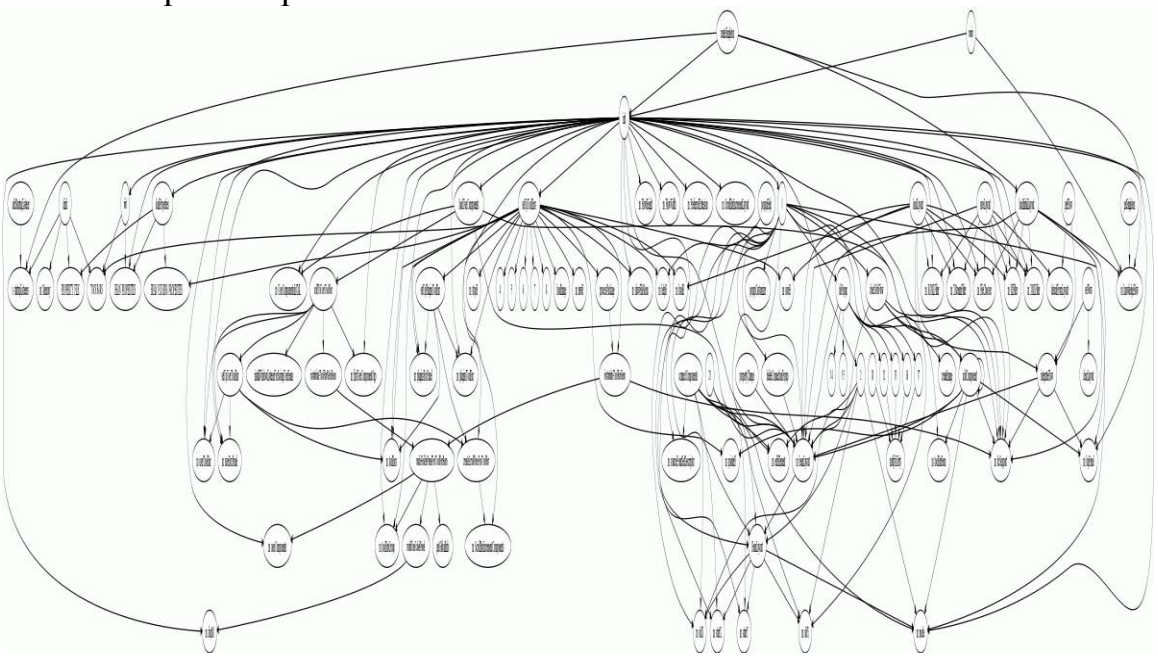
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 6140 (Revision 17)

C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

EXPERIMENT 2 – Evaluation.java

In this experiment, different versions of class Evaluation.java were mined for patterns.

Details about class Evaluation.java and its versions appear below.

Name: Evaluation.java
Number of Revisions: 17
Location: /trunk/weka/src/main/java/weka/classifiers
Subversion URL: https://svn.scms.waikato.ac.nz/svn/weka

Version Number	Lines of Code	Date and Time	Author	Message
4698	3650	2:39:21 PM, Thursday, November 13, 2008	mhall	Move out of top level
4838	3706	12:28:31 AM, Tuesday, December 09, 2008	mhall	Added a general option to output the global information (synopsis) for the classifier.
4842	3713	6:46:31 PM, Tuesday, December 09, 2008	mhall	Now no longer prints classification summary and confusion matrix when -p option is selected (i.e. now more like what the book version outputs in this situation).
4899	3713	5:09:05 PM, Thursday, December 25, 2008	fracpete	fixed Javadoc documentation errors that showed up during Javadoc generation ("ant docs")
4997	3713	5:52:20 PM, Sunday, January 25, 2009	fracpete	added new Instances.equalHeadersMsg(...) method to errors message for more informative output (why the equality test failed)
5072	3713	1:29:03 PM, Sunday, February 15, 2009	mhall	Now suppresses detailed class information when printing predictions.
5093	3752	7:11:05 PM, Monday, February 23, 2009	mhall	Added George Forman's unweighted micro and macro averaged F-measure methods.
5162	3764	2:33:54 PM, Wednesday, March 18, 2009	fracpete	fixed source code generation: generated classifier didn't implement getRevision() method
5197	3539	6:52:13 PM, Sunday, March 29, 2009	fracpete	I unified the generation of predictions in Explorer and on command-line by introducing now class hierarchy derived from weka.classifiers.evaluation.

				output.prediction.AbstractOutput this allows the implementation of custom output generators; initially: PlainText (original format), CSV and HTML
5678	3619	8:42:02 PM, Wednesday, June 24, 2009	eibe	Implemented code for evaluation of conditional density estimates and interval estimators and reduced code redundancy. Note that SF_* statistics in numeric prediction case have changed---they can now be used to evaluate conditional density estimates---and are based on a simpler kernel estimator as well.
5685	3624	5:33:24 PM, Thursday, June 25, 2009	eibe	Entropy statistics are now output again for nominal class problems. Also outputs coverage statistics and rel. interval widths for nominal class problems now.
5688	3624	8:52:41 PM, Thursday, June 25, 2009	eibe	Small bug fix for case where classifier implements only one of IntervalEstimator and ConditionalDensityEstimator: now the correct statistics are output in that case.
5714	3645	11:22:47 PM, Thursday, July 02, 2009	fracpete	NumericPrediction class can store the prediction intervals returned by IntervalEstimator classifiers now as well Evaluation class now records numeric predictions as well (and prediction intervals returned by IntervalEstimator classifiers); added member variable to store header information of dataset (m_Header, getHeader()), which means that one can generated classifier errors visualizations based on an Evaluation object now (for nominal and numeric class attributes)

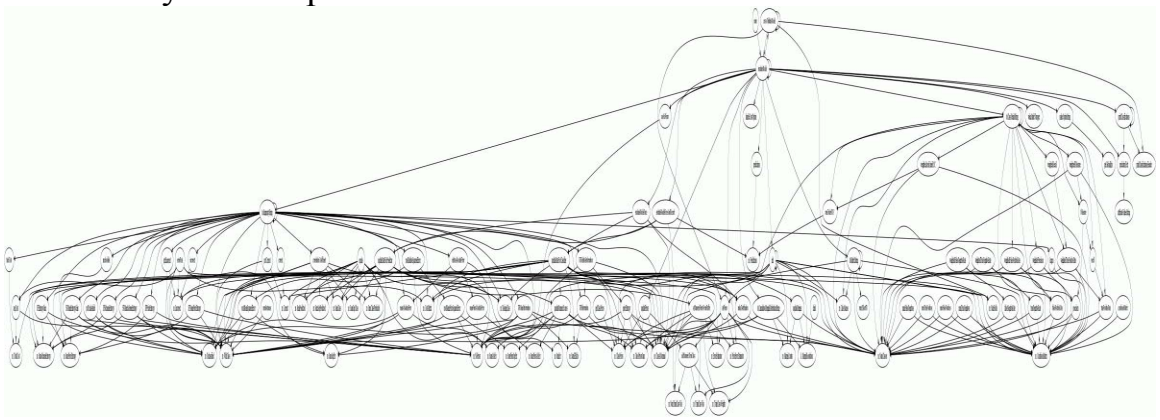
				moved code for generating classifier errors and cluster assignments in Explorer into separate classes (AbstractPlotInstances, ClassifierErrorsPlotInstances, ClusterAssignmentsPlotInstances); the KnowledgeFlow's ClassifierPerformanceEvaluator uses this code now as well; the labels for the cluster assignments now start at 1, to be consistent with the AddCluster filter
5928	3647	5:07:59 PM, Wednesday, September 02, 2009	eibe	Classifier.java is now an interface. Added new class AbstractClassifier.java.
5987	3647	3:44:39 PM, Thursday, September 24, 2009	eibe	Instance is an interface now. There is also an AbstractInstance, with two concrete sub classes DenseInstance and SparseInstance. DenseInstance objects are now used wherever Instance objects where used previously.
6041	3641	5:40:33 PM, Thursday, October 22, 2009	mhall	Code formatting tidy up courtesy of Michael C. Harris. Trailing space removed; reformat to two-space indent; join statements that are split over multiple lines if the result is less than 80 characters.
6344	3642	11:27:25 PM, Friday, March 12, 2010	mhall	Made the computation of training/test split size when doing a percentage split consistent with how it is done in the Explorer and TrainTestSplitMaker.

Patterns discovered in each version of the class are grouped into classes (for a definition of the six classes, please see Section 5.2).

VERSION 4698 – Failed Build. There were some issues with the build of this version and the project did not compile.

VERSION 4838 (Revision 0)

C1 – Newly formed patterns



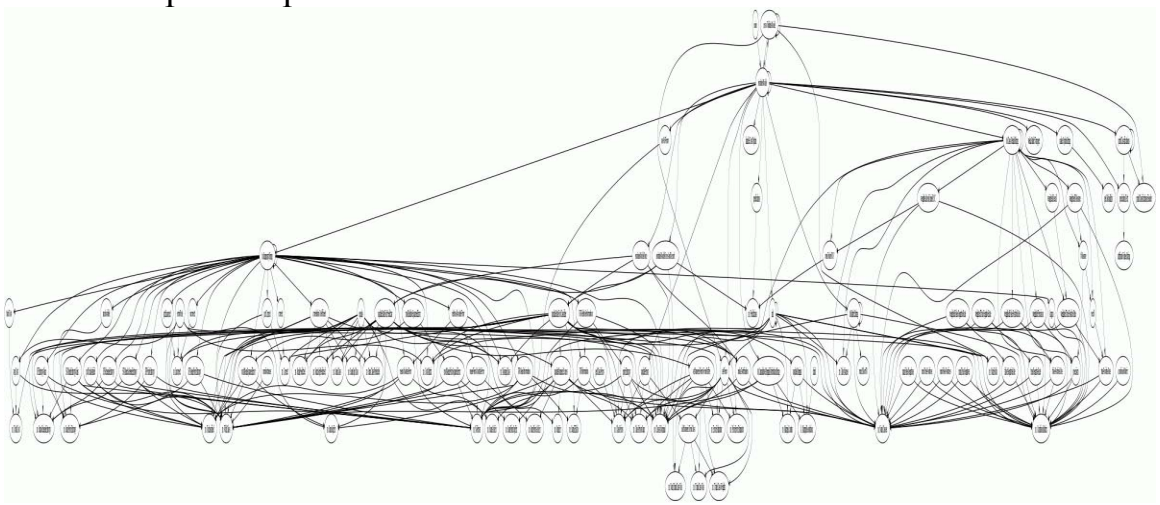
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 4842 (Revision 1)

C1 – Newly formed patterns

NONE FOUND

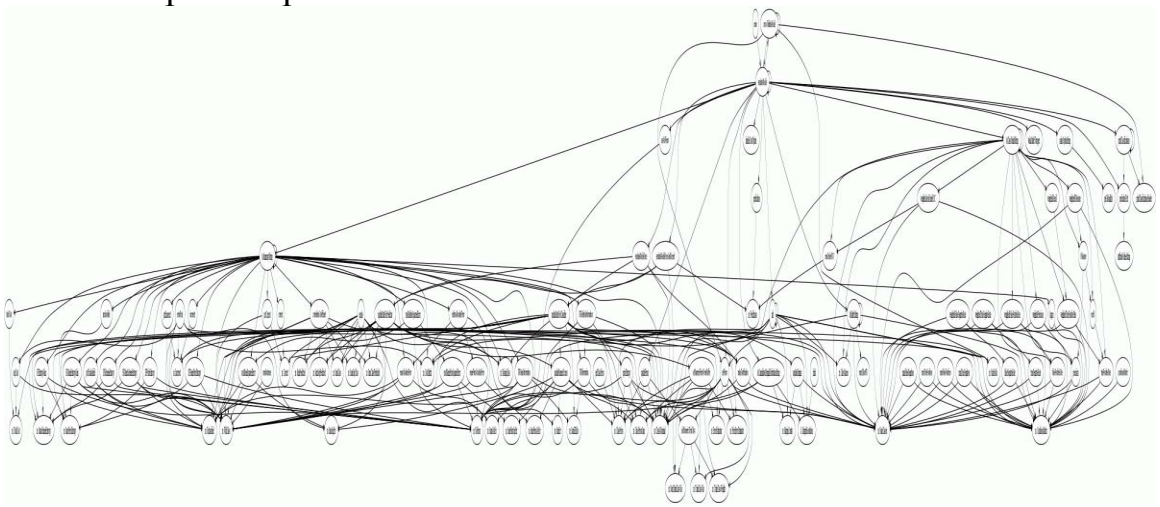
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

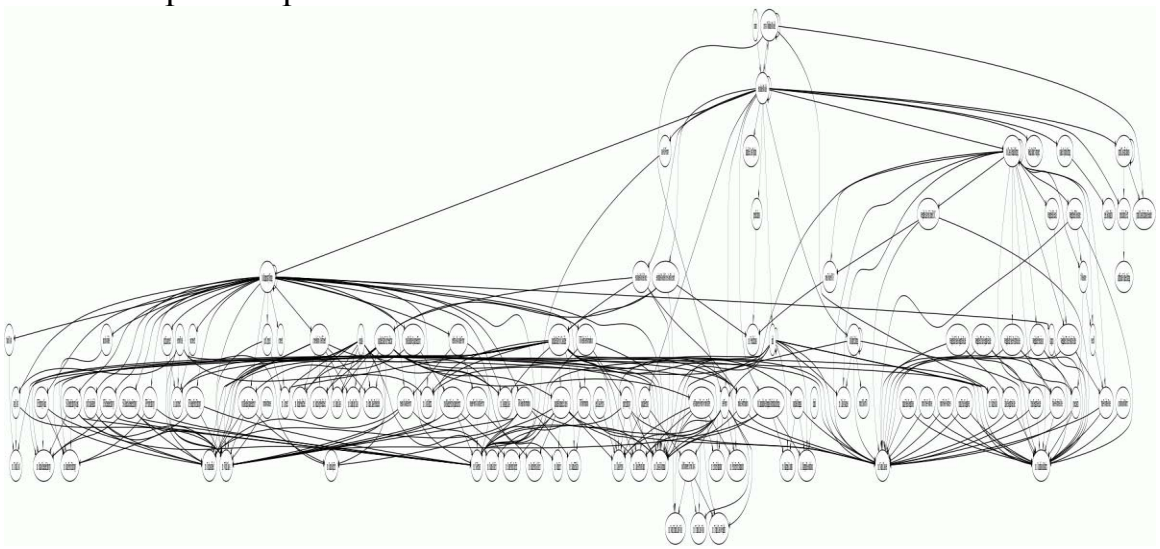
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 4899 (Revision 2)

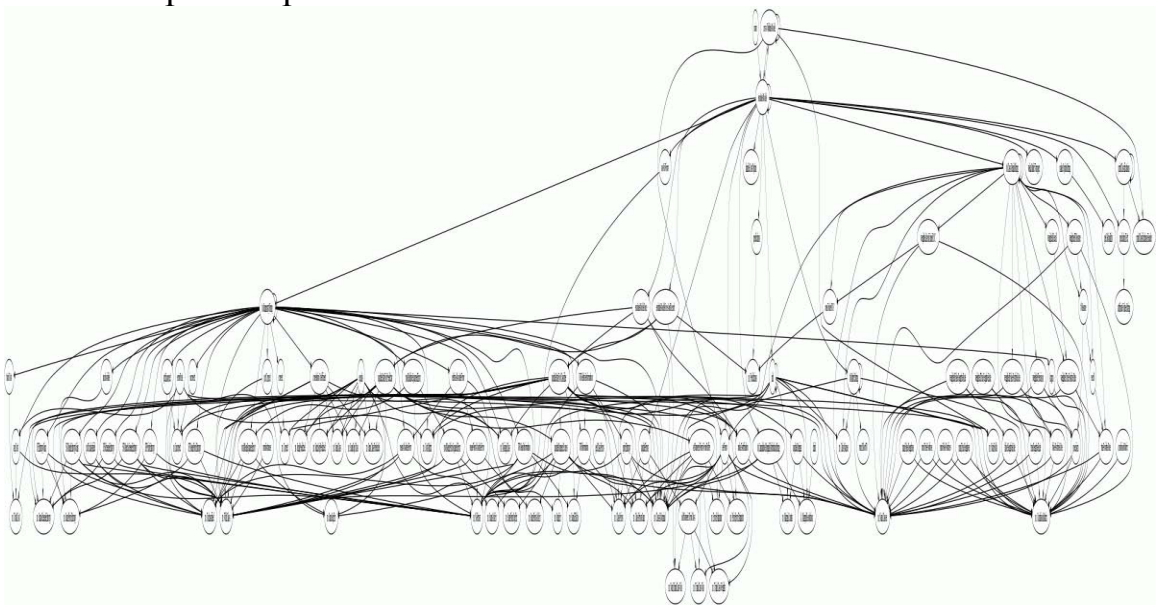
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 4997 (Revision 3)

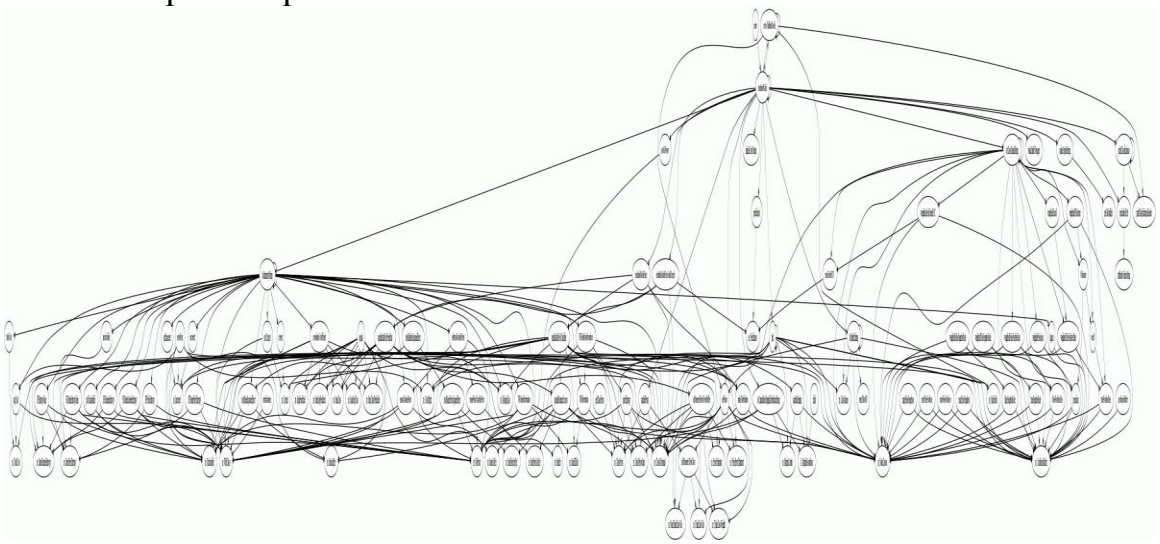
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 5072 (Revision 4)

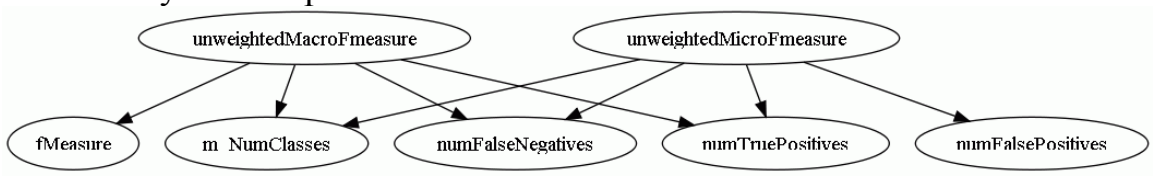
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 5093 (Revision 5)

C1 – Newly formed patterns



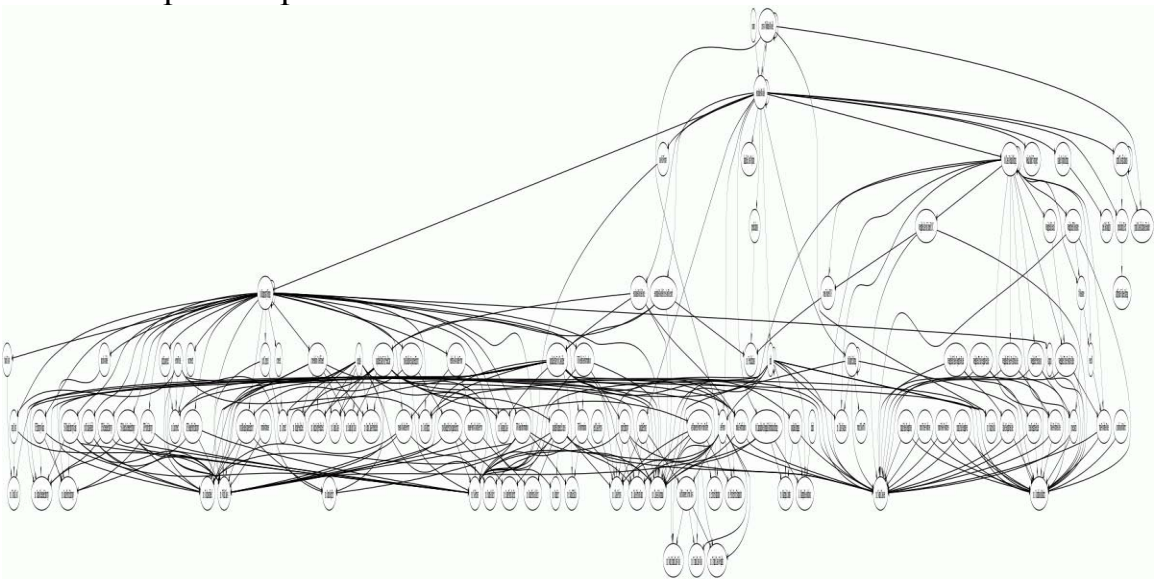
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

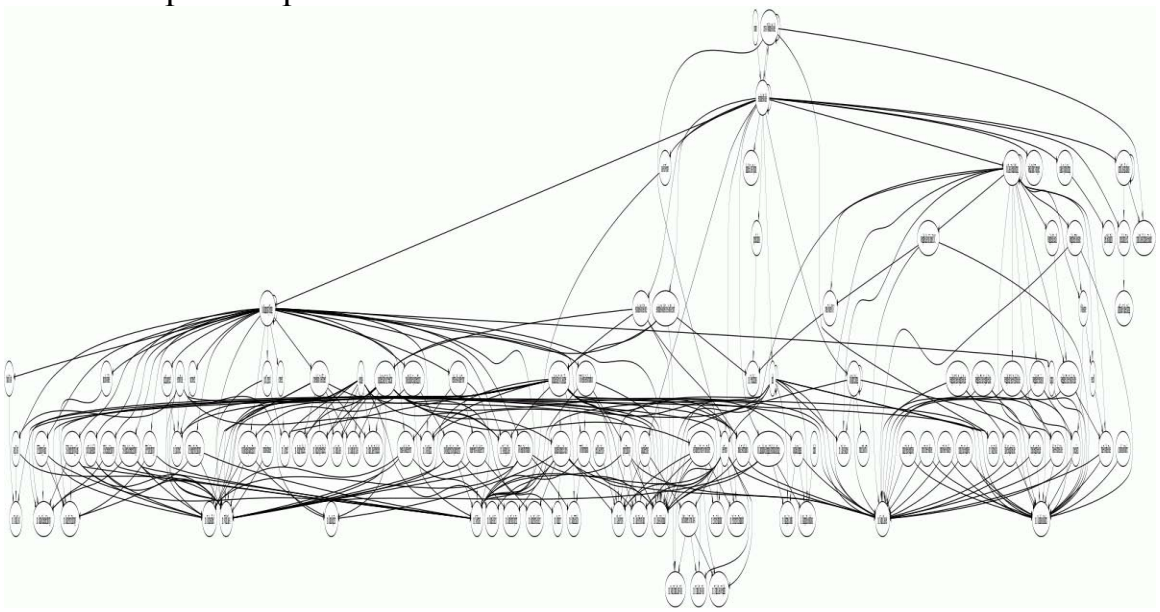
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 5162 (Revision 6)

C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 5197 (Revision 7)

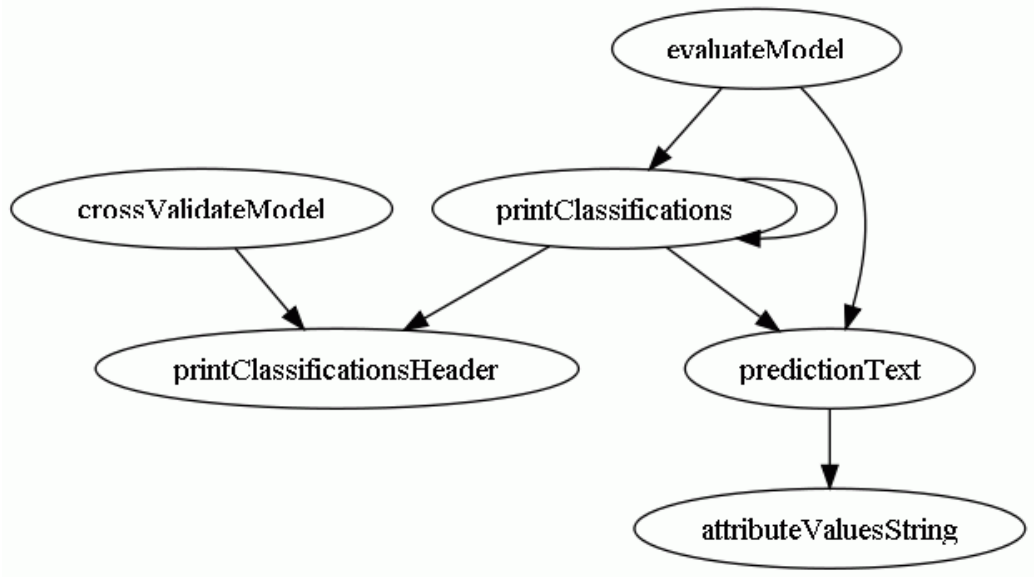
C1 – Newly formed patterns

NONE FOUND

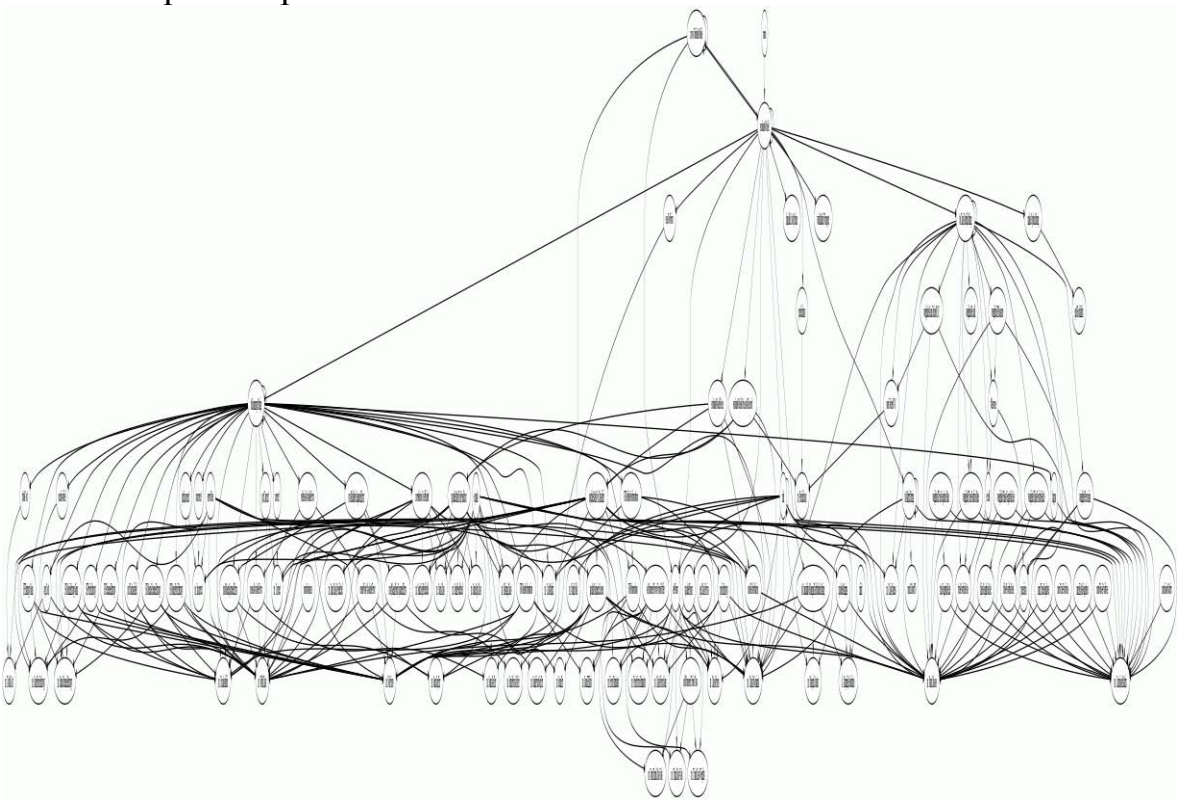
C2 – Patterns disappearing right after their creation

NONE FOUND

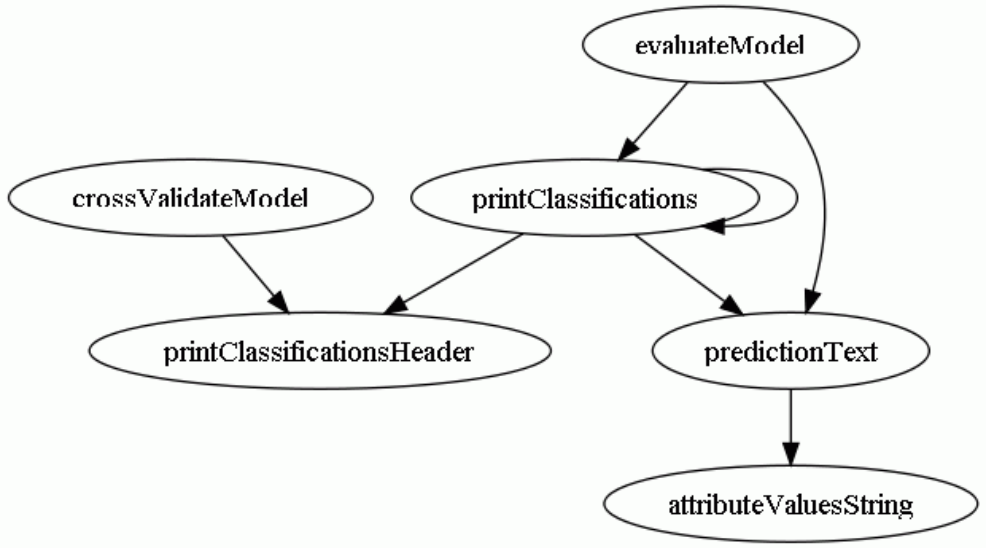
C3 – Impermanent patterns



C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

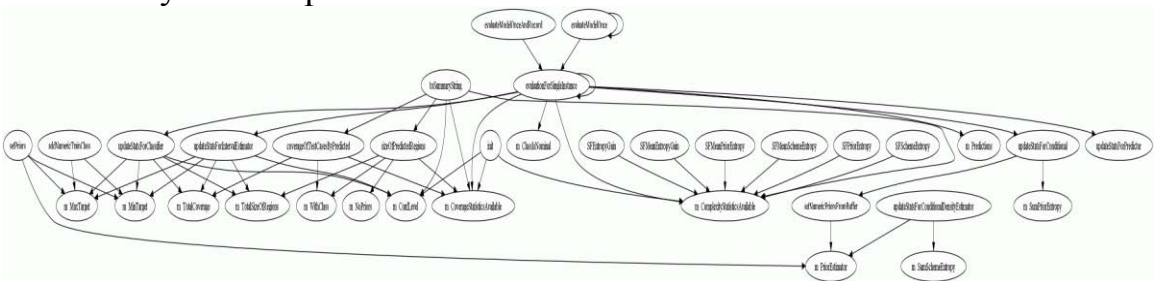


C6 – Reincarnated patterns

NONE FOUND

VERSION 5685 (Revision 8)

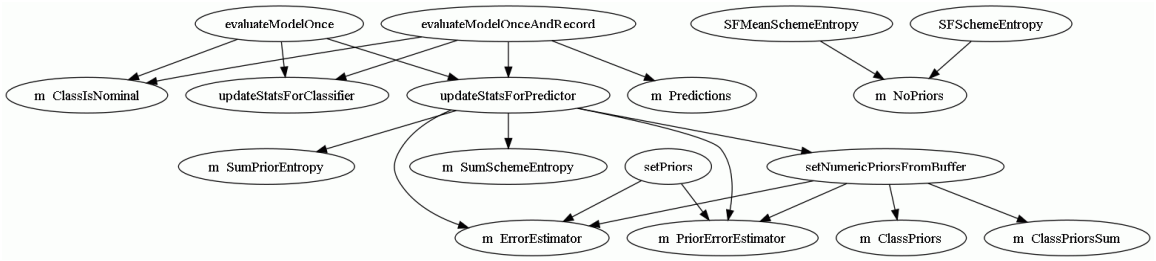
C1 – Newly formed patterns



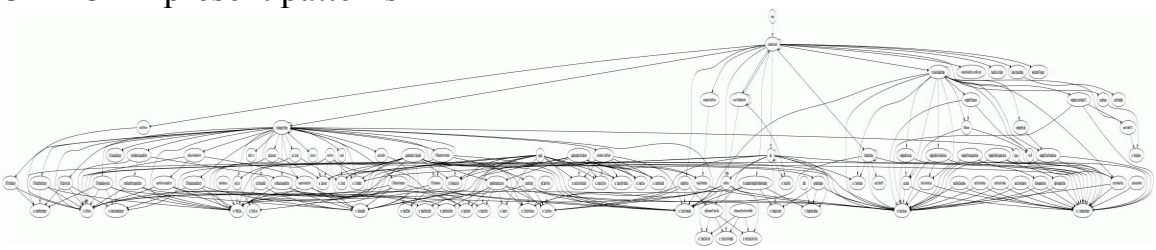
C2 – Patterns disappearing right after their creation

NONE FOUND

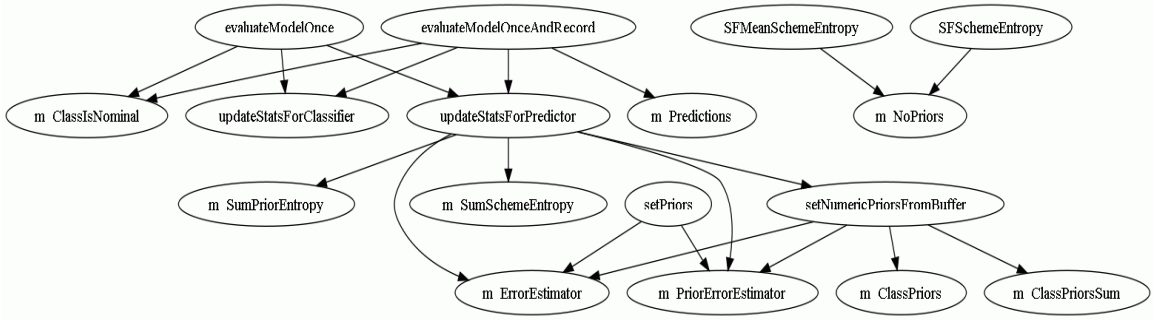
C3 – Impermanent patterns



C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

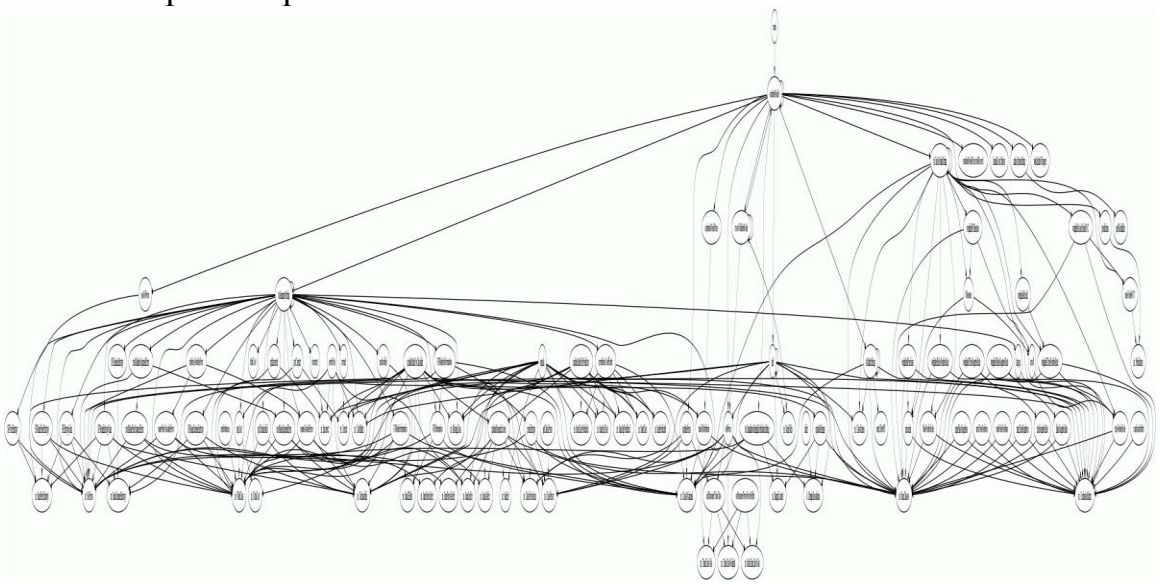


C6 – Reincarnated patterns

NONE FOUND

VERSION 5688 (Revision 9)

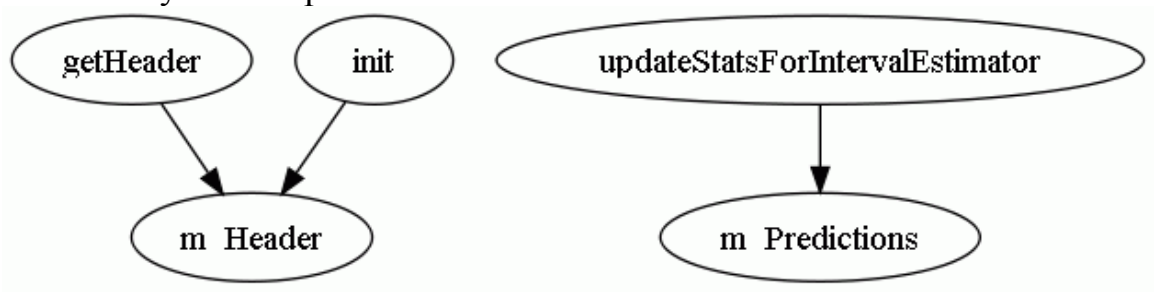
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 5714 (Revision 10)

C1 – Newly formed patterns



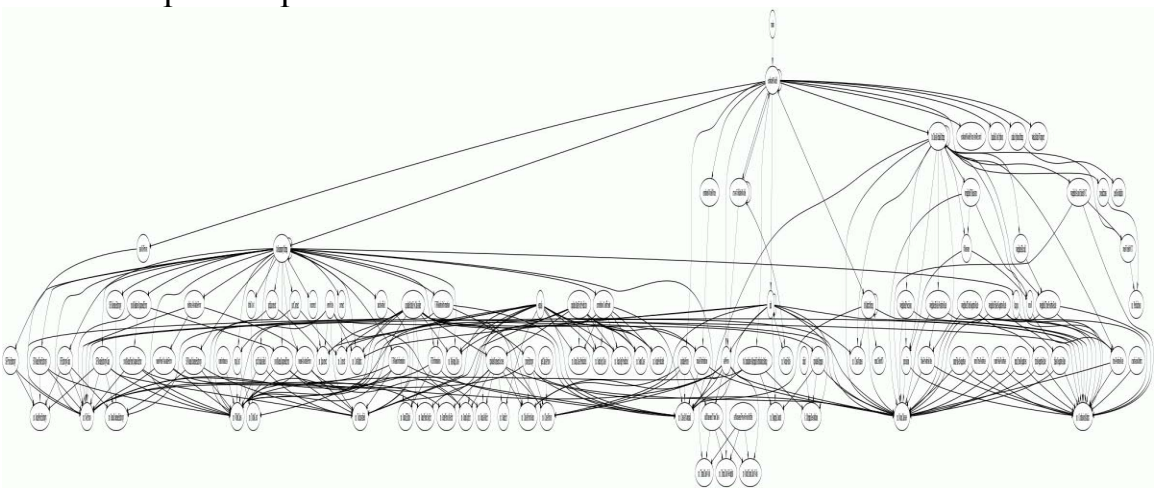
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

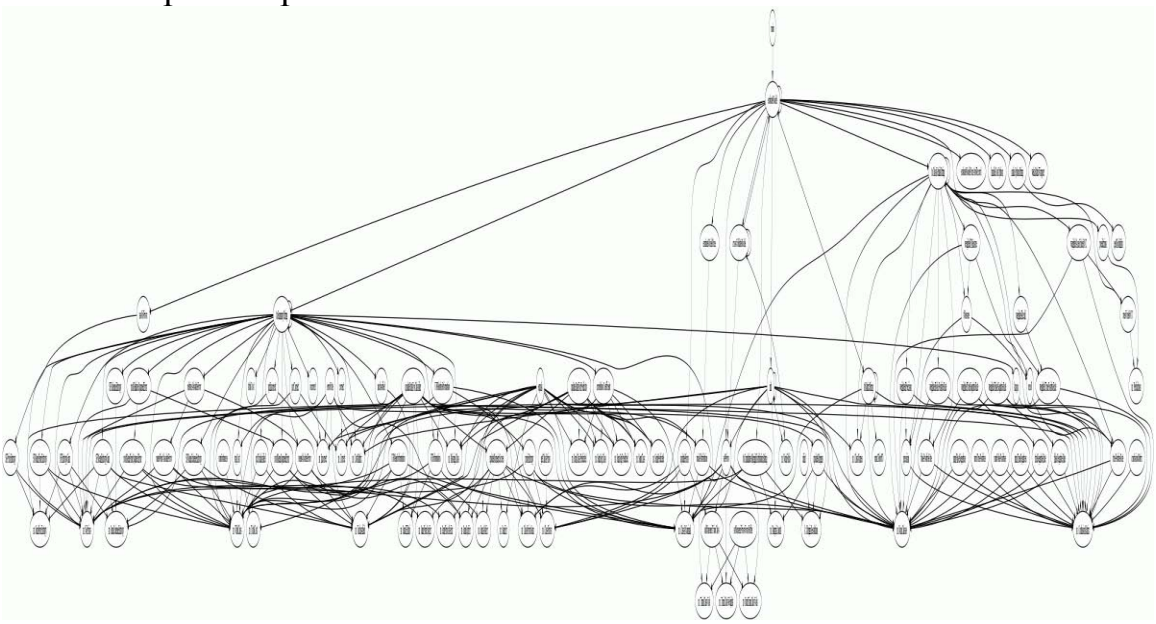
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 5928 (Revision 11)

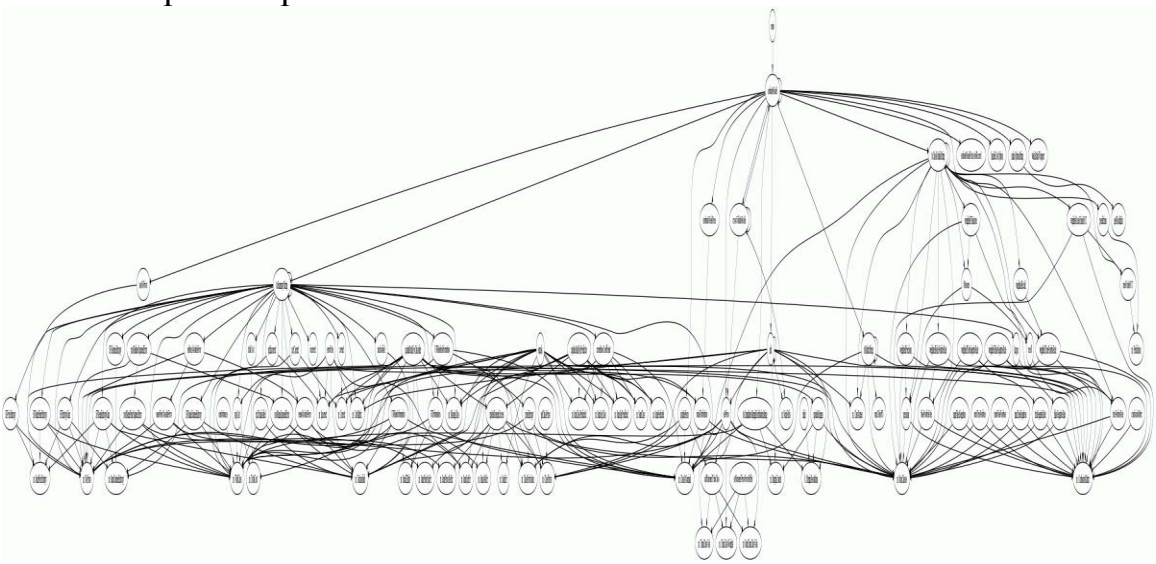
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 5987 (Revision 12)

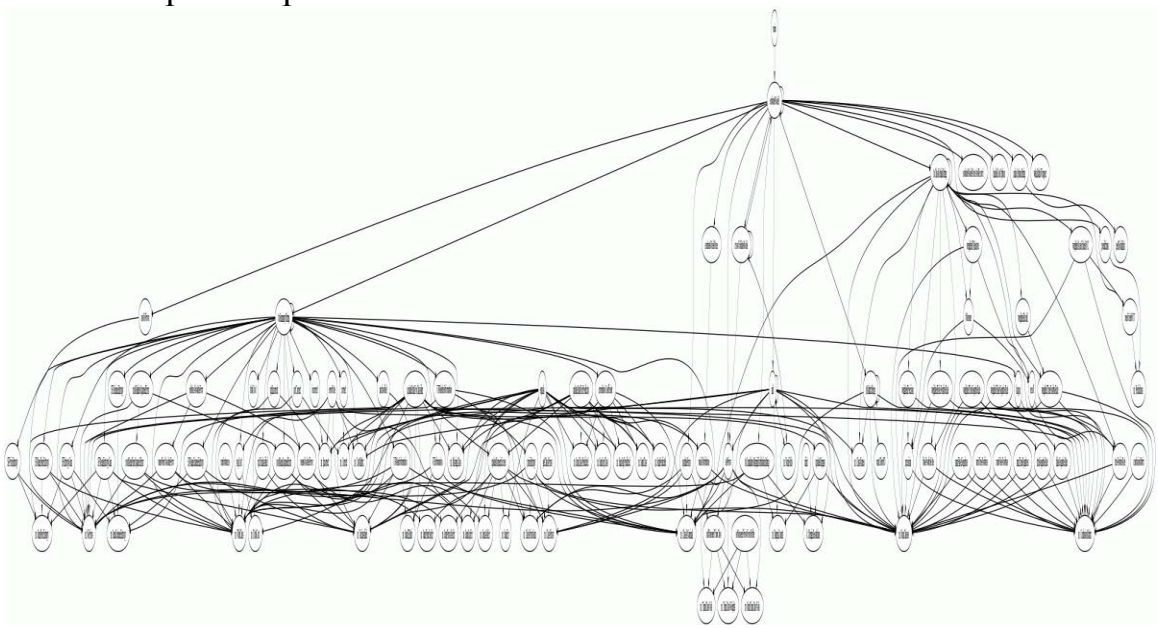
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 6041 (Revision 13)

C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

EXPERIMENT 3 – WEKA (GridSearch.java)

In this experiment, different versions of class GridSearch.java were mined for patterns.

Details about class GridSearch.java and its versions appear below.

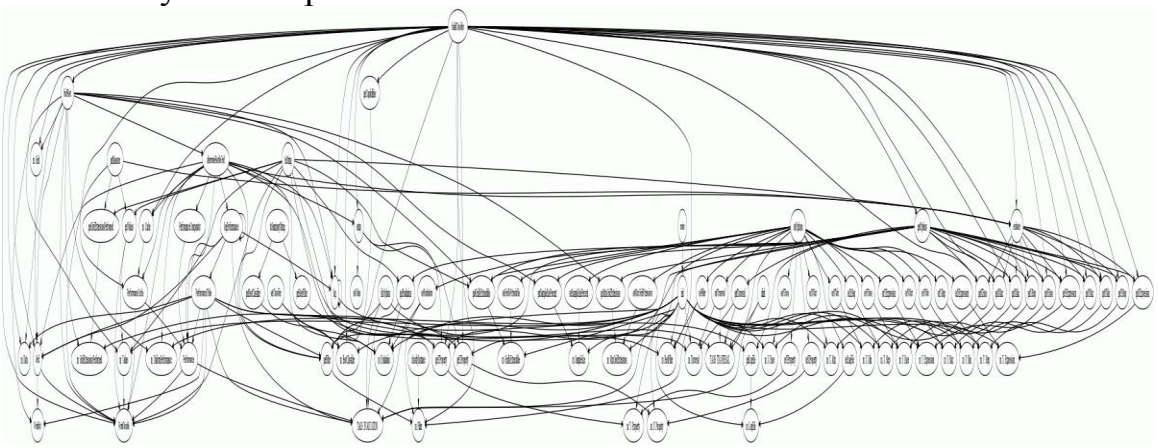
Name: GridSearch.java
Number of Revisions: 5
Location: /trunk/weka/src/main/java/weka/classifiers/meta
Subversion URL: https://svn.scms.waikato.ac.nz/svn/weka

Version Number	Lines of Code	Date and Time	Author	Message
4698	3487	2:39:21 PM, Thursday, November 13, 2008	mhall	Move out of top level
4828	3505	3:05:36 PM, Wednesday, December 03, 2008	fracpete	added the "kappa statistic" to the list of measures that one can evaluate on
5803	3507	3:48:37 PM, Tuesday, July 28, 2009	mhall	Enabled missing class values capability.
5928	3508	5:07:59 PM, Wednesday, September 02, 2009	eibe	Classifier.java is now an interface. Added new class AbstractClassifier.java.
6263	3866	12:58:29 PM, Sunday, February 07, 2010	fracpete	can take advantage of multi-core machines now

Patterns discovered in each version of the class are grouped into classes (for a definition of the six classes, please see Section 5.2).

VERSION 4828 (Revision 0)

C1 – Newly formed patterns



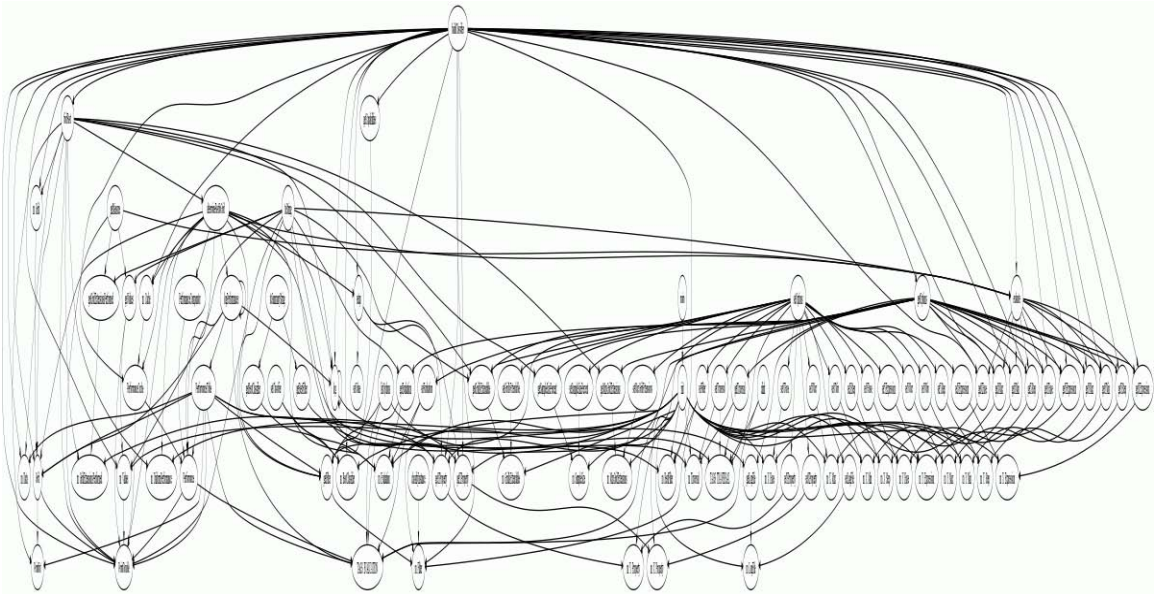
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

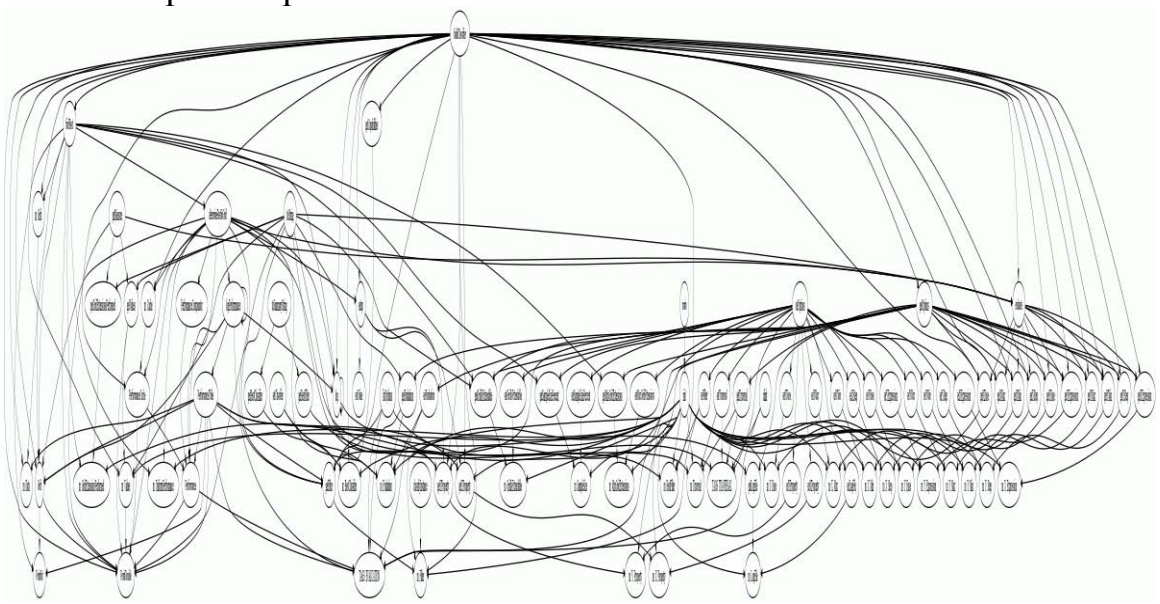
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 5803 (Revision 1)

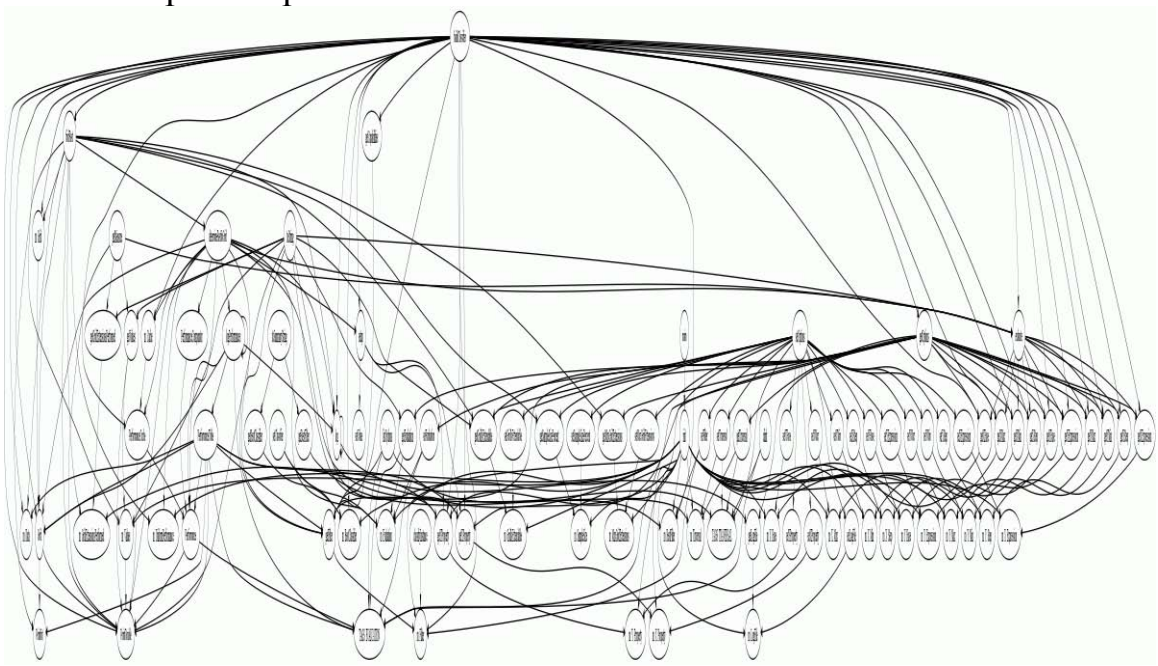
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 5928 (Revision 2)

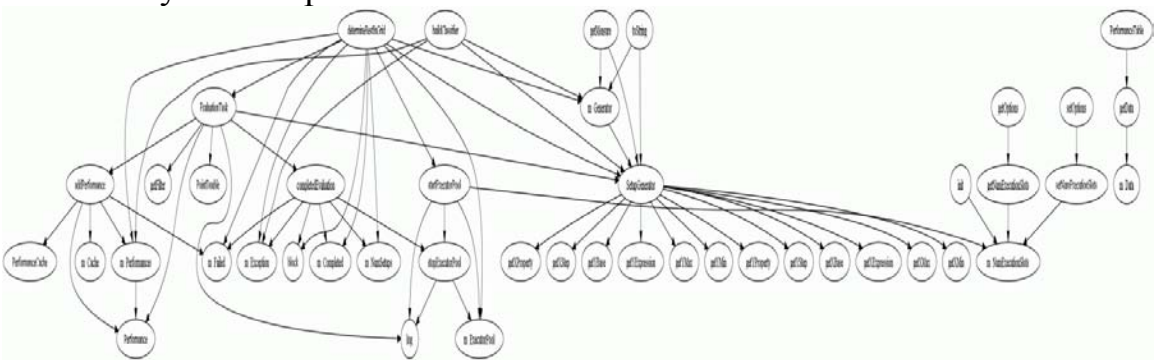
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 6263 (Revision 3)

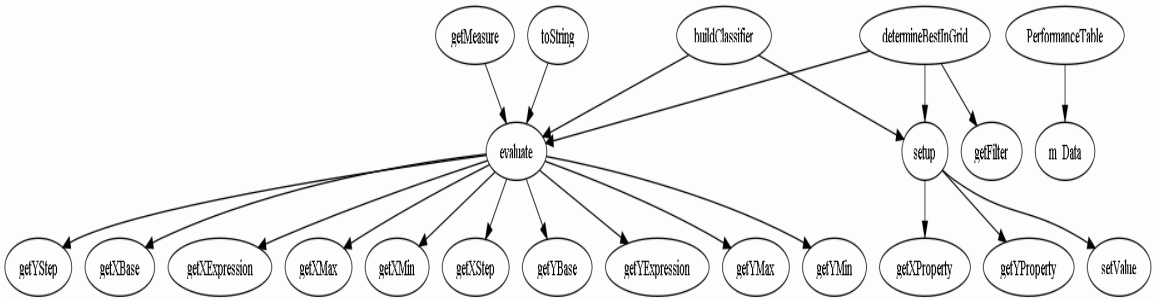
C1 – Newly formed patterns



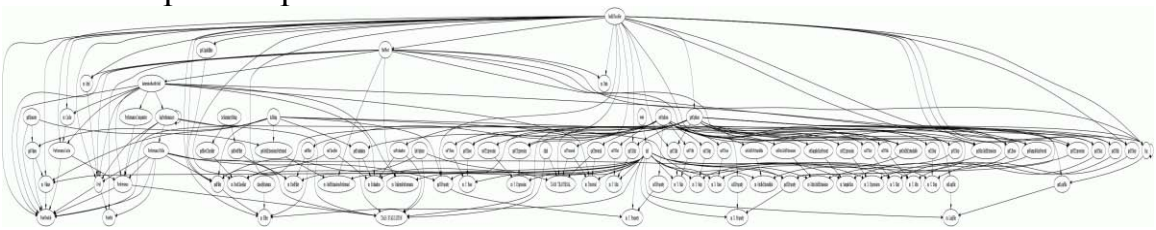
C2 – Patterns disappearing right after their creation

NONE FOUND

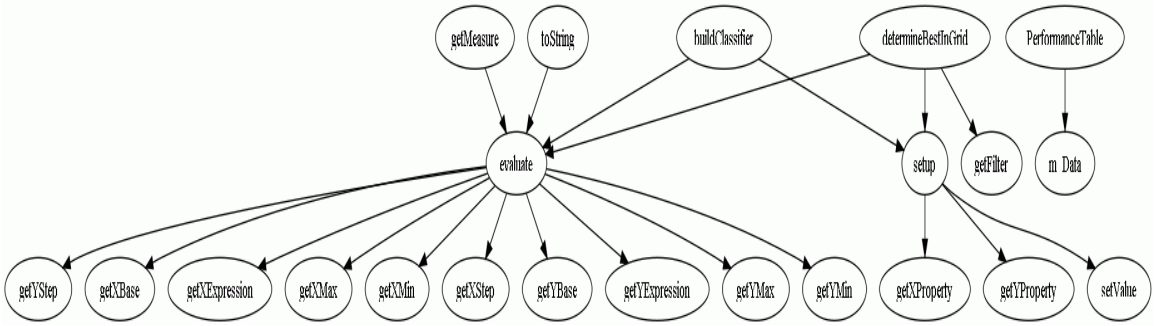
C3 – Impermanent patterns



C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns



C6 – Reincarnated patterns

NONE FOUND

EXPERIMENT 4 – DRDAConnThread.java

In this experiment, different versions of class DRDAConnThread.java were mined for patterns.

Details about class DRDAConnThread.java and its versions appear below.

Name: DRDAConnThread.java
Number of Revisions: 34
Location: /db/derby/code/trunk/java/drda/org/apache/derby/impl/drda
Subversion URL: https://svn.apache.org/repos/asf/db/derby/code/trunk

Version Number	Lines of Code	Date and Time	Author	Message
495543	8216	2:49:44 AM, Friday, January 12, 2007	kahatlen	DERBY-2121: Remove JDK 1.3 build dependency in network server
497748	8223	1:13:10 AM, Friday, January 19, 2007	bernt	DERBY-2166 Implement proper handling of SocketTimeoutException in DRDAConnThread. Submitted by Bernt M. Johnsen
515102	8222	4:27:05 AM, Tuesday, March 06, 2007	kristwaa	DERBY-2405: Remove @author tags from the source files. Patch contributed by Saurabh Vyas.
515563	8228	5:40:14 AM, Wednesday, March 07, 2007	kristwaa	DERBY-2347: Add code to support request and return of locators over DRDA. Patch contributed by Øystein Grøvlen.
515793	8228	2:01:37 PM, Wednesday, March 07, 2007	Fuzzylogic	DERBY-2400 (partial): Replace references to Cloudscape with Derby. This commit takes care of the rest of the separate source trees, except for engine, build and testing.
517131	8234	Date: 1:40:06 AM, Monday, March 12, 2007	kahatlen	Message: DERBY-2220: Uncommitted transactions executed through XAResource will hold locks after the application terminates (or crashes during the transaction). Abort the global transaction on a derby server when the

				network socket is closed.
534610	8270	1:45:17 PM, Wednesday, May 02, 2007	rhillegas	DERBY-2506: Committed Narayanan's PreparedCallable_DRDA_ v5.diff, adding some BLOB locator support.
534985	8277	1:39:18 PM, Thursday, May 03, 2007	kmarsden	DERBY-2381 ParameterMappingTest fails due to ArrayIndexOutOfBoundsException executing a procedure Formerly, the server would rely on the input parameter type information received from the client to determine the output parameter type. This patch changes the server to look at the parameter metadata to determine the drda type to send. It also enables the test ParameterMappingTest for client.
542925	8289	1:26:17 PM, Wednesday, May 30, 2007	rhillegas	DERBY-2695: Oystein's soft upgrade support for LOB locators.
545454	8289	1:22:47 AM, Friday, June 08, 2007	bernt	DERBY-2748 TimeSlice and Socket-Timeout bounds checking wrong
556589	8289	4:16:45 AM, Monday, July 16, 2007	kahatlen	Fixed javadoc.
557032	8291	12:21:22 PM, Tuesday, July 17, 2007	kmarsden	DERBY-2941 With 10.2, Closing a resultset after retrieving a large > 32665 bytes value with Network Server does not release locksPort to 10.4. Verified with 10.2 client running LargeDataLocksTest
562524	8321	10:26:38 AM, Friday, August 03, 2007	kmarsden	DERBY-2933 (partial) When network server disconnects due to an I/O Exception it does not always log the exception that caused the error Committing change for IOExceptions during writeScalarStream(). There also may be exceptions during disconnect of a session when the server shuts down and I left these

				unlogged.
570663	8321	12:02:03 AM, Wednesday, August 29, 2007	kahatlen	DERBY-3025: NPE when connecting to database with securityMechanism=8 Use an internal attribute name when passing security mechanism from the network server to the embedded driver. This prevents confusion if an embedded connection is established with securityMechanism specified (in which case the security mechanism should be ignored).
574870	8393	2:50:43 AM, Wednesday, September 12, 2007	oysteing	DERBY-3060: Network Server incorrectly assumes that all SQLExceptions with error code 08004 are caused by an authentication failure.Contributed by Jørgen Løland
581012	8402	9:49:37 AM, Monday, October 01, 2007	kmarsden	DERBY-3085 Fails to handle BLOB fields with a PreparedStatement with size >32750 bytes Store a reference to the stream for the streamed parameter in the DRDAStatement.paramSta te and then drain the stream after statement execution if needed. There is only one parameter ever streamed, so only one field needed to be added. I added a test for both BLOB's and CLOB's to Blob4ClobTest.java.
611272	8397	11:30:48 AM, Friday, January 11, 2008	djd	Cleanup unnecessary check in DRDAConnThread.writeS QLCINRD() and remove code that fetched the prepared statement but never used it.
613169	8404	6:19:59 AM, Friday, January 18, 2008	dyre	DERBY-3311: Client ResultSet.getHoldabilty will return incorrect value when the ResultSet is obtained from a procedure call Patch contributed by Daniel John Debrunner

				Patch file: derby_3311_diff.txt
614549	8412	6:36:10 AM, Wednesday, January 23, 2008	oysteing	DERBY-3184: Add error handling to SlaveDatabase. Contributed by Jorgen Loland
617186	8415	10:48:30 AM, Thursday, January 31, 2008	kmarsden	DERBY-3365 Network Server stores a duplicate entry in the lob hash map for every lob Change network server to use existing lob hash map entry instead of creating a second entry.
631593	8502	5:52:59 AM, Wednesday, February 27, 2008	dyre	DERBY-3192: Cache session data in the client driver Piggy-backs the current isolation level and the current schema onto messages going back to the client. The client caches this information so that it can be returned to a user (app) without an extra round-trip.See also http://wiki.apache.org/db-derby/Derby3192Writeup Patch file: derby-3192-mark2.v8.diff
631997	8513	7:20:49 AM, Thursday, February 28, 2008	bernt	DERBY-3435 Added some live data to the network server MBean
632413	8513	11:17:53 AM, Friday, February 29, 2008	djd	DERBY-3484 For JDBC 3.0 java.sql.Types constants use directly instead of through JDBC30Translation as JSR169 supports all the types
632456	8513	2:02:27 PM, Friday, February 29, 2008	djd	DERBY-3484 For JDBC 2.0/3.0 java.sql.ResultSet constants use directly instead of through JDBC[2,3]0Translation as JSR169 supports all the types
633011	8509	1:50:46 AM, Monday, March 03, 2008	dyre	DERBY-3192: Cache session data in the client driver Remove special handling of SYNCCTL in sanity-check ASSERT. Since the final version of the real patch

				piggy-backs changes caused by SYNCCTL there is no longer any need to omit the sanity check after SYNCCTL. Patch file: derby-3192-fup.v1.diff
642707	8506	4:09:05 AM, Sunday, March 30, 2008	kristwaa	DERBY-3576: Merge EngineBlob and EngineClob into a single interface. Merged the two interfaces and added the method 'free'. There was no need to sepearate between a Blob and a Clob where the interface was used. If such a need arises, one should consider adding new interfaces extending EngineLOB. Patch file: derby-3576-1b-enginelob_interface.diff
666088	8586	5:46:11 AM, Tuesday, June 10, 2008	kristwaa	DERBY-3596: Creation of logical connections from a pooled connection causes resource leak on the server. Exposed method 'resetFromPool' through EngineConnection. The network server now detects when a client is requesting new logical connectio ns. This triggers some special logic, where the physical connection on the serve r side is kept and reset instead of being closed and opened again (this caused resources to leak earlier). The special logic must *not* be triggered for XA connections, as the XA code is already well-behaved. Patch file: derby-3596-5a-complex_skip_creds.diff
674354	8586	3:04:55 PM, Sunday, July 06, 2008	kmarsden	DERBY-3706 NetworkServer console messages should print a time stamp Contributed by Suran Jayathilaka

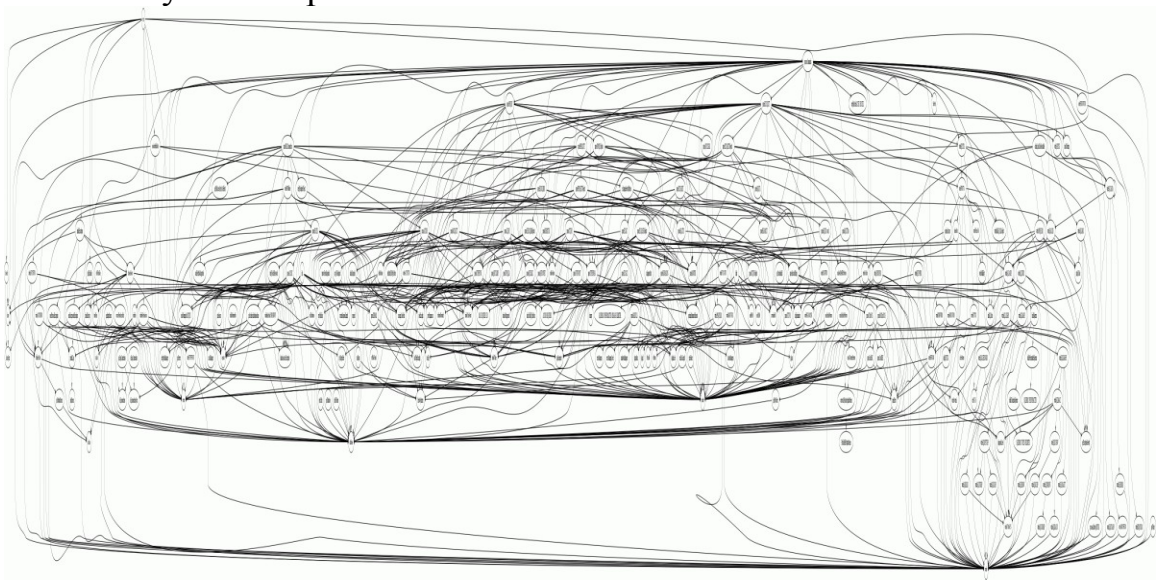
700948	8601	3:07:48 PM, Wednesday, October 01, 2008	myrnavl	DERBY-3390; preventing ClassCastException and disconnect on SQLException thrown from a user function
701199	8601	10:53:43 AM, Thursday, October 02, 2008	myrnavl	DERBY-3390; adjust javadoc after fix
703170	8599	7:16:32 AM, Thursday, October 09, 2008	myrnavl	DERBY-1411 - remove never-thrown SQLState NO_SUCH_DATABASE and check for it
734190	8610	9:21:57 AM, Tuesday, January 13, 2009	kmarsden	DERBY-4004 Remove required RDBNAM from ACCSEC. Use SECCHK RDBNAM if none is provided on ACCSEC
899733	8697	9:54:20 AM, Friday, January 15, 2010	rhillegas	DERBY-4491: Correct the network metadata for UDTs and make it possible to pass UDT values across the network.
901219	8697	6:42:57 AM, Wednesday, January 20, 2010	rhillegas	DERBY-4491: Fix javadoc error introduced by subversion revision 899733.
924746	8699	5:27:32 AM, Thursday, March 18, 2010	kahatlen	DERBY-4483: Provide a way to change the hash algorithm used by BUILTIN authentication Added more comments about the incompatibility between the configurable hash scheme and strong password substitution. Changed a symbol that still referred to the SHA-1 based authentication scheme as the new scheme.

Builds for versions from year 2006 and before failed. Therefore, class versions from January 2007 and onwards could be mined for patterns.

Patterns discovered in each version of the class are grouped into classes (for a definition of the six classes, please see Section 5.2).

VERSION 495543 (Revision 0)

C1 – Newly formed patterns



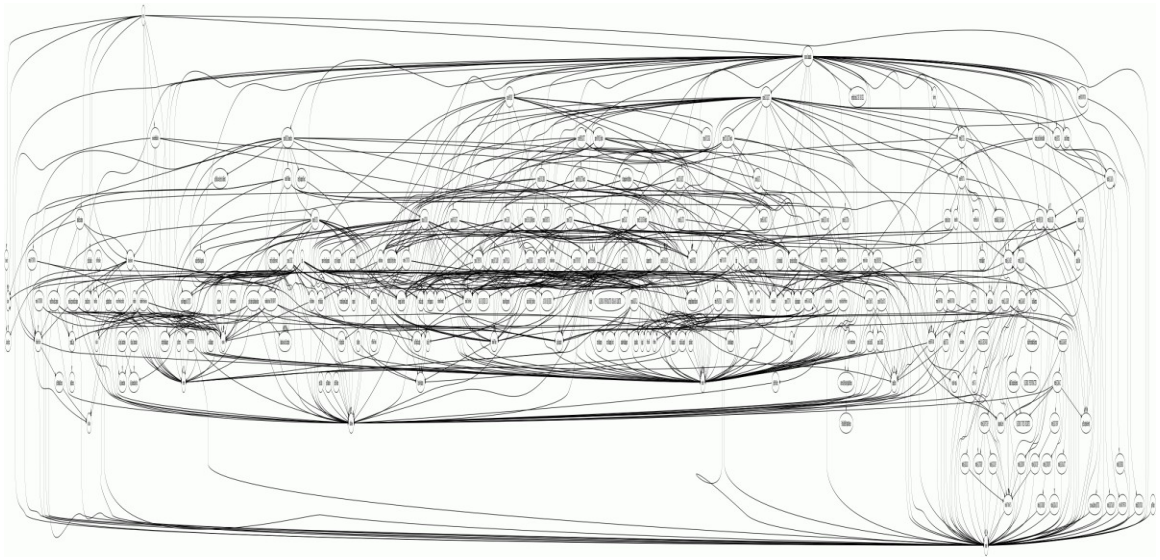
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

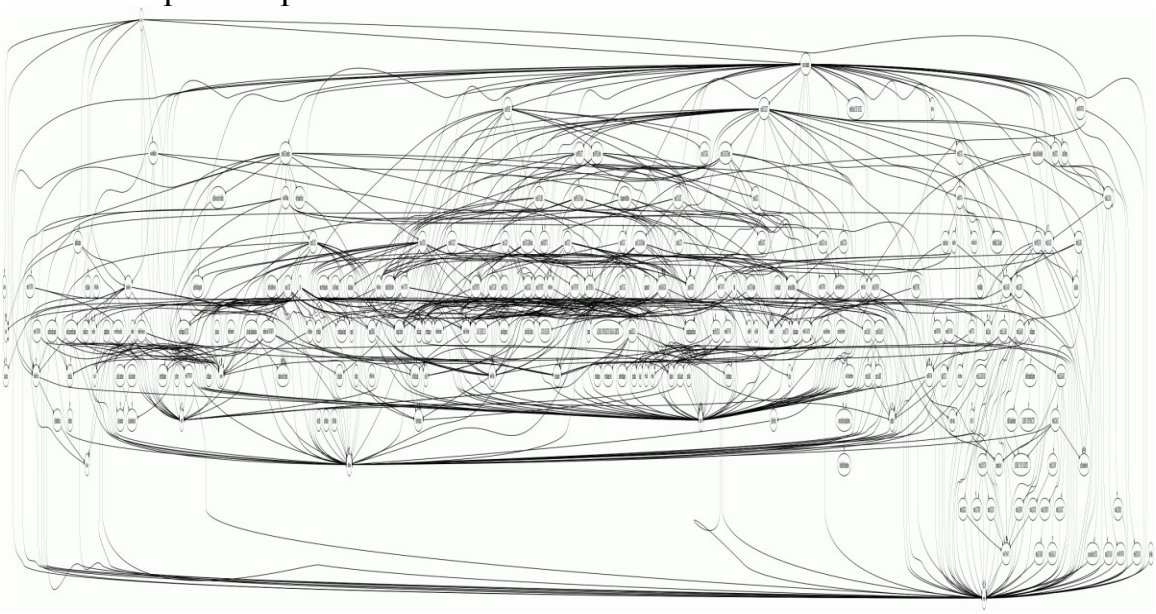
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 497748 (Revision 1)

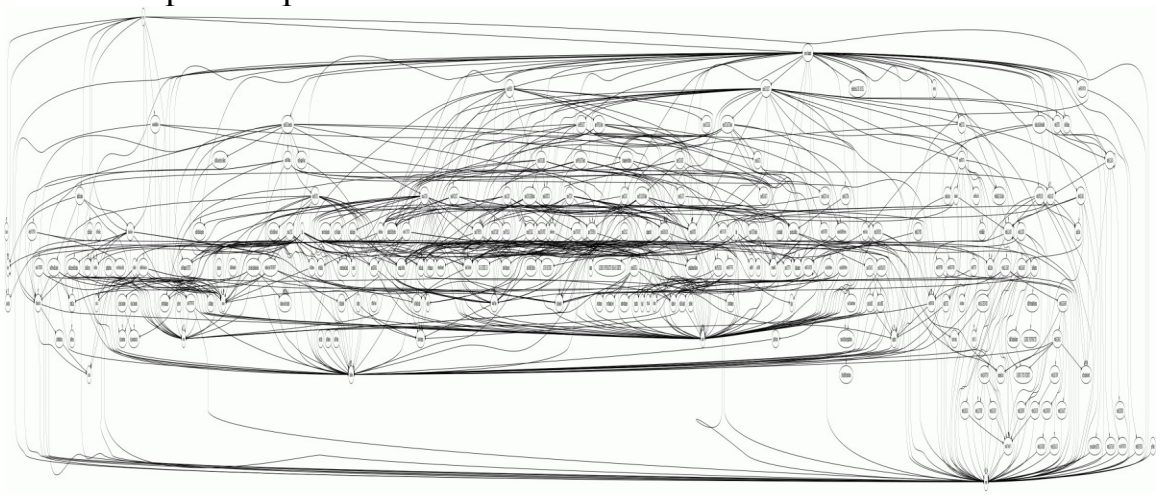
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 515102 (Revision 2)

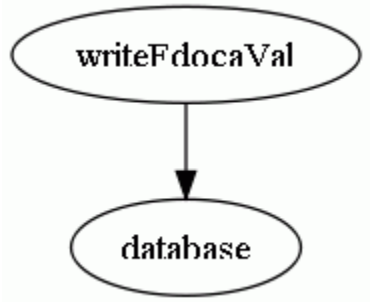
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 515563 (Revision 3)

C1 – Newly formed patterns



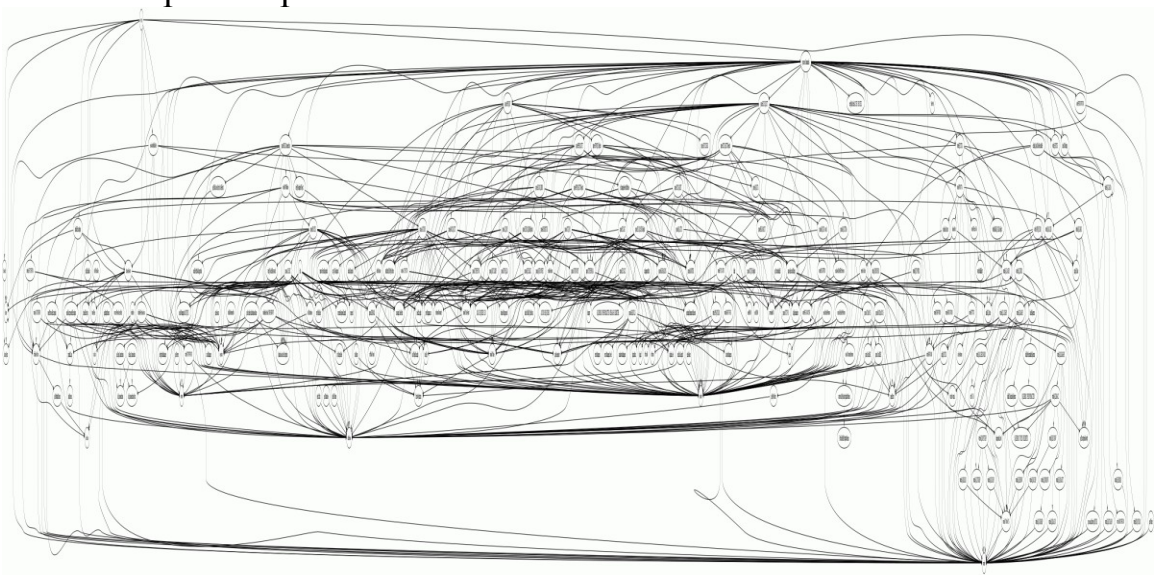
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

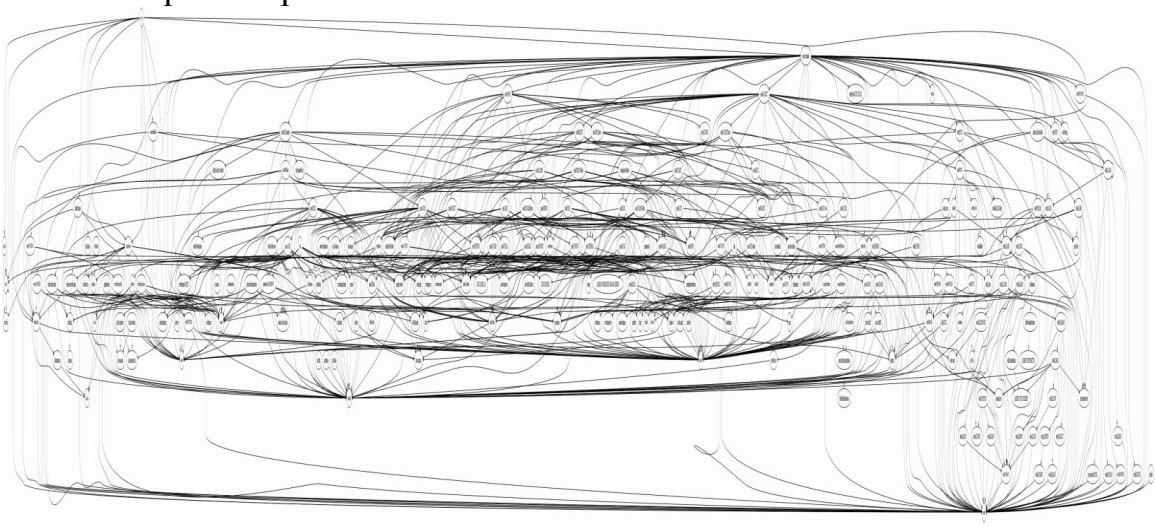
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 515793 (Revision 4)

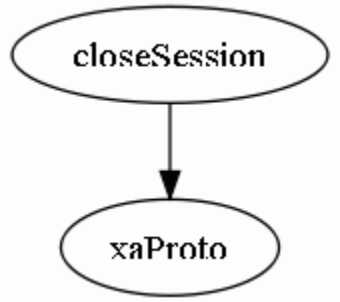
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 517131 (Revision 5)

C1 – Newly formed patterns



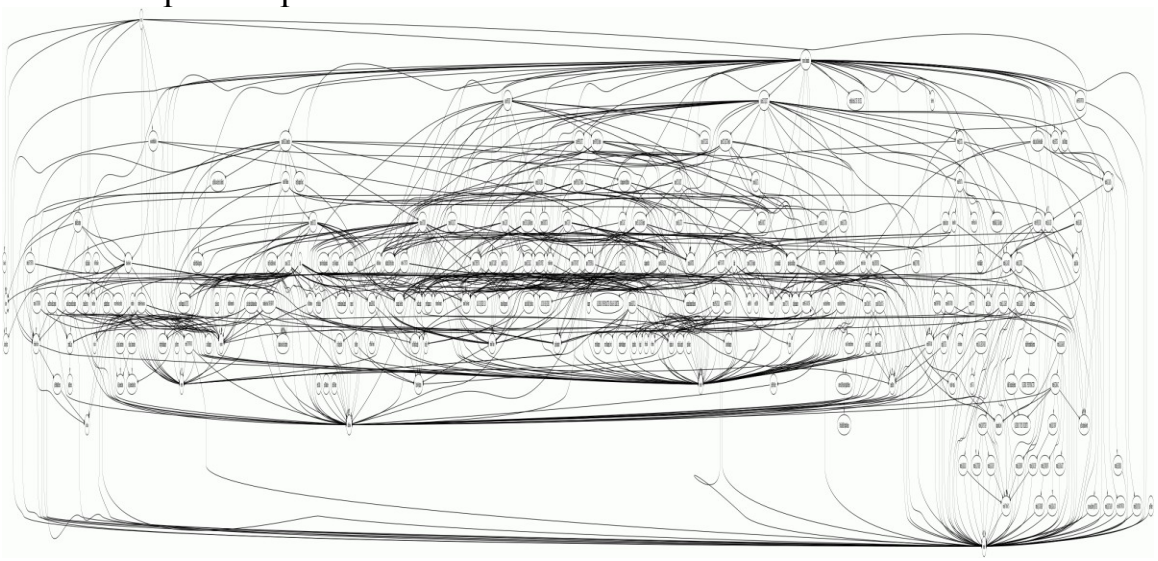
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

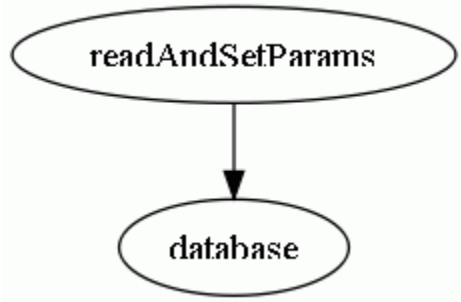
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 534610 (Revision 6)

C1 – Newly formed patterns



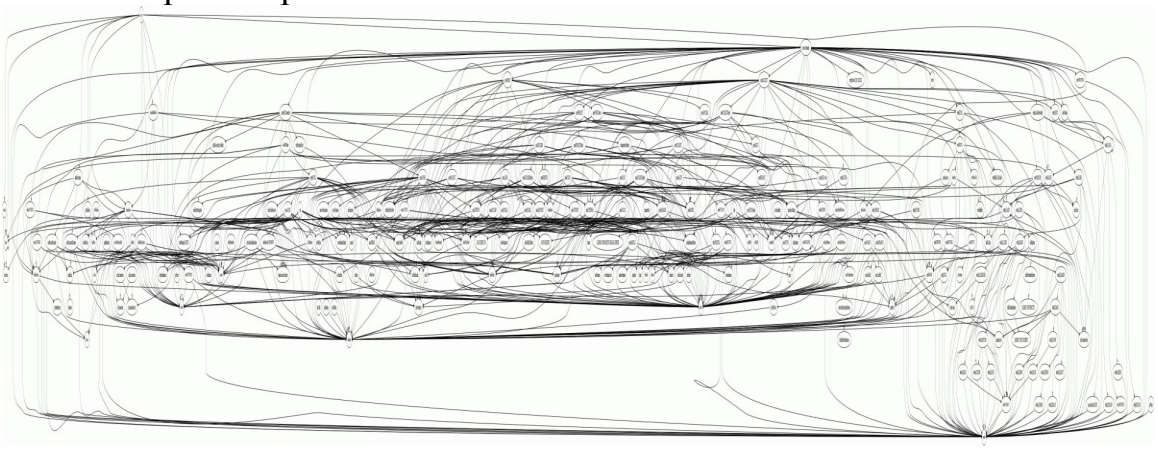
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

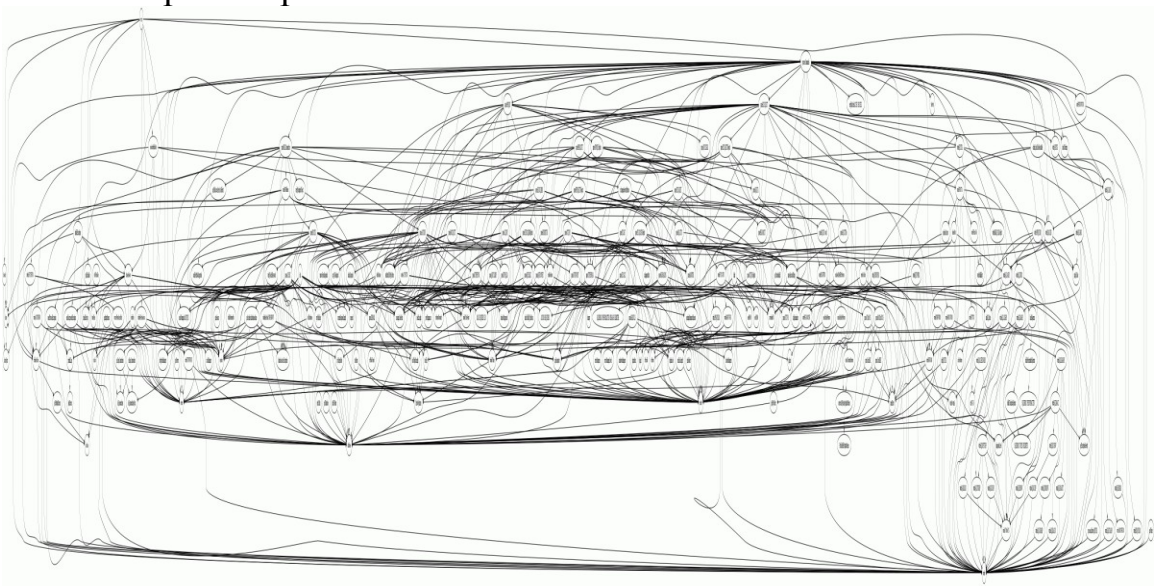
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 534985 (Revision 7)

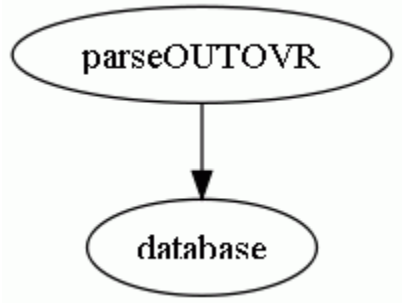
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 542925 (Revision 8)

C1 – Newly formed patterns



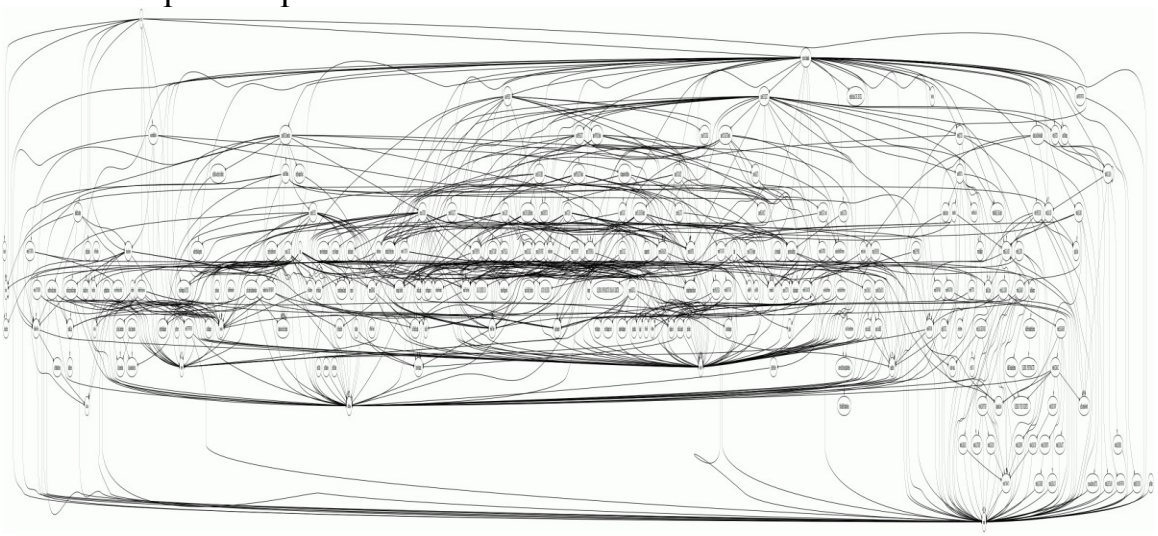
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

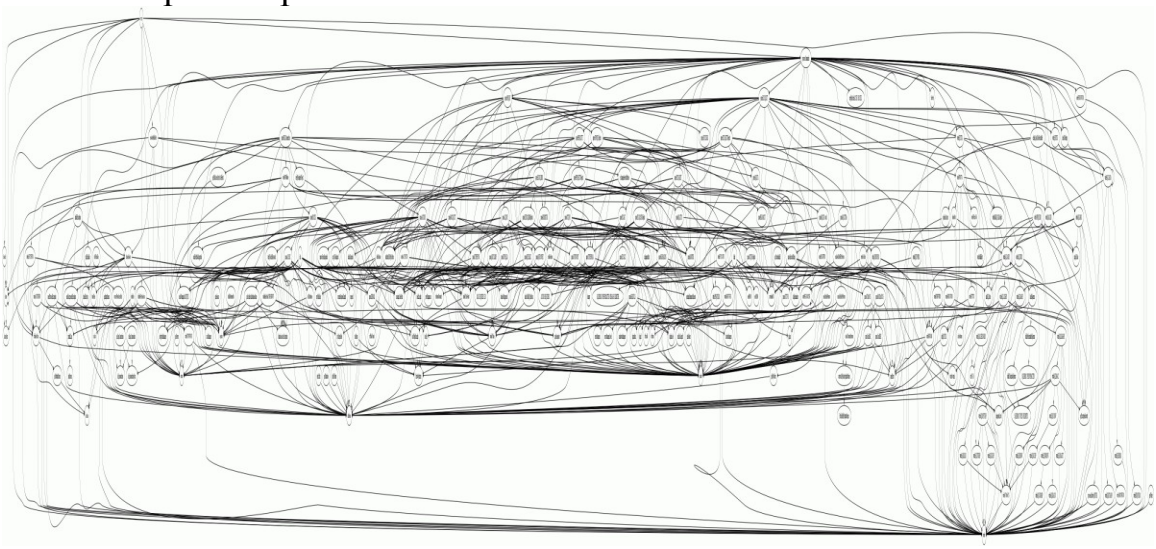
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 545454 (Revision 9)

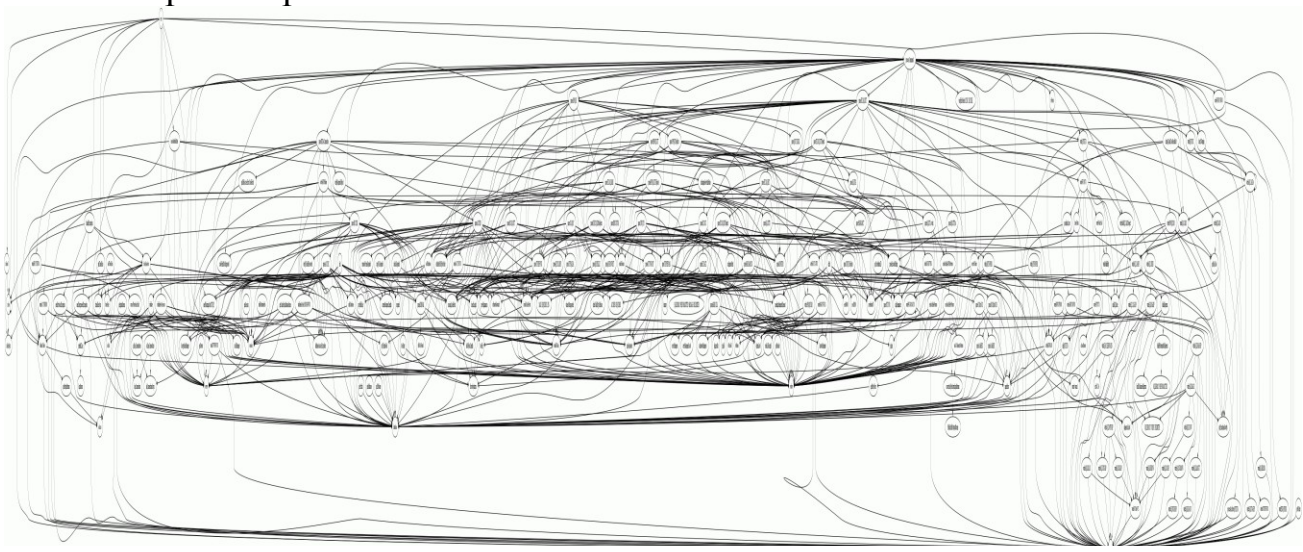
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 556589 (Revision 10)

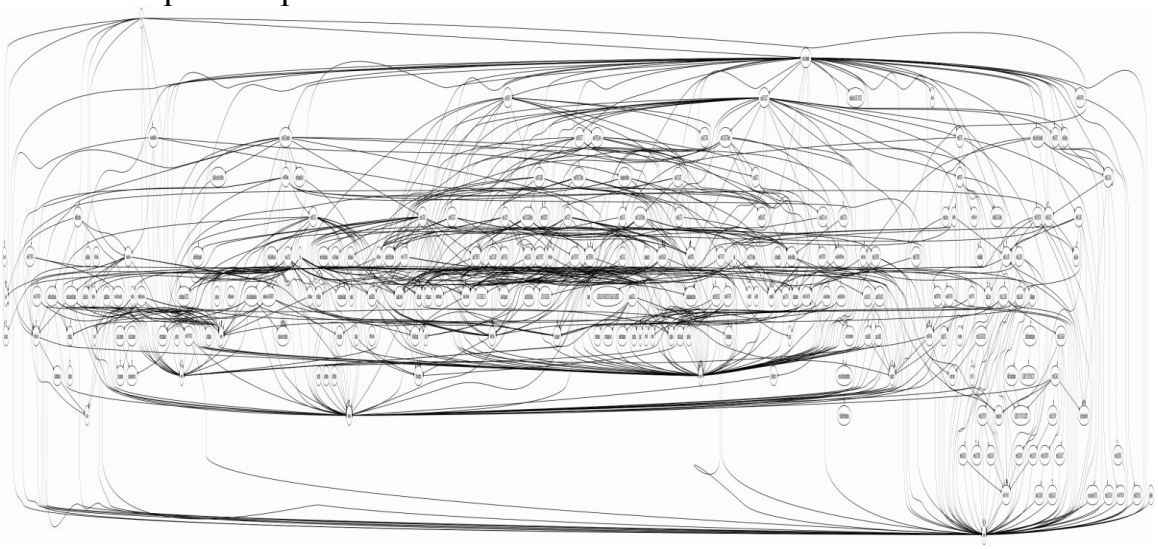
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 557032 (Revision 11)

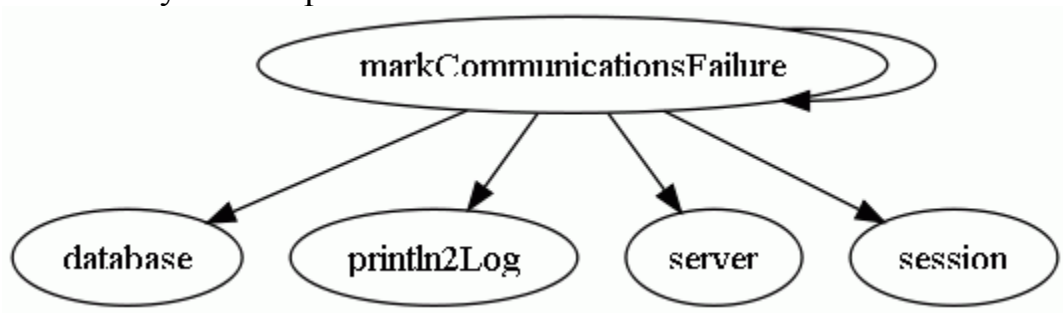
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 562524 (Revision 12)

C1 – Newly formed patterns



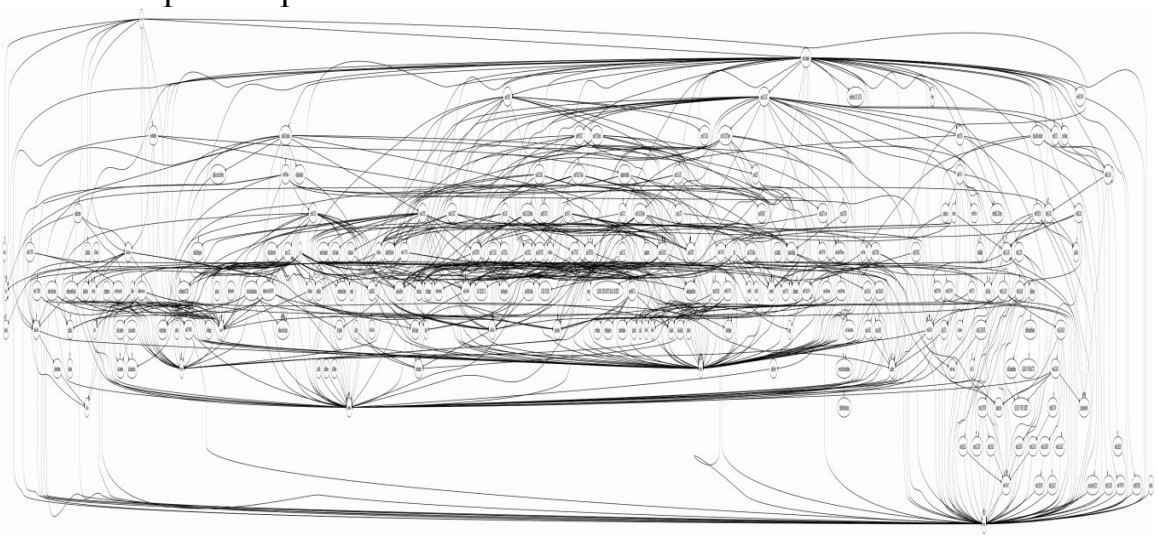
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

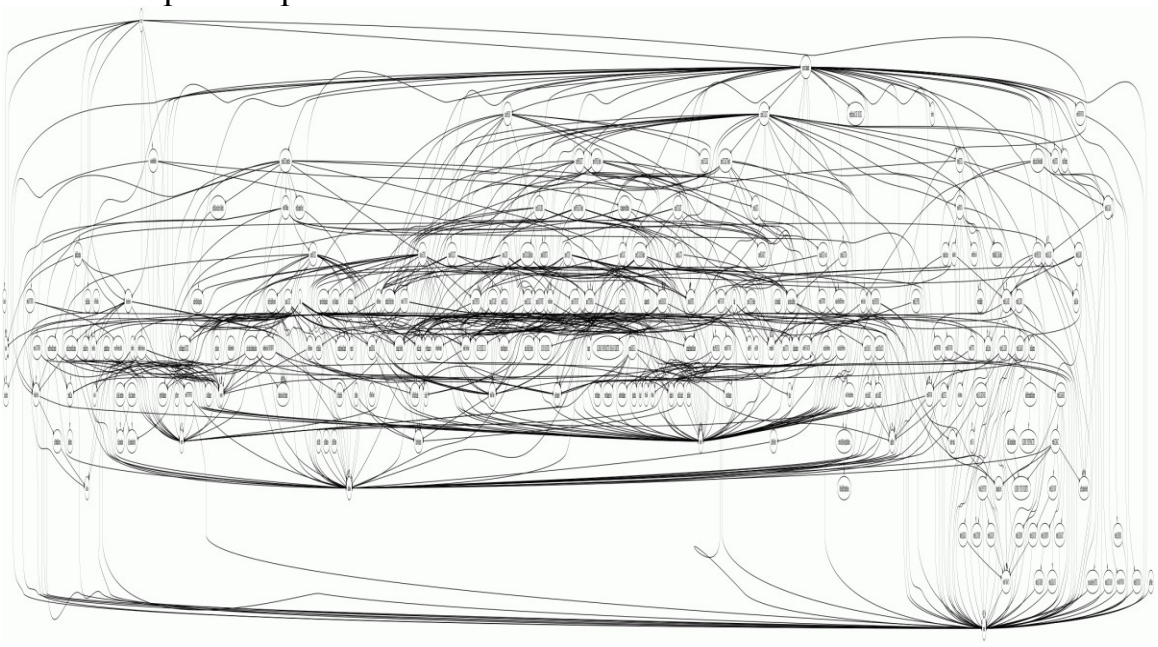
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 570663 (Revision 13)

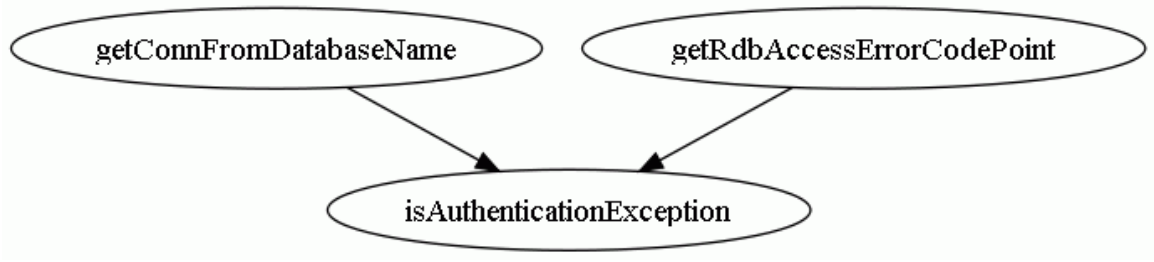
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 574870 (Revision 14)

C1 – Newly formed patterns



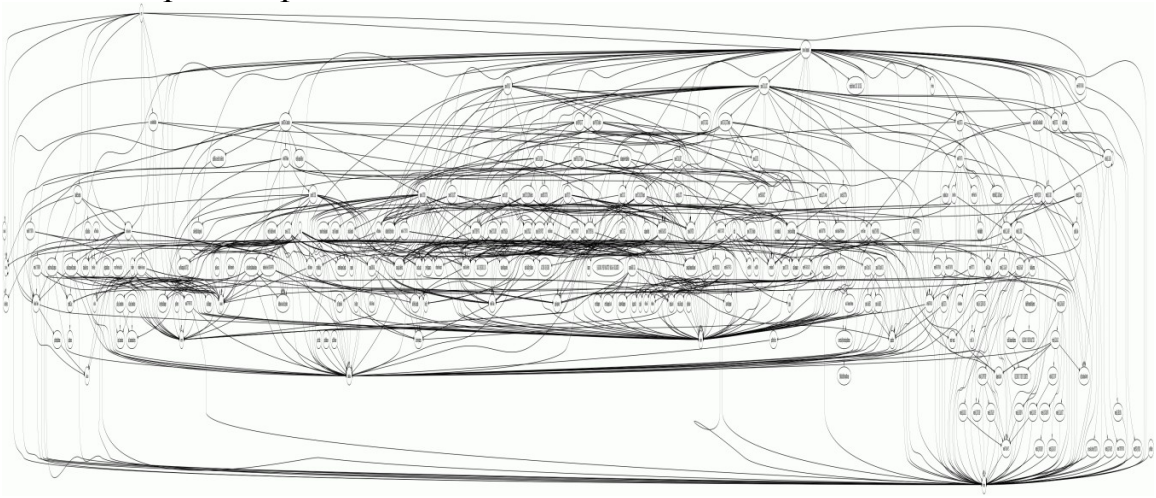
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

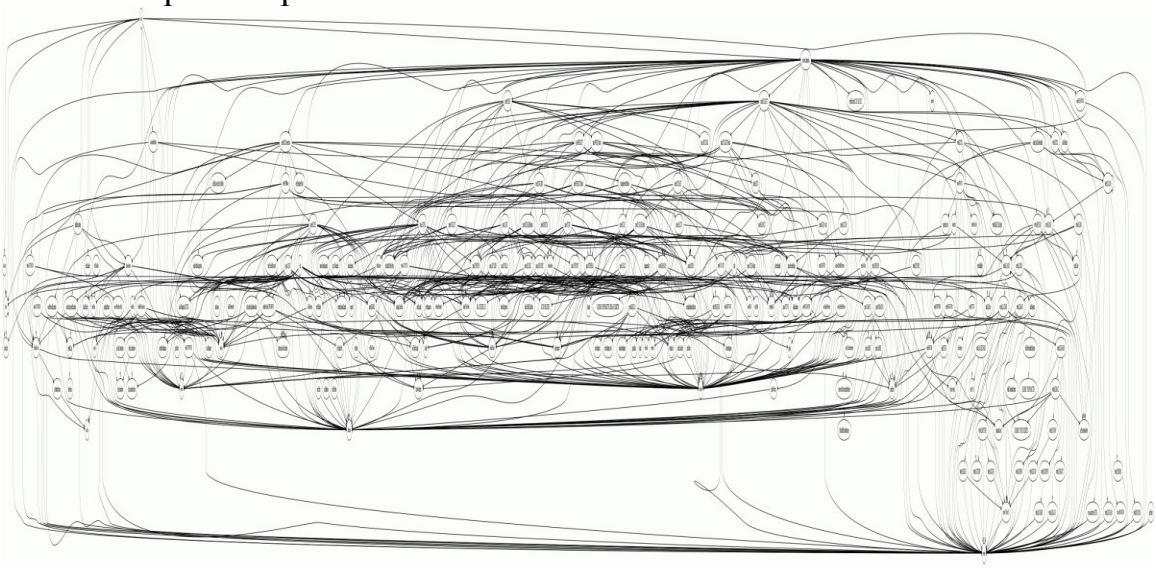
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 581012 (Revision 15)

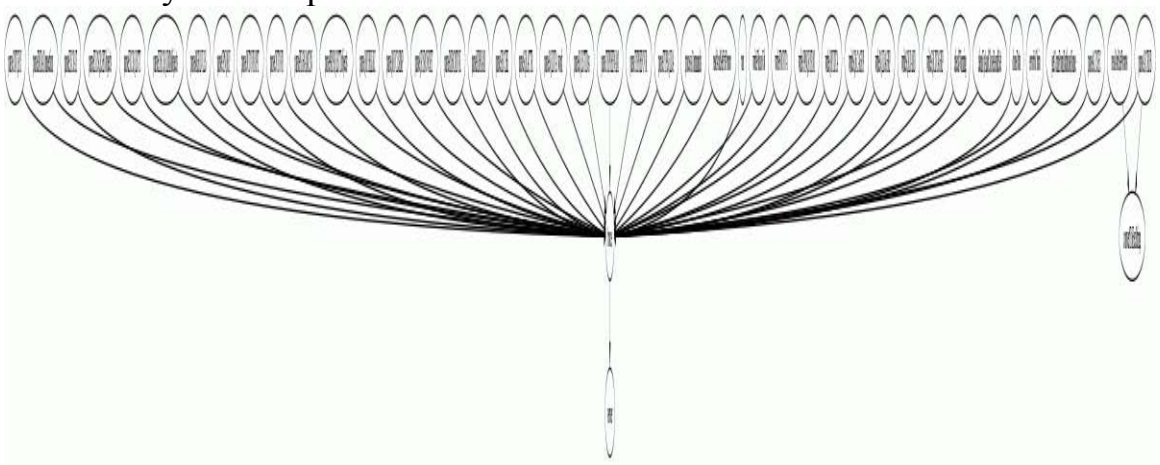
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 611272 (Revision 16)

C1 – Newly formed patterns



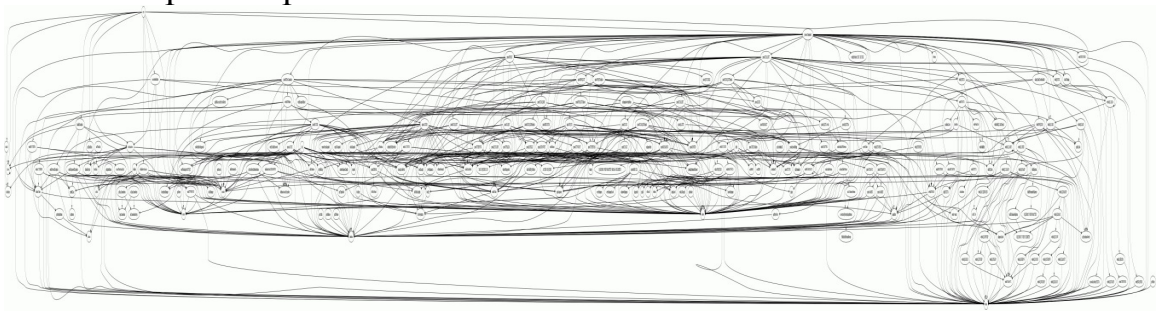
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

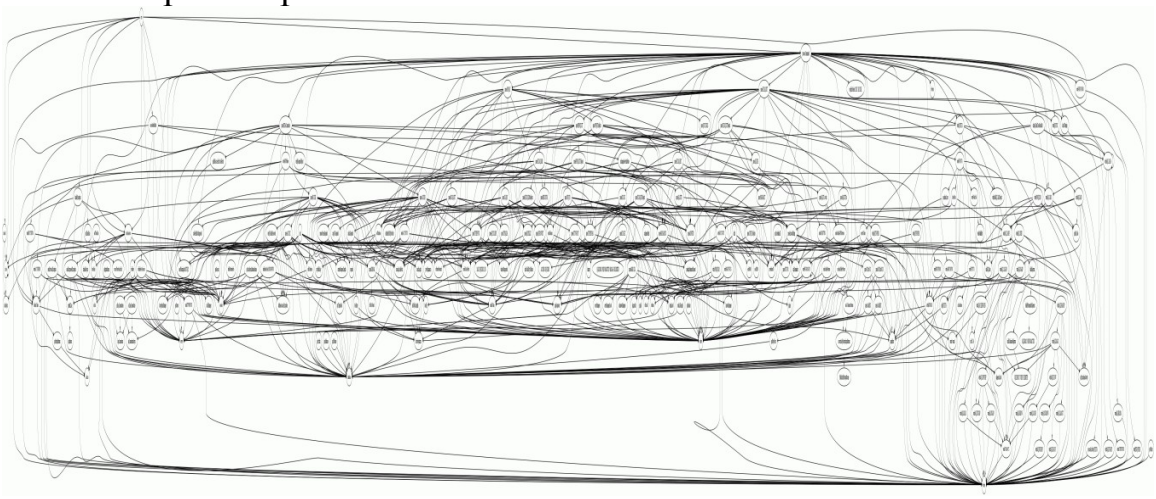
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 613169 (Revision 17)

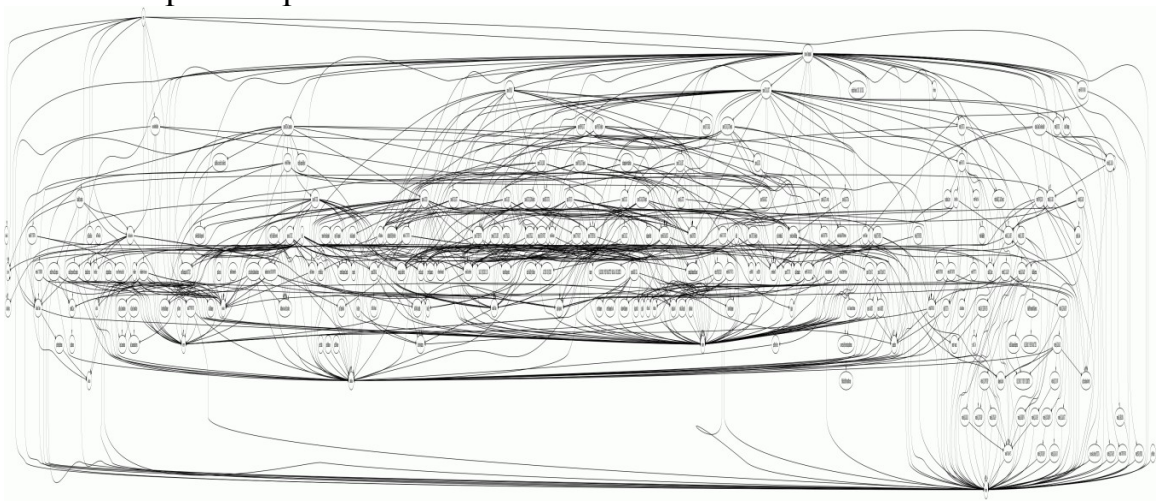
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 614549 (Revision 18)

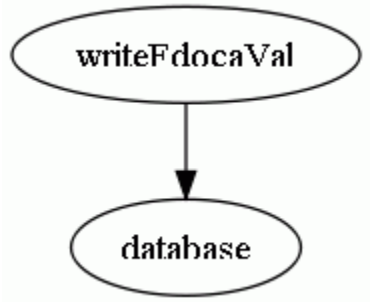
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



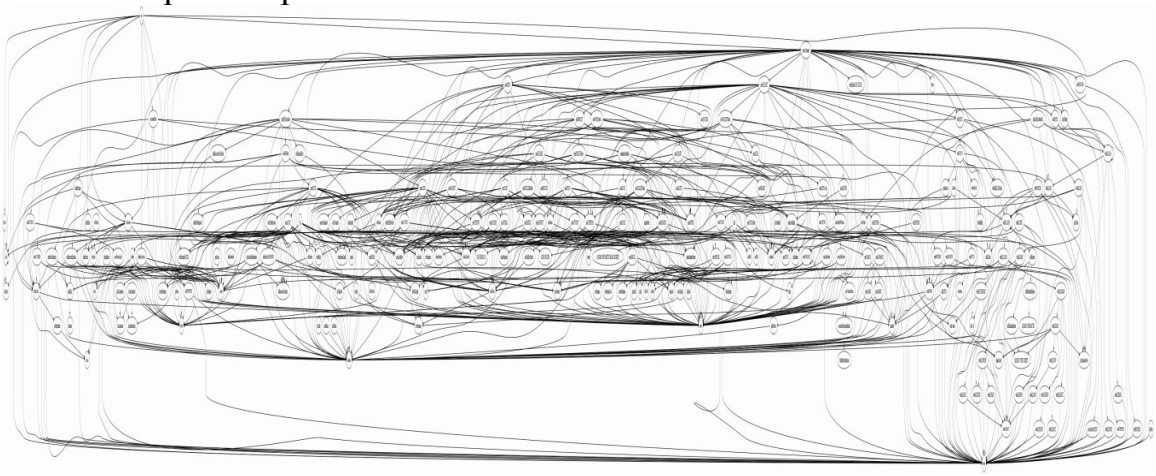
C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 617186 (Revision 19)

C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns



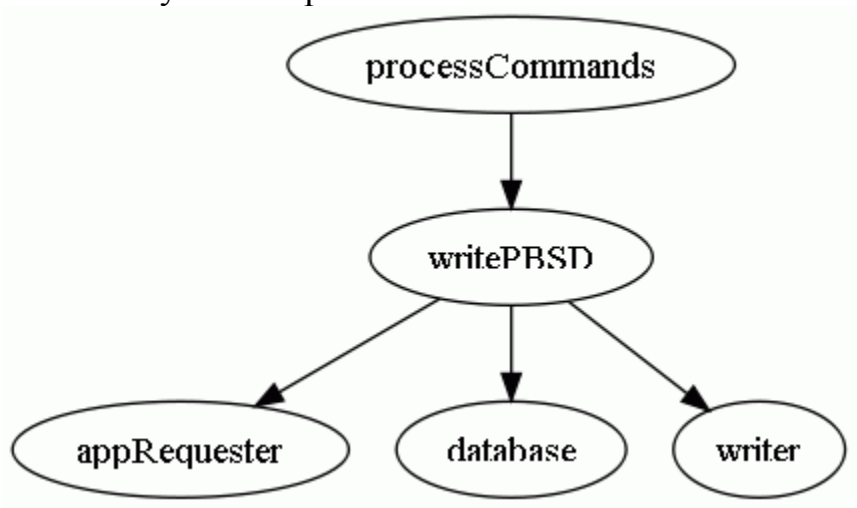
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 631593 (Revision 20)

C1 – Newly formed patterns



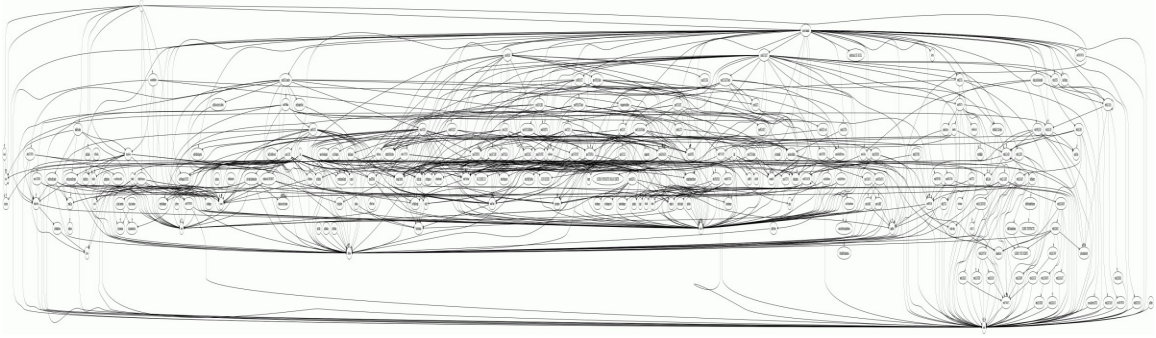
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

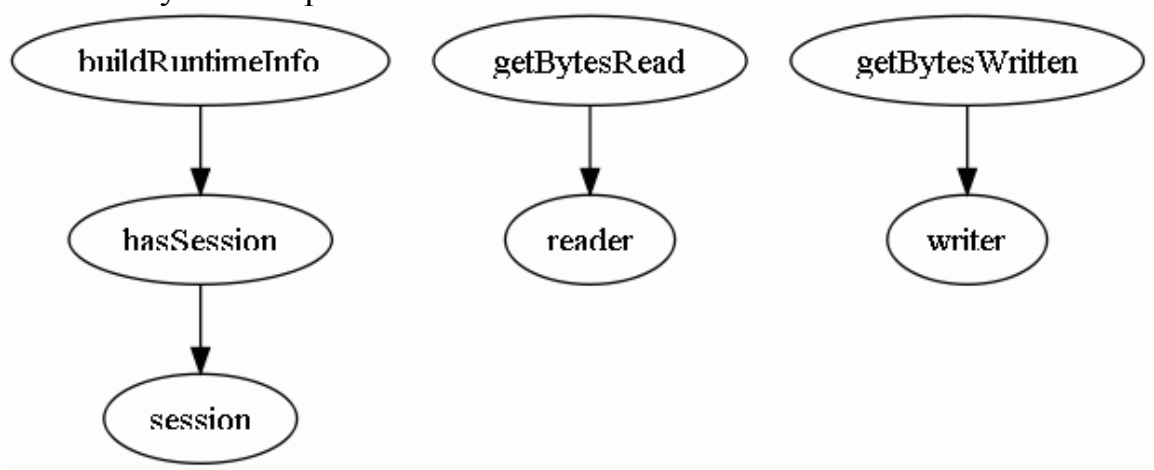
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 631997 (Revision 21)

C1 – Newly formed patterns



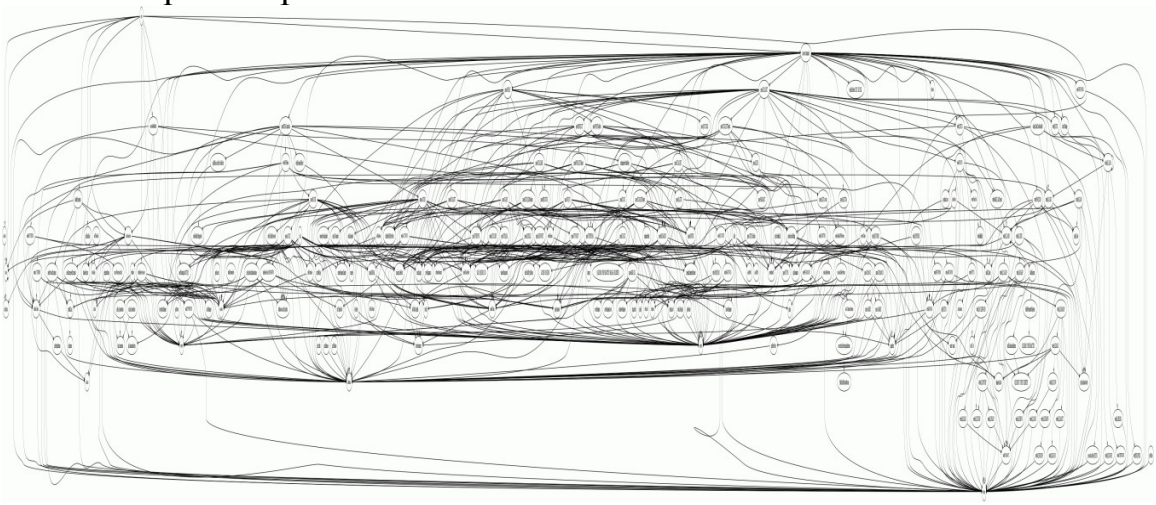
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

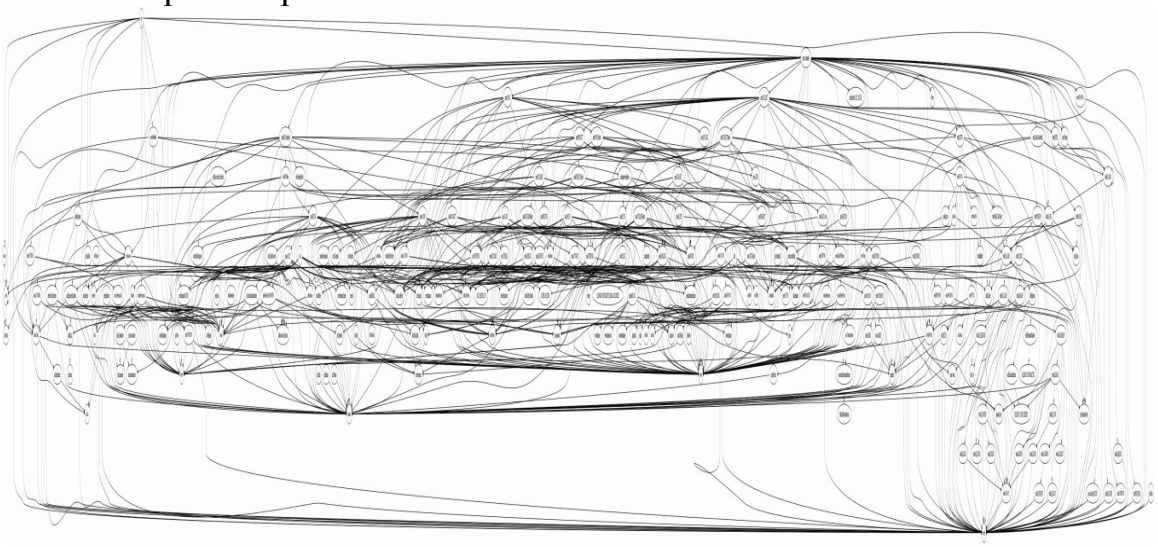
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 632413 (Revision 22)

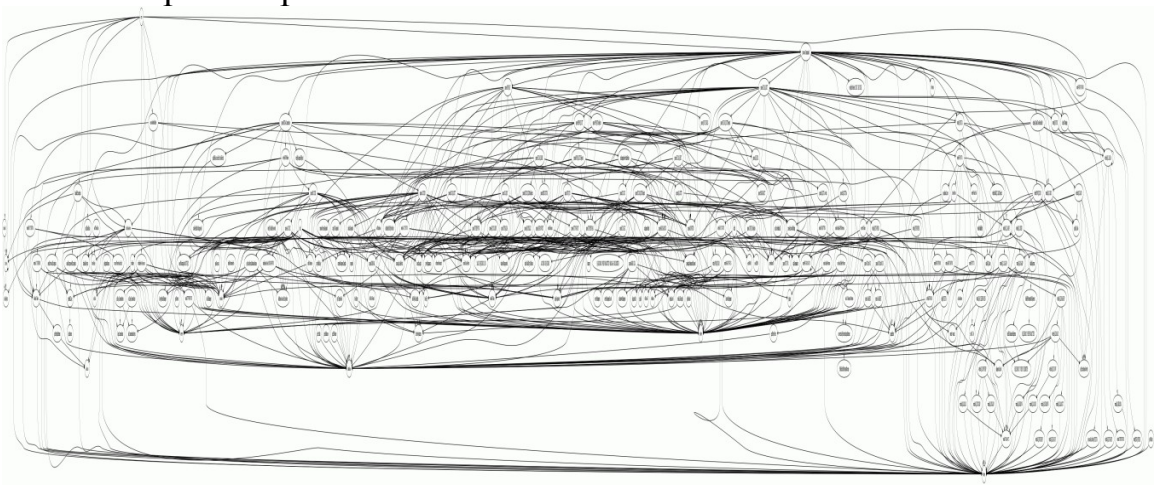
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 632456 (Revision 23)

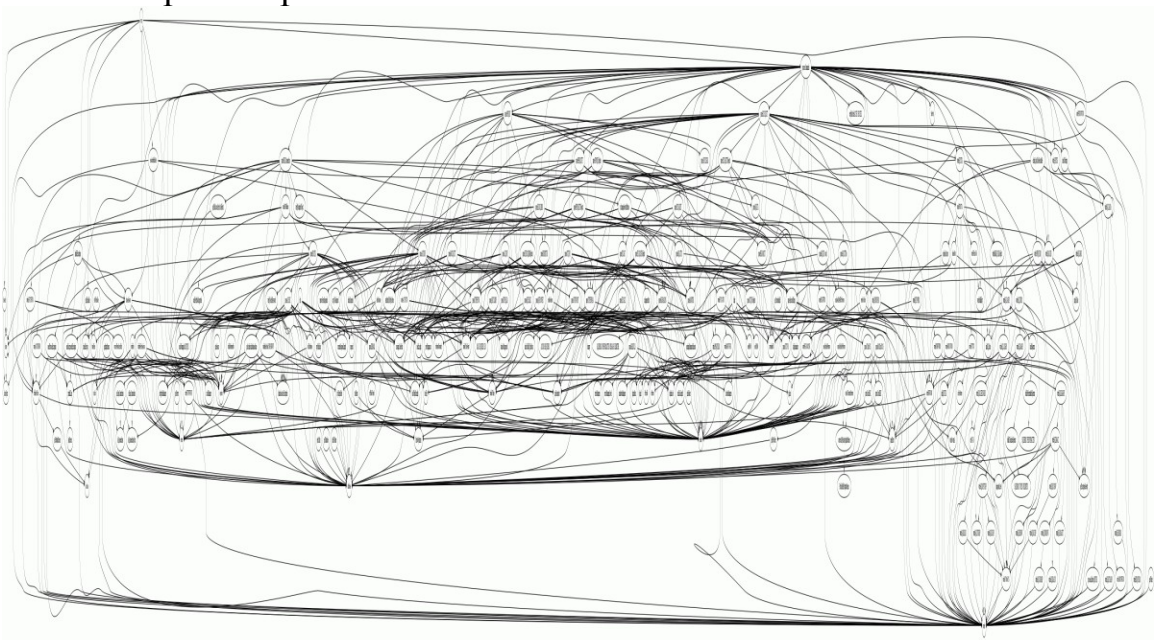
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 633011 (Revision 24)

C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

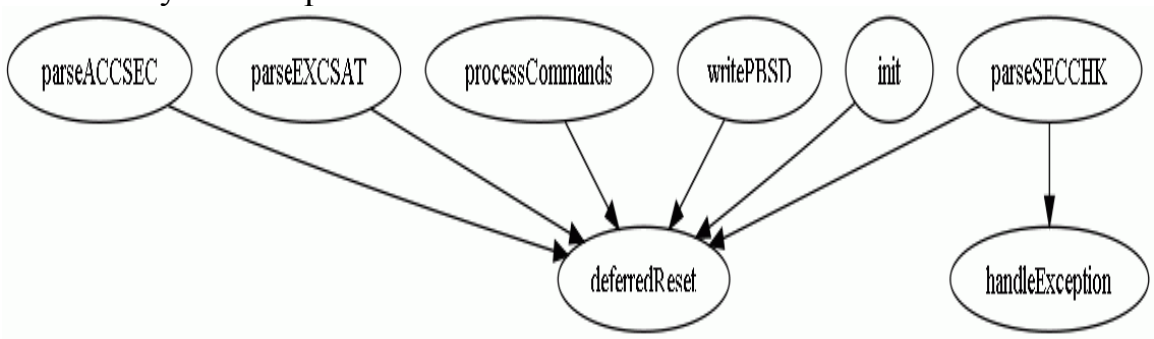
VERSION 642707 (Revision 25)

C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns

C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 666088 (Revision 26)

C1 – Newly formed patterns



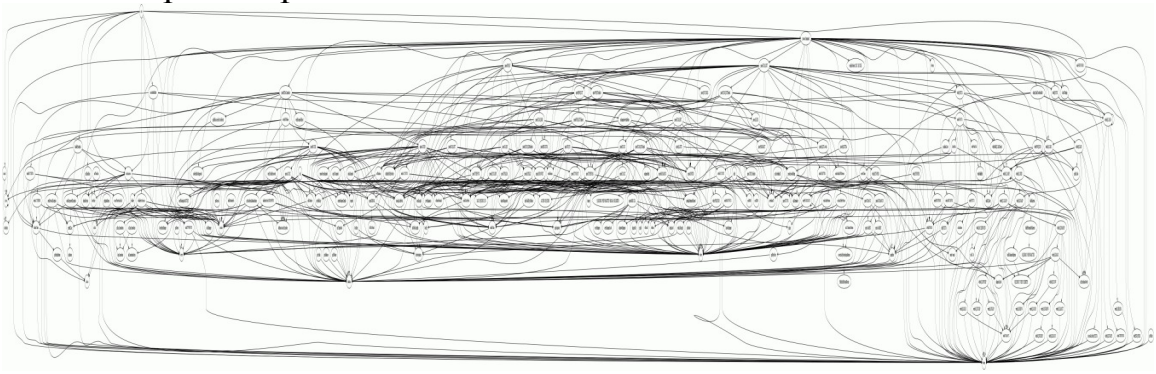
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

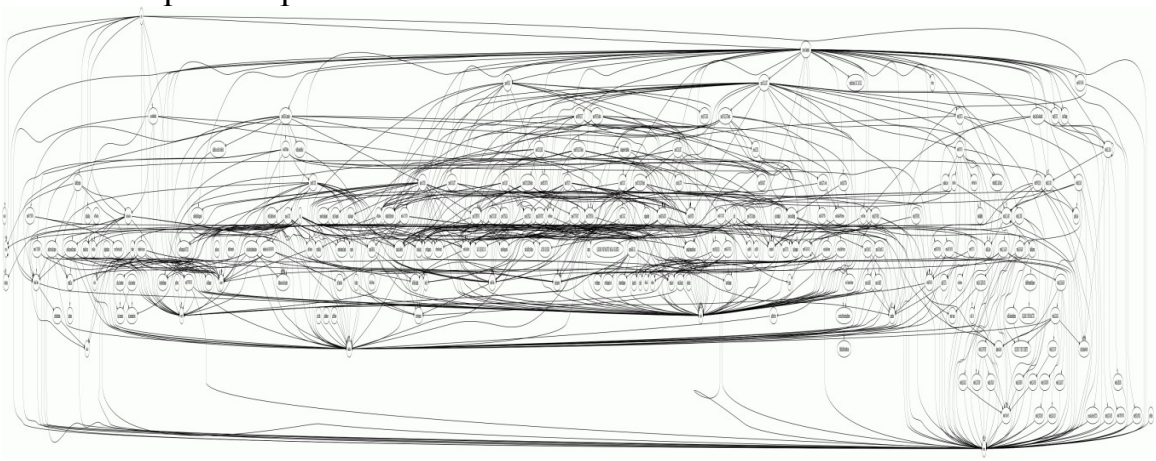
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 674354 (Revision 27)

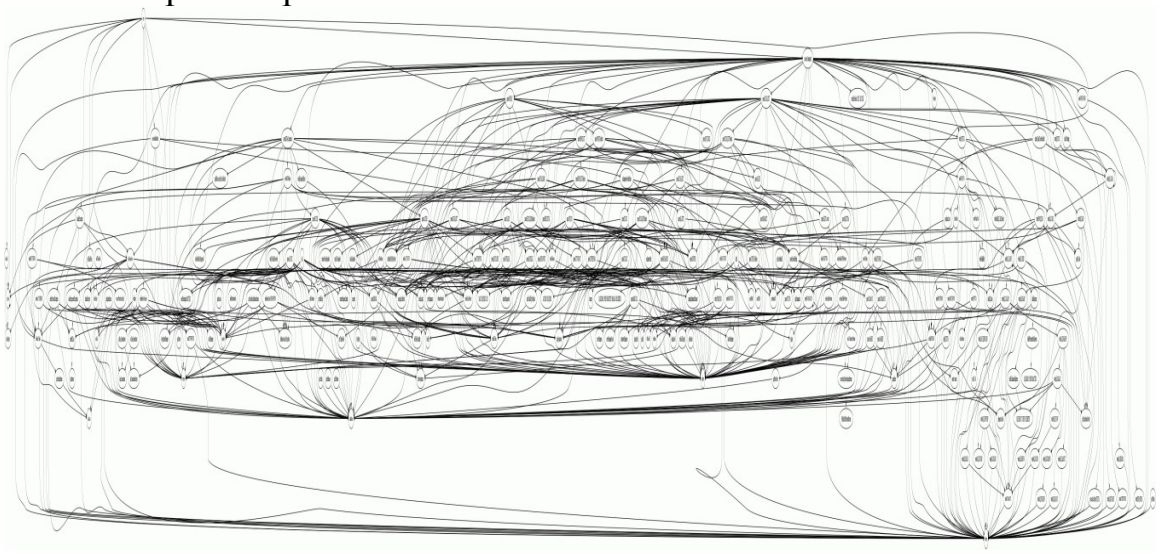
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 700948 (Revision 28)

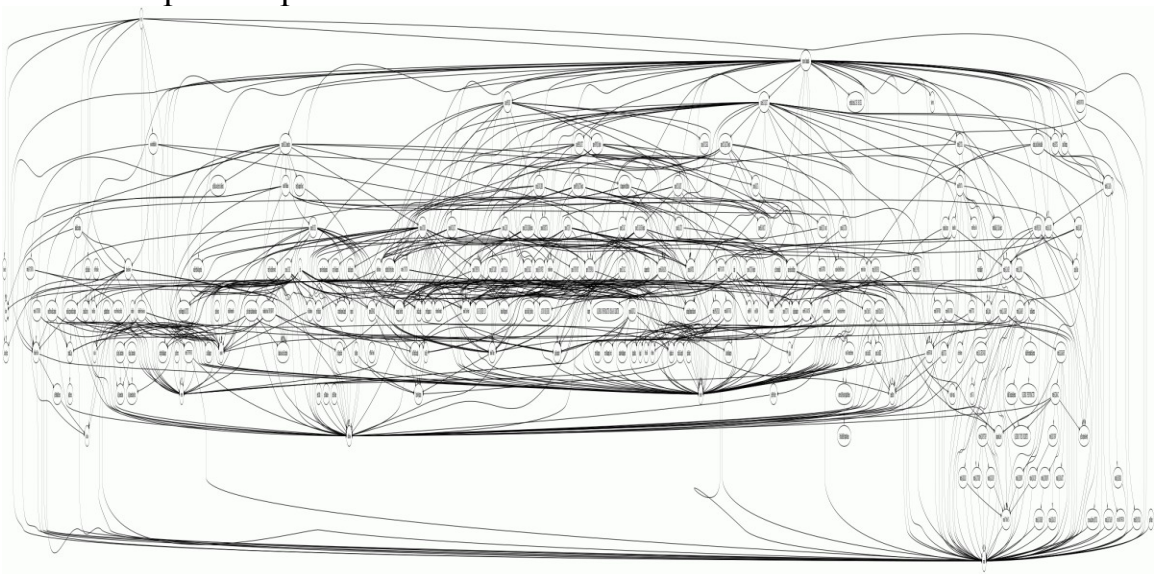
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 701199 (Revision 29)

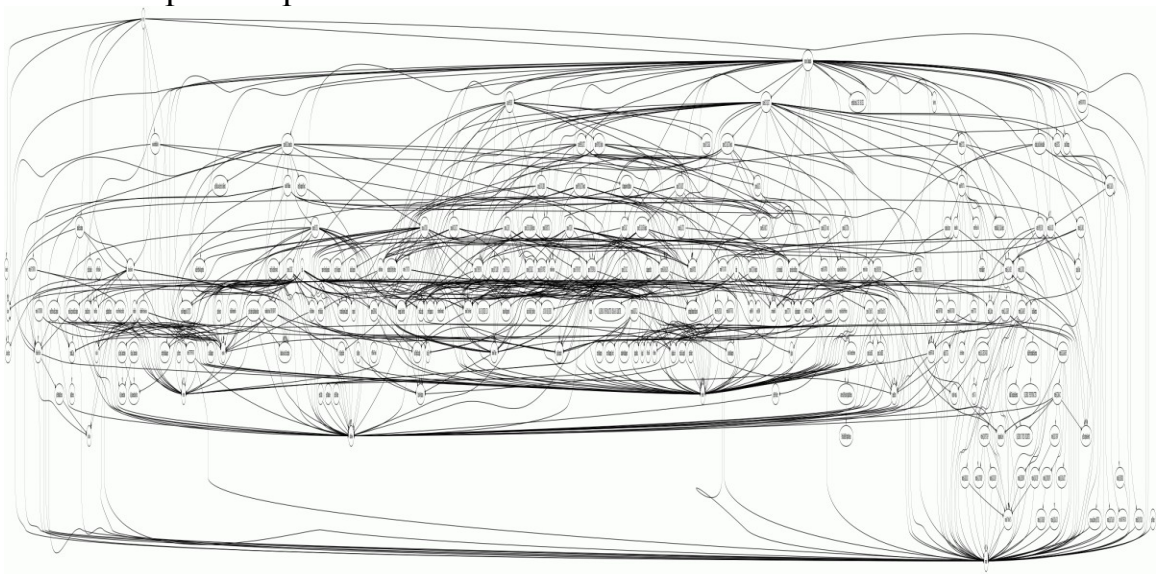
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 703170 (Revision 30)

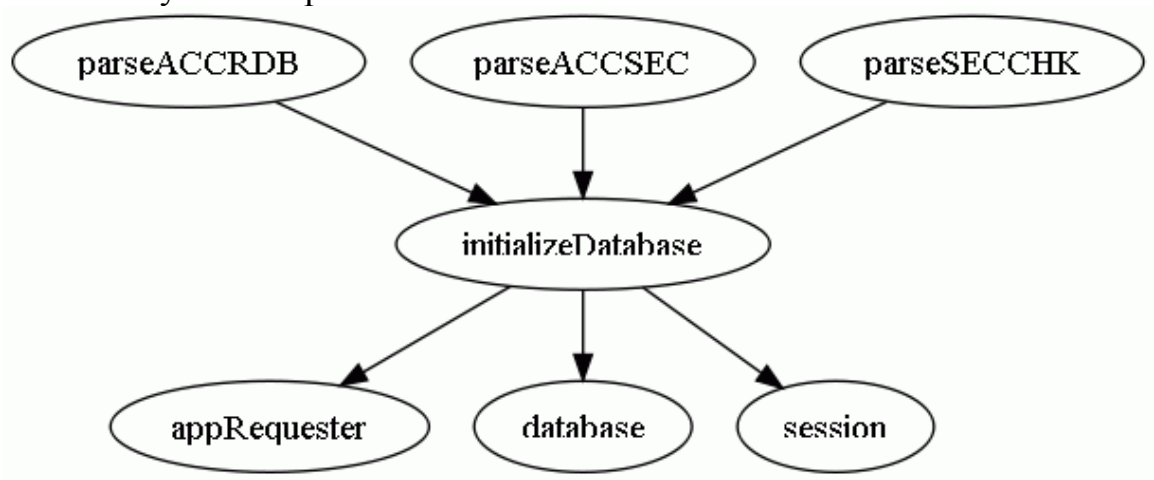
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 734190 (Revision 31)

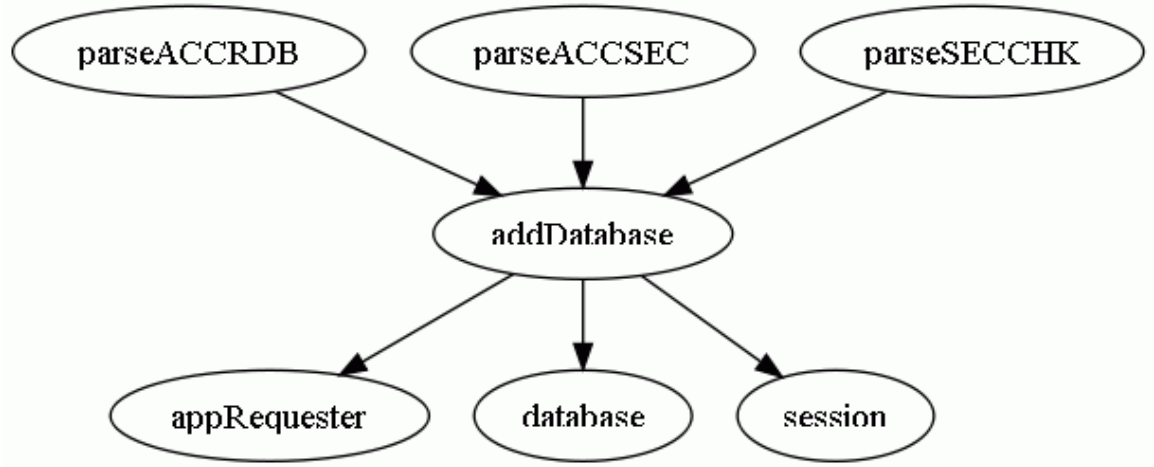
C1 – Newly formed patterns



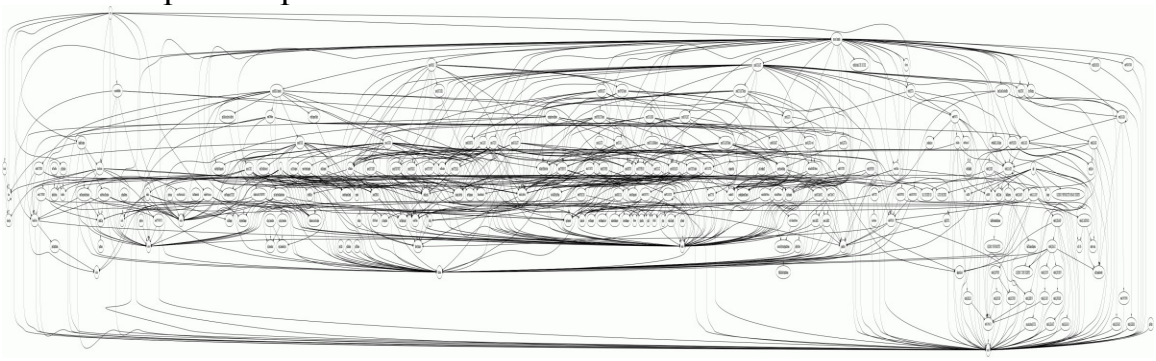
C2 – Patterns disappearing right after their creation

NONE FOUND

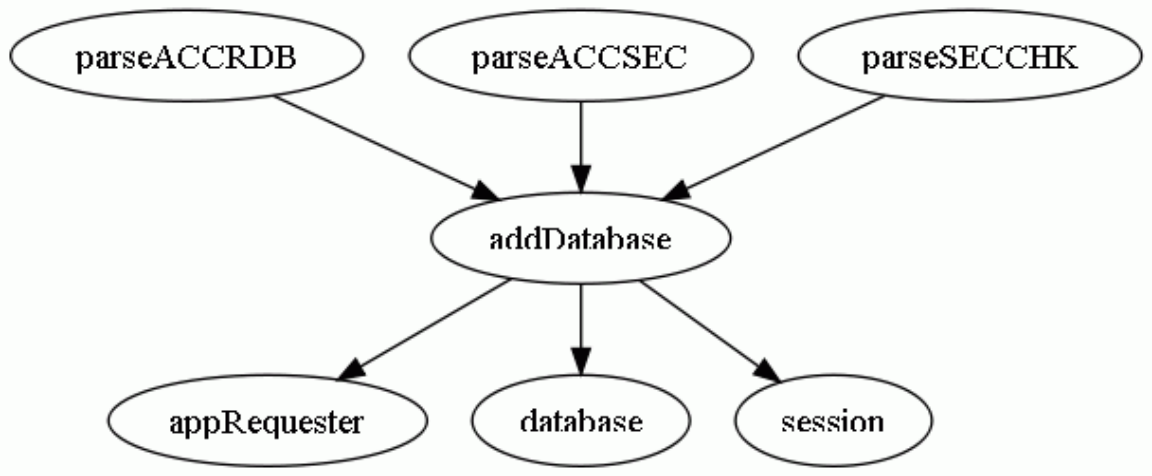
C3 – Impermanent patterns



C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

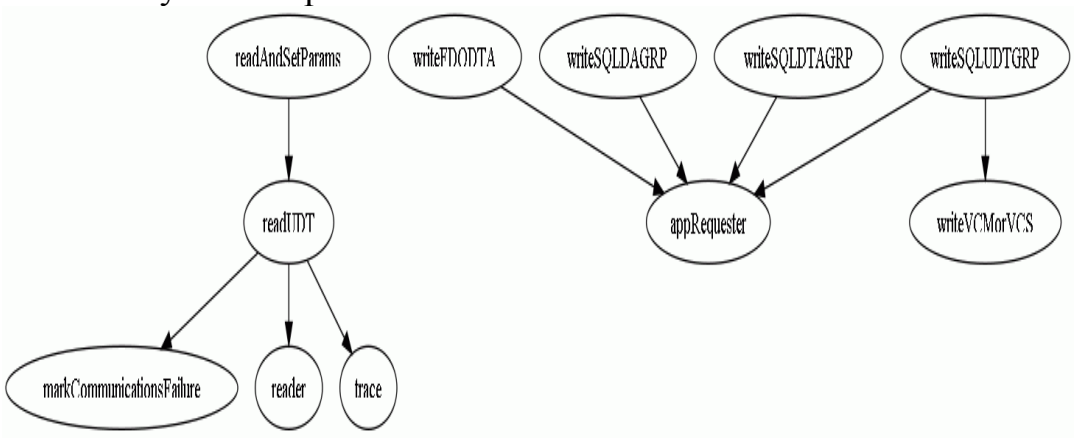


C6 – Reincarnated patterns

NONE FOUND

VERSION 899733 (Revision 32)

C1 – Newly formed patterns



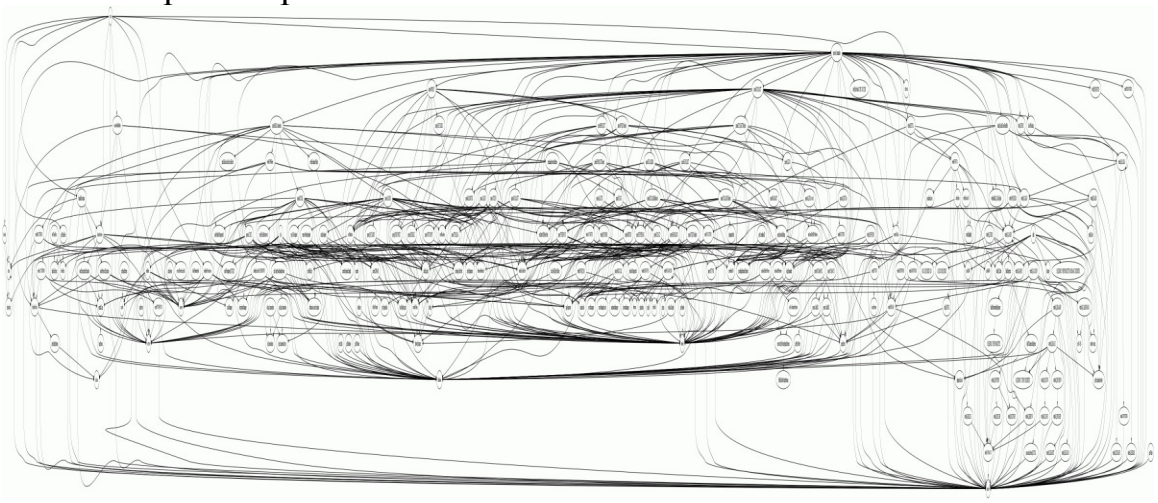
C2 – Patterns disappearing right after their creation

NONE FOUND

C3 – Impermanent patterns

NONE FOUND

C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns

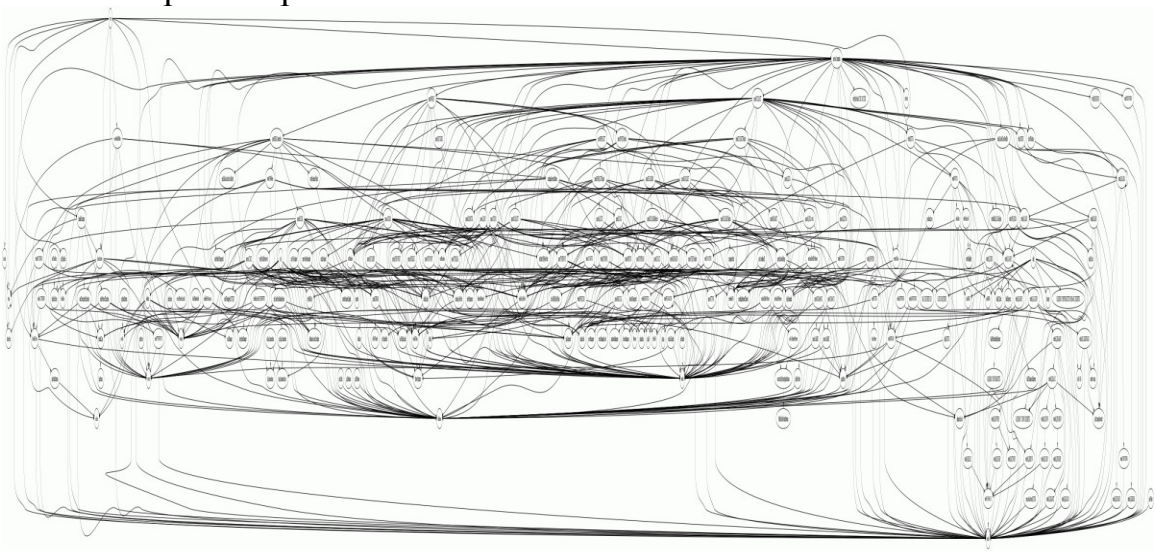
NONE FOUND

C6 – Reincarnated patterns

NONE FOUND

VERSION 901219 (Revision 33)

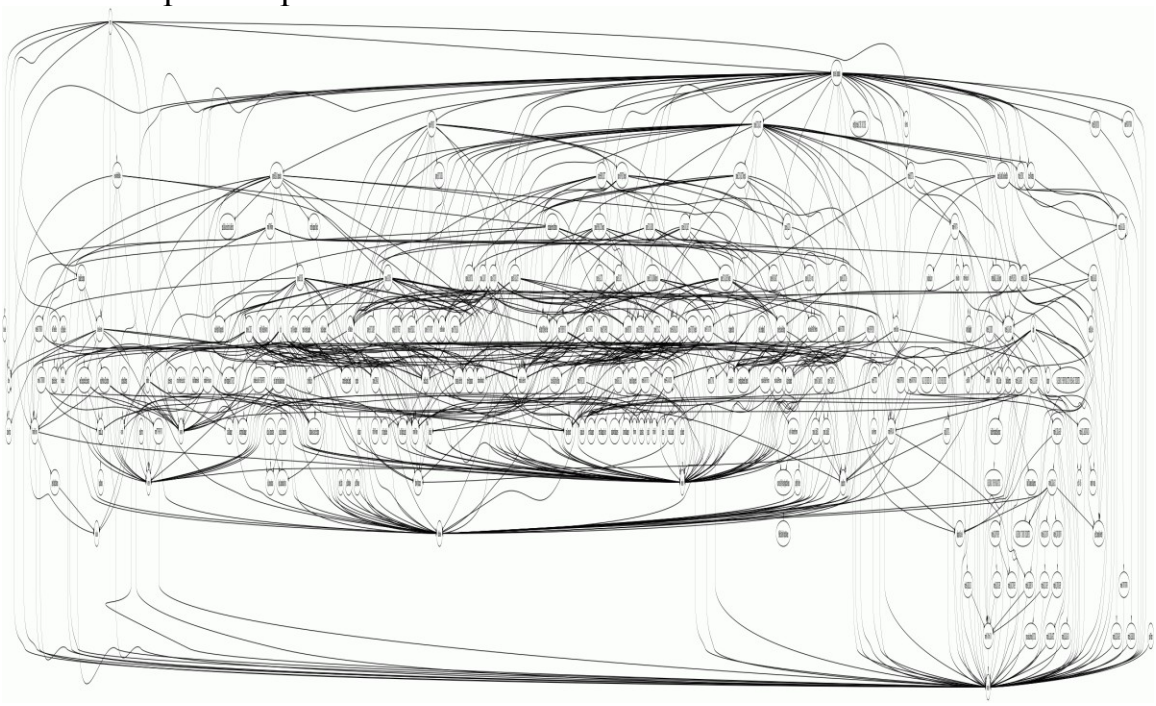
C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

VERSION 924746 (Revision 34)

C1 – Newly formed patterns
NONE FOUND
C2 – Patterns disappearing right after their creation
NONE FOUND
C3 – Impermanent patterns
NONE FOUND
C4 – Omnipresent patterns



C5 – Disappearing omnipresent patterns
NONE FOUND
C6 – Reincarnated patterns
NONE FOUND

APPENDIX E

PROGRAM CODE

This appendix contains the source code of the program that was used to discover patterns in program graphs.

```
/**
* @author imran
* ParserMain.java takes the output files generated by structure101 and
converts
* them to an ARFF.
*
*/
package edu.okstate.cs.imran.DDGraphParser;
import java.io.*;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Hashtable;
import java.util.List;
import java.util.Set;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.xpath.XPath;
import org.jdom.Element;

import sun.java2d.loops.DrawGlyphListAA.General;

import java.sql.*;

public class ParserMain {

/**
* @param args
* @throws JDOMException
* @throws SQLException
* @throws ClassNotFoundException
* @throws IOException
*
* ParserMain
set fos
set dir
set filter
set out
*/

public static void main(String[] args) throws JDOMException,
SQLException, ClassNotFoundException, IOException {


genARFF gen = new genARFF();


gen.generate();


}
```

```
}

/*
* This class generates ARFF files from the graph output files
* generated by structure101.
*
*/
class genARFF{
/*
* attribList arrayList contains the names of attributes (meaning
nodes/items).
*/
ArrayList<String> attribList = new ArrayList<String>();
/*
* attribRows contains all the versions of the graphs.
*/
ArrayList<HashMap> attribRows = new ArrayList<HashMap>();
BufferedReader in;
BufferedWriter out;
FileOutputStream fos;
ObjectOutputStream oos;
FileInputStream fis;
ObjectInputStream ois;
File dir;

/*
* constructor - It creates an object of type genARFF
* and initializes:
*  1) fos with the location of attribList.dat (the dat file used to
keep track of all the edges)
*     in the program we are using attributes to represent edges.
*     2) oos with the newly created fos to serialize it.
*     3) dir with the directory where all the files are located.
*/
public genARFF() throws IOException{

/*
*  initialization and creation of a genARFF object.
*/
fos = new
FileOutputStream("C:/deleteit/API/WEKA/GridSearch/attribList.dat");
oos = new ObjectOutputStream(fos);
//dir is the directory where all the input files are stored.
dir = new File("C:/deleteit/API/WEKA/GridSearch");
}

/*
*  generate() actually generates the arff file. the name of the file is
the current datetime
*  to avoid overwriting the older files.
*
*/
public void generate() throws IOException, ClassNotFoundException{

/*
```

```
* This filter returns only the files that should be used for
* the ARFF generator.
*
*/
FilenameFilter filter = new FilenameFilter() {
public boolean accept(File dir, String name) {
return name.contains("Grid_");

}
};

//create an array which will conatin the files that need to be
processed.
ArrayList<File> files = new ArrayList<File>();

//get all the files from the directory that need to be processed.
File[] filteredFiles = dir.listFiles(filter);
for (File iFile: filteredFiles){
files.add(iFile);
}

/*
* Create ARFF file in the increasing number of instances.
*
*/


List<File> fToBeProcessed = files;

/*
* Files must be sorted to make sure that we process the files
* with the older versions first and newer versions later.
*/
Collections.sort(fToBeProcessed);

/*
* This for loop takes all the attributes from the files.
* These attributes need to be listed in the start of the ARFF file.
* Process all files one by one.
*/
for (File inFile : fToBeProcessed) {
/*
* For each attribute in the file translate it into a the format A->B
* and store them in a map.
*
*/
System.out.println("FileName:"+inFile.getName());
in = new BufferedReader(new FileReader(inFile));
in.readLine();


while (in.ready()) {
String strLine = in.readLine();
String attribItem = strLine.split("\t")[0] + "->"/*

* +strLine.split
```

```
* (

* "\t")[1]+

* "_"

*/
            + strLine.split("\t")[2];

/*
 * If it doesnt exist add attribList
 * or dont do anything. just print already exists.
 */
boolean rslt = attribList.contains(attribItem);

if (!rslt) {
      attribList.add(attribItem);
} else {
      System.out.println(attribItem + " already exists");
}

}

}

/*
* This for loop gathers data for the second part of the ARFF file which
* list the instances. One row represents on version of the source file.
*/
for (File inFile : fToBeProcessed) {
in = new BufferedReader(new FileReader(inFile));
String strLine2 = in.readLine();

HashMap<String, String> attribMap = new HashMap<String, String>();
/*
* initialize the map to set the value of the map as n. n reperesents 0
* or absence of the attribute. If the attribute exists we set the value
for attribute
* as y.
*/
for (String strAttrib : attribList) {
attribMap.put(strAttrib, "n");

}

while (in.ready()) {
strLine2 = in.readLine();
String attribItem = strLine2.split("\t")[0] + "->"/*

* +strLine.split

* (

* "\t")[1]+

* "_"
```

```
*/
            + strLine2.split("\t")[2];
/*
 * Set the value as y for the attribute.
 */
attribMap.put(attribItem, "y");

}

/*
* add the attribMap which represents an instance/ version of the class.
*/
attribRows.add(attribMap);

}

/*
* put all the attributes in oos.
*/
oos.writeObject(attribList);
oos.close();

/*
* print the string in the file.
*/
this.printString();


}

/*
* this function takes the values in attribRows and
* writes them in the ARFF file.
*/
public  void printString() throws IOException{

out = new BufferedWriter(new
FileWriter("C:/deleteit/API/WEKA/GridSearch/dep_graph_output_"+ (new
java.util.Date()).getTime()+".arff"));
out.write("@relation dependence\n");

/*
* write the firtst part of the ARFF file by listing all
* the attributes and all the possible values for the attributes.
*/

Set tempKeySet = attribRows.get(0).keySet();
for(Object strKeySetItem : tempKeySet){
out.write("@attribute '"+strKeySetItem.toString()+"' {'n', 'y'}\n");
}

/*
* write the second part of the ARFF file. List one instance in each
row.
*/
out.write("@data\n");
```

```
for(HashMap<String, String> attribMapItr  : attribRows){

out.write(attribMapItr.values().toString().replaceAll("\\[",
"").replaceAll("]","")+"\n");


}

out.flush();
out.close();
}
}
```

```
/**
* @author imran
*     This class reads the ARFF file and loads it into WEKA.
*     The instances are then stored in the database.
*/

package edu.okstate.cs.imran.MLearner;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Serializable;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import weka.core.Instances;

public class LearnerMain implements Serializable {

/**
* @param args
* @throws Exception
*
*
*             Change reader
*             loadDataToDB
*/
public static void main(String[] args) throws Exception {

/*
 * Loading the ARFF file to WEKA.
 * WEKA models the data in the ARFF file as
 * Instances (type).
 */
BufferedReader reader = new BufferedReader(new FileReader(

      "C:/deleteit/API/WEKA/GridSearch/dep_graph_output_1270371172138.a
rff"));
Instances data = new Instances(reader);
reader.close();

System.out.println("Total Number of instances in
data:"+data.numInstances());

//Load instances to database and populate summary table.
loadDataToDB(data, "GRIDSEARCH");



}
```

```
/*
* This function takes the data from Instances and
* stores it into the database.
*/
public static void loadDataToDB(Instances data, String dataSetName)
throws ClassNotFoundException, SQLException{
//Make the connection to the database and initialize
Class.forName("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://localhost:3306/graphdb";
Connection con = DriverManager.getConnection( url, "root","Lahore123");
Statement stmt = con.createStatement();

/*
 * try to create a database GRAPHDB. If the database
 * already exists then move on.
 */

try{
stmt.executeUpdate("CREATE DATABASE GRAPHDB");
}catch(java.sql.SQLException ex){
      System.out.println("Message:"+ ex.getMessage());
      System.out.println("SQLState:"+ ex.getSQLState());
      System.out.println("Stack Trace:"+
ex.getStackTrace().toString());

}


/*
 * Create a table for the dataset. Each experiment is represented
 * by a dataset. For example, data for KnoweledgeFlowAPP.java is stored
 * in KFLOWAPP table.
 */
try{
      stmt.executeUpdate("CREATE TABLE "+dataSetName+"(ID INT,BUILD
VARCHAR(30) ) ");
      }catch(java.sql.SQLException ex){
            System.out.println("Message:"+ ex.getMessage());
            System.out.println("SQLState:"+ ex.getSQLState());
            System.out.println("Stack Trace:"+
ex.getStackTrace().toString());

      }

      /*
       * For each attribute (item or edge) in the ARFF file
       * create a column in the table.
       */
      try{
            for(int i=0; i< data.numAttributes(); i++){
                  String colName = data.attribute(i).name().replace("-
>", "_TO_").replaceAll("<","_").replaceAll(">","_");
                  stmt.execute("ALTER TABLE "+dataSetName+" ADD
"+colName+" INT");
            }
      }catch(java.sql.SQLException ex){
            System.out.println("Message:"+ ex.getMessage());
```

```
            System.out.println("SQLState:"+ ex.getSQLState());
            System.out.println("Stack Trace:"+
ex.getStackTrace().toString());

      }


      /*
       * For each instance insert the attribute values in the table.
       * If the value in the ARFF file for an attribute is y
       * then insert 1 otherwise insert 0.
       */

      try{
            for(int i=0; i< data.numInstances(); i++){
                  String colVals = "";
                  for(int k=0; k < data.numAttributes(); k++){
                        String numVal = "0";

      if(data.instance(i).stringValue(data.attribute(k)).equalsIgnoreCa
se("y")){
                              numVal = "1";
                        }
                        colVals += "'"+numVal+"'";
                  }


                  colVals =
colVals.replaceAll("''",",").replaceAll("'", "" );

      //System.out.println("colSize:"+colVals.split(",").length);
                  System.out.println("colVals:"+colVals);
                  String strStmt = "INSERT INTO "+dataSetName+"
VALUES("+i+",'build_ver',"+colVals+")";
                  System.out.println("strStmt:"+strStmt);
                  stmt.execute(strStmt);
            }

      }catch(java.sql.SQLException ex){
            System.out.println("Message:"+ ex.getMessage());
            System.out.println("SQLState:"+ ex.getSQLState());
            System.out.println("Stack Trace:"+
ex.getStackTrace().toString());
      }

}

}
```

```
/*
 * @author imran
 *    This class calculates the support count of each attribute
 *    in each instance and stores it into the database then
 *    it uses this data to discover attributes based on the
 *    different conditions for support count and value.
 */

package edu.okstate.cs.imran.MLearner;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DataAnalyzer {

/**
* @param args
*/
private static Connection con = null;

public static void updateBuildStats(int buildId, String tableName)
throws SQLException{
ResultSet rSetColName = null;
Statement stmt = null;
stmt = con.createStatement();

/*
 * Calculate support count for each attribute
 */

//Get the name of all the attributes from the informationschema

String query = "SELECT COLUMN_NAME FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME ='"+tableName +"' and COLUMN_NAME NOT IN
('ID','BUILD')" ;

rSetColName = stmt.executeQuery(query);

while(rSetColName.next()){
      String columnName = rSetColName.getString("COLUMN_NAME");

      /*
       * Get the value of the attribute for that build
       */
      query = "SELECT "+columnName+" AS ATTRIBVAL FROM "+tableName + "
where ID = "+buildId;
      Statement attribValStmt = con.createStatement();
      ResultSet rsAttribVal = attribValStmt.executeQuery(query);
      rsAttribVal.next();
```

```
      String strAttribVal = rsAttribVal.getString("ATTRIBVAL");



      query = "SELECT SUM("+columnName+")AS SCOUNT from "+tableName + "
where ID <= "+buildId;

      Statement tempStmt = con.createStatement();
      ResultSet rsTemp = tempStmt.executeQuery(query);
      rsTemp.next();

      String sCount = rsTemp.getString("SCOUNT");

      try{
            query = "INSERT INTO BUILD_STATS
(DATASET,BUILD_ID,ATTRIBNAME, SCOUNT, VALUE) VALUES('"+tableName+"','"+
buildId+"','"+columnName+"','"+sCount+"','"+strAttribVal+"')";
            tempStmt.execute(query);
            //
      }catch(java.sql.SQLException ex){
            if(ex.getMessage().contains("Duplicate entry")){
                  tempStmt.executeUpdate("UPDATE BUILD_STATS SET SCOUNT
= '"+sCount+"' , VALUE = '"+strAttribVal+"' WHERE DATASET = '"+
tableName+"' AND BUILD_ID = '"+buildId+"' and ATTRIBNAME = '"+
columnName+"'");

            }
            else{
                  System.out.println("***************"+ex);
                  throw ex;
            }
      }



}
}


public static String generateGraph(String itemsets,  String graphName){

String outStr = "digraph "+graphName+"{\n";
if(itemsets != ""){
      for(String str : itemsets.split("\\|")){
            str = str.replaceAll("_TO_", "->")+";\n";
            outStr += str;
      }
}
outStr += "}";
return outStr;
}

public static void printGraph(String strGraph, String graphName, String
dirFullPath) throws IOException{
BufferedWriter out = new BufferedWriter(new
FileWriter(dirFullPath+graphName));
out.write(strGraph);
```

```
out.close();
}

public static void main(String[] args) throws ClassNotFoundException,
SQLException, IOException {
// TODO Auto-generated method stub

/*
 * Parameters that need to go to a config file or some other input
 */
String dataSet = "gridsearch";
int buildId = 3;
String dir = "c:/deleteit/API/WEKA/GridSearch/FPs/";

/*
 * Create DB Connection
 */
Class.forName("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://localhost:3306/graphdb";
con = DriverManager.getConnection( url, "root","Lahore123");


/*
 * Update stats. buildId is the row number the index starts at 0.
 */

for(int i=0; i<= buildId; i++){
      System.out.println("Updating build "+i);
      updateBuildStats(i, dataSet);
}


/*
 * Calling the analyzer methods
 */



for(int i=0; i<= buildId; i++){
      System.out.println("Analuzing build "+i);

      String itemSets = newlyBornThisVersion(i, dataSet);
      itemSets = generateGraph(itemSets, "newlyBornThisVersion_"+i);
      printGraph(itemSets, "newlyBornThisVersion_"+i+".gv", dir );

      itemSets = justDiedAge1(i, dataSet);
      itemSets = generateGraph(itemSets, "justDiedAge1_"+i);
      printGraph(itemSets, "justDiedAge1_"+i+".gv", dir );

      itemSets = omnipresent(i, dataSet);
      itemSets = generateGraph(itemSets, "omnipresent_"+i);
      printGraph(itemSets, "omnipresent_"+i+".gv", dir );

      itemSets = justDeceasedOmnipresent(i, dataSet);
      itemSets = generateGraph(itemSets, "justDeceasedOmnipresent_"+i);
      printGraph(itemSets, "justDeceasedOmnipresent_"+i+".gv", dir );
```

```
	itemSets = reincarnated(i, dataSet);
	itemSets = generateGraph(itemSets, "reincarnated_"+i);
	printGraph(itemSets, "reincarnated_"+i+".gv", dir );

	itemSets = justDied(i,dataSet);
	itemSets = generateGraph(itemSets, "justDied_"+i);
	printGraph(itemSets, "justDied_"+i+".gv", dir );
}




}

/*
* NEWLY BORN INFANT
* Just spits out in println all the items with supportcount1 with
value(N)=y
* It represents a dependency which just appeared in this version.
* Newly born infant dependency age1
*/
public static String newlyBornThisVersion(int buildId, String
tableName) throws SQLException{
String outStr = "";
String query = "SELECT ATTRIBNAME FROM build_stats WHERE DATASET='"+
tableName+"' AND BUILD_ID = "+buildId+" AND SCOUNT=1 AND VALUE=1";
Statement fStmt = con.createStatement();
ResultSet fRsltSet = fStmt.executeQuery(query);

while(fRsltSet.next()){
	//System.out.println(fRsltSet.getString("ATTRIBNAME"));
	outStr += fRsltSet.getString("ATTRIBNAME") +"|";
}

return outStr;
}

/*
* DECEASED INFANTS
* Deceased Infant age1
* Represents dependency that had support count1 and value(N)=n and
value(N-1)=y
*
*/
public static String justDiedAge1(int buildId, String tableName) throws
SQLException{
String outStr = "";
/*
 * If buildId is 0 then there is nothing before 0 to compare with
 */
if(buildId == 0){
	return outStr;
}
String query = "SELECT ATTRIBNAME FROM build_stats WHERE DATASET='"+
tableName+"' AND BUILD_ID = "+buildId+" AND SCOUNT=1 AND VALUE=0";
Statement fStmt = con.createStatement();
ResultSet fRsltSet = fStmt.executeQuery(query);
```

```
while(fRsltSet.next()){
      //System.out.println(fRsltSet.getString("ATTRIBNAME"));
      query = "SELECT VALUE FROM build_stats WHERE DATASET='"+
tableName + "' AND BUILD_ID = "+ (buildId - 1) + " AND
ATTRIBNAME='"+fRsltSet.getString("ATTRIBNAME")+"'";
      //System.out.println(query);
      Statement prevStmt = con.createStatement();
      ResultSet preRsltSet = prevStmt.executeQuery(query);
      preRsltSet.next();
      if(preRsltSet.getString("VALUE").equalsIgnoreCase("1"))
      {
            //System.out.println(fRsltSet.getString("ATTRIBNAME"));
            outStr += fRsltSet.getString("ATTRIBNAME") + "|";
      }

}

return outStr;
}

/*
* OMNIPRESENT
* SupportCount = N: Omnipresent dependency
*/
public static String omnipresent(int buildId, String tableName) throws
SQLException{
String outStr = "";
String query = "SELECT ATTRIBNAME FROM build_stats WHERE DATASET='"+
tableName+"' AND BUILD_ID = "+buildId+" AND SCOUNT="+(buildId+1);
Statement fStmt = con.createStatement();
ResultSet fRsltSet = fStmt.executeQuery(query);

while(fRsltSet.next()){
      //System.out.println(fRsltSet.getString("ATTRIBNAME"));
      outStr += fRsltSet.getString("ATTRIBNAME") + "|";
}

return outStr;
}

/*
* JUST DECEASED OMNIPRESENT
* SupportCountN-1 with value(N)=n and value(N-1)=y:just deceased
omnipresent
*/
public static String justDeceasedOmnipresent(int buildId, String
tableName) throws SQLException{
String outStr = "";
/*
 * If buildId is 0 then there is nothing before 0 to compare with
 */
if(buildId == 0){
      return outStr;
}
```

```
String query = "SELECT ATTRIBNAME FROM build_stats WHERE DATASET='"+
tableName+"' AND BUILD_ID = "+buildId+" AND VALUE=0 AND
SCOUNT="+(buildId);
Statement fStmt = con.createStatement();
ResultSet fRsltSet = fStmt.executeQuery(query);

while(fRsltSet.next()){
      //System.out.println(fRsltSet.getString("ATTRIBNAME"));
      outStr += fRsltSet.getString("ATTRIBNAME") + "|";
}

return outStr;
}

/*
* RE_INCARNATED
* SupportCount > 1 and with value(N)=Y and value(N-1)=N
* How to find the previous incarnation of this dependency?
*/
public static String reincarnated(int buildId, String tableName) throws
SQLException{
String outStr = "";

/*
 * If buildId is 0 then there is nothing before 0 to compare with
 */
if(buildId == 0){
      return outStr;
}

String query = "SELECT ATTRIBNAME FROM build_stats where DATASET = '"+
tableName+"' AND BUILD_ID = "+buildId+" AND VALUE=1 AND SCOUNT > 1";
Statement fStmt = con.createStatement();
ResultSet fRsltSet = fStmt.executeQuery(query);

/*
 * If in the previous build the same attribute had a value n then its a
reincarnation provided its support count > 1
 */

while(fRsltSet.next()){
      query = "SELECT VALUE FROM build_stats WHERE DATASET='"+
tableName + "' AND BUILD_ID = "+ (buildId - 1) + " AND
ATTRIBNAME='"+fRsltSet.getString("ATTRIBNAME")+"'";
      //System.out.println(query);
      Statement prevStmt = con.createStatement();
      ResultSet preRsltSet = prevStmt.executeQuery(query);
      preRsltSet.next();
      if(preRsltSet.getString("VALUE").equalsIgnoreCase("0"))
      {
            //System.out.println(fRsltSet.getString("ATTRIBNAME"));
            outStr += fRsltSet.getString("ATTRIBNAME") + "|";
      }

}

return outStr;
```

```
}

/*
* JUST DIED
* value(N)=N and value(N-1)=Y
*
*/
public static String justDied(int buildId, String tableName) throws
SQLException{
String outStr = "";
/*
 * If buildId is 0 then there is nothing before 0 to compare with
 */
if(buildId == 0){
      return outStr;
}
String query = "SELECT ATTRIBNAME FROM build_stats where DATASET = '"+
tableName+"' AND BUILD_ID = "+buildId+" AND VALUE=0";
Statement fStmt = con.createStatement();
ResultSet fRsltSet = fStmt.executeQuery(query);

/*
 * If in the previous build the same attribute had a value n then its a
reincarnation provided its support count > 1
 */

while(fRsltSet.next()){
      query = "SELECT VALUE FROM build_stats WHERE DATASET='"+
tableName + "' AND BUILD_ID = "+ (buildId - 1) + " AND
ATTRIBNAME='"+fRsltSet.getString("ATTRIBNAME")+"'";
      //System.out.println(query);
      //System.out.println(fRsltSet.getString("ATTRIBNAME"));

      Statement prevStmt = con.createStatement();
      ResultSet preRsltSet = prevStmt.executeQuery(query);
      preRsltSet.next();
      //System.out.println(preRsltSet.getString("VALUE"));
      if(preRsltSet.getString("VALUE").equalsIgnoreCase("1"))
      {
            //System.out.println(fRsltSet.getString("ATTRIBNAME"));
            outStr += fRsltSet.getString("ATTRIBNAME") + "|";
      }

}

return outStr;
}



}
```

VITA

Imran Afzal

Candidate of the Degree of

Master of Science

Thesis: MINING FOR PATTERNS IN PROGRAM DEPENDENCE GRAPHS

Major Field: Computer Science

Biographical:

Personal Data: Born in Abbotabad, Pakistan, on March 11, 1973, son of Mr. and Mrs. Muhammad Afzal.

Education: Received the Bachelor of Electrical Engineering Degree from University of Engineering and Technology, Lahore, Pakistan in December 1995. Completed the requirements for the Master of Science Degree in Computer Science at the Computer Science Department at Oklahoma State University in July 2010.

Experience: Employed by Safeguard Packages, Lahore, Pakistan, as System Engineer, January 1996 to July 1998; employed by Xavor Pvt. Limited, Lahore, Pakistan, as Software Engineer, August 1999 to August 2000; employed by Habib Bank Limited, New York, NY, USA, as IT Manager, December 2002 to April 2007; employed by Infonox, A Tsys Company, CA, USA, as Client Server Applications Analyst Lead, April 2007 to present.