Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process

Andy Zaidman*, Toon Calders+, Serge Demeyer*, Jan Paredaens+

+ Advanced Database Research and Modelling (ADReM)
* Lab On Re-Engineering (LORE)

University of Antwerp
Department of Mathematics and Computer Science
Middelheimlaan 1, 2020 Antwerp, Belgium
{Andy.Zaidman, Toon.Calders, Serge.Demeyer, Jan.Paredaens}@ua.ac.be

Abstract

Well-designed object-oriented programs typically consist of a few key classes that work tightly together to provide the bulk of the functionality. As such, these key classes are excellent starting points for the program comprehension process. We propose a technique that uses webmining principles on execution traces to discover these important and tightly interacting classes. Based on two medium-scale case studies – Apache Ant and Jakarta JMeter – and detailed architectural information from its developers, we show that our heuristic does in fact find a sizeable number of the classes deemed important by the developers.

Keywords
Reverse engineering, dynamic analysis, webmining, program comprehension

1 Introduction

Reverse engineering is defined as the analysis of a system in order to identify its current components and their dependencies and to create abstractions of the systems design [3]. A reverse engineering operation almost always takes place in service of a specific purpose, such as re-engineering to add a specific feature, maintenance to improve the efficiency of a process, reuse of some of its modules in a new system, etc.[23] In order to perform any of these operations the software engineer must comprehend a given program sufficiently well to plan, design and implement modifications and/or additions. As such, program comprehension can be defined as the process the software engineer goes through when studying the software artifacts, with as goal the sufficient understanding of the system to perform the required operations [18]. Such software artifacts could include the source code, documentation and/or abstractions from the reverse engineering process.

Gaining understanding of a program is a time-consuming task taking up to 40% of the time-budget of a maintenance operation [26, 21, 4]. The manner in which a programmer gets understanding of a software system varies greatly and depends on the individual, the magnitude of the program, the level of understanding needed, the kind of system, ... [17, 28, 8, 22]

As such sizeable gains in overall efficiency can be attained by providing assistance to the software (reverse) engineer for his/her program understanding process. We propose a heuristic in this paper that can help the engineer with finding important classes that should be looked at first when starting the comprehension process.

Studies and experiments reveal that the success of decomposing a program into effective mental models depends on one's general and program-specific domain knowledge [25]. While a number of different models for the cognition process have been identified, most models fall into one of three categories: top-down comprehension, bottom-up comprehension or a hybrid model combining the previous two [19]. The top-down model is traditionally employed by programmers with code domain familiarity. By drawing on their existing domain knowledge, programmers are able to efficiently reconcile application source code with system goals. The bottom-up model is often applied by programmers working on unfamiliar code [5]. To comprehend the application, they build mental models by evaluating program code against their general programming knowledge [18].

Because of the human cognition process [28], program understanding can never be a fully automated process: the programmer should be free to explore the software, with the help of specialized tools [9, 6]. These program exploration tools should identify those parts of the program that are likely to be interesting from a program understanding point of view [14]. For instance, in the case of object-oriented programs – which is the main focus of our work – program exploration tools should reveal those classes that form core parts of the design.

Orthogonal to the selection of the cognitive strategy, i.e. which mental model to employ, is the choice between several technical strategies, namely (1) static analysis, i.e., by examining the source code, (2) dynamic analysis, i.e., by examining the program's behavior, or (3) a combination of both.

In the context of object-oriented systems, due to polymorphism, static analysis is often imprecise with regard to the actual behavior of the application [27]. Dynamic analysis, however, allows to create an exact image of the program's intended runtime behavior. Our actual goal is to find frequently occurring interaction patterns between classes. These interaction patterns can help us build up understanding of the software.

In this paper we propose a technique that applies datamining techniques to event traces of program runs. As such, our technique can be catalogued in the dynamic analysis context. The technique we use was originally developed to identify important *hubs* on the Internet, i.e., pages with many links to authorative pages, based on only the links between web pages [16]. Hence, the Internet is viewed as a large graph. We verify that important classes in the program correspond to the hubs in the dynamic call-graph of a program trace.

We apply the proposed technique to two medium-scale case studies, namely Apache Ant and Jakarta JMeter. The results show that the *hubiness* is indeed a good measure for finding important classes in the system's architecture.

The organization of the paper is as follows. First, in Section 2, we give an overview of the different steps in the process and the different algorithms we use. Section 3 shows how we plan to validate the results of our technique. Section 4 explains the datamining algorithm in detail, while in Section 5 the results of applying our technique on the two case studies are discussed. Section 6 explores related work, while Section 7 points to future research and concludes the paper.

2 Overview of our proposed technique

The technique we propose can be seen as a 4-step process. In this section we explain each of the 4 steps.

Define execution scenario. Applying dynamic analysis requires that the program is executed at least once. The execution scenario, i.e., which functionality of the program gets executed, is very important as it has a great influence on the results of the technique. For example, if the software engineer is reverse engineering a banking application and more specifically wants to know the inner workings of how interest rates are calculated, the execution scenario should at least contain one interest rate calculation.
On the other hand, by keeping the execution scenario specific, e.g. only calculating the interest rate, and not executing money transfers, the final results will be more precise. In terms of UML, this would be the same as limiting the number of use cases [13].

Non-selective profiling. Once the execution scenario has been defined, the program must be executed according to the defined scenario. During the execution all calls to and returns from methods are logged in the event trace. For this step, we relied on a custom-made JVMPI[1] profiler.
Please note however that even for small and medium-scale software systems and precisely defined execution scenarios, event traces become very large. Table 1 gives an overview of some metric-data for our two case studies.

Datamining. By examining the event trace we want to discover the classes in the system that play an active role in the execution scenario. Classes that have an active role depend on many other classes to perform functions for them.

In Figure 1 we show an example of a *compacted call graph*. The compacted call graph is derived from the dynamic call graph; it shows an edge between two classes A $\rightarrow$ B if an instance of class A sends a message to an instance of class B. The weights on the edges give an indication of the tightness of the collaboration as it is the number of distinct messages that are sent between instances of both classes. More formally:

$$weight(A, B) = |\bigcup_{i,j} M(a_i, b_j)|$$

where a_i and b_j are instances of respectively class A and class B and $M(a, b)$ is the set of messages sent from a to b.

[1]Java Virtual Machine Profiler Interface: for more information see: http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html

	Apache Ant 1.6.1	Jakarta JMeter 2.0.1
Classes (traced)	127	189
Classes (total)	1 216	245
Lines of Code (LOC), total	98 681	22 234
Events	24 270 064	138 704
# objects at runtime	18500	4180
Scenario	building Ant	1 simulation run
Execution time (without tracing)	23 s	82 s

- *The number of events as shown in Table 1 takes into account both method entries and exits. As thus, the number of method invocations is actually 50% of this number.*
- *In the case of Apache Ant, we only took into account the 127 core classes of Ant. Tracing all classes of the Ant suite would take the number of participating classes to around one thousand. The same reasoning applies to the 189 core classes of Jakarta JMeter.*
- *What stands out is the big difference in the number of events between Ant and JMeter even though the execution time for the latter is much longer. This can be attributed to the fact that: (1) JMeter uses a lot of Java base classes for e.g. network-related functionality and (2) network-related operations can take a relatively long time due to the uncertain network conditions.*

Table 1. Size of an event trace of two medium-size programs

This compacted call graph is the input to the datamining algorithm that is presented in detail in section 4.

Interpretation of results. The goal we wish to attain is guiding the software engineer through the software in order to direct him/her in his/her program comprehension process. Because the original event trace is (1) too large to study directly (even in a visualized form), and (2) shows too many unimportant sections, e.g. long loops in the execution, we want to be able to deliver the software engineer with a number of *slices* of the trace that form good starting points for the program understanding process.

To the user, these starting points can be:

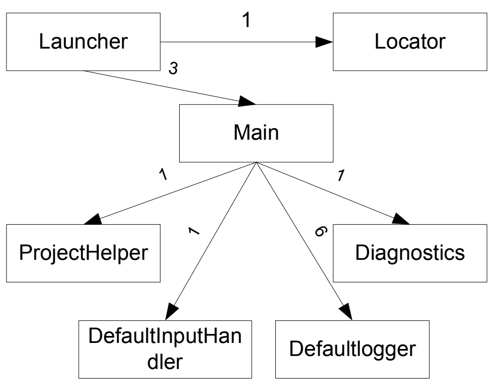

Figure 1. A compacted call graph

- A list of key classes they should examine first. The software engineer can then examine these key classes through inspection of the code. This is a *statical approach* to interpreting the results.
- A visualization of these key classes and their immediately collaborating classes. These so-called *slices* [20] can e.g. be obtained through the use of aspect oriented programming (AOP) [15]. The use of AOP *pointcuts*, targets in the original source code where extra functionality – called *advice* – allows for a more specific trace to be constructed, i.e. a trace that focusses only on interesting sections in program execution, but that contains more detailed information, e.g. values of parameters, access to certain fields of the class, ...

3 Validation

As validation we propose to verify whether the classes our technique marks as important are also deemed important by the developers. For our two case studies we have made sure that extensive design documentation is available that covers not only the original design choices, but also the architectural evolution throughout the life-cycle. This documentation is normally intended for software engineers willing to participate in the open-source development process of both projects. As such, this kind of documentation is meant to give a good high-level insight into the structure and the working of the software for software engineers who are new to the project. Because these novices are the intended users of our technique, these design documents will serve as validation for our results.

The public nature of the documentation furthermore ensures repeatability of the experiments.

Our technique is concentrated around dynamic analysis, because we believe that dynamic analysis is better at

capturing certain inter-class relationships, which are not immediately evident from static examination of a software project. In order to verify this claim, we also look at static coupling measures. For this, we take *Coupling Between Objects* (CBO) [2] as a representative. CBO can be seen as a typical static coupling measure which can help in identifying classes with a coordinating role.
Chidamber and Kemerer [2] explain CBO as being:

> CBO for a class is a count of the number of other classes to which it is coupled. CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another.

4 Webmining techniques

In datamining, many successful techniques have been developed to analyze the structure of the web [1, 10, 16]. Typically, these methods consider the Internet as a large graph, in which, based solely on the hyperlink structure, important web pages can be identified. In this section we show how to apply these successful web mining techniques to a compacted call graph of a program trace, in order to uncover important classes.

First we introduce the HITS webmining-algorithm [16] to identify so-called hubs and authorities on the web. Then, the HITS algorithm is combined with the compacted call graph. We argue that the classes that are associated with good "hubs" in the compacted call graph are good candidates for early program understanding.

4.1 Identifying hubs in large webgraphs

In [16], the notions of *hub* and *authority* were introduced. Intuitively, on the one hand, hubs are pages that rather refer to pages containing information then being informative themselves. Standard examples include web directories, lists of personal pages, ... On the other hand, a page is called an authority if it contains useful information. Hence, a web-page is a good hub if it points to important information pages, e.g., to good authorities. A page can be considered a good authority if it is referred to by many good hubs. The HITS algorithm is based on this relation between hubs and authorities.

Example Consider the webgraph given in Figure 2. In this graph, 2 and 3 will be good authorities, and 4 and 5 will be good hubs, and 1 will be a less good hub. The authority of 2 will be larger than the authority of 3, because the only in-links that they do not have in common are $1 \to 2$ and $2 \to 3$, and 1 is a better hub than 2. 4 and 5 are better hubs than 1, as they point to better authorities.

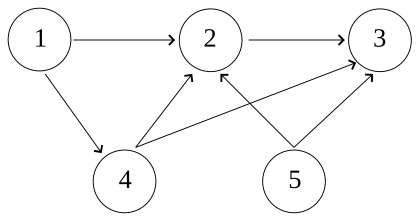

Figure 2. Example web-graph

The HITS algorithm works as follows. Every page i gets assigned to it two numbers; a_i denotes the authority of the page, while h_i denotes the hubiness. Let $i \to j$ denote that there is a hyperlink from page i to page j. The recursive relation between authority and hubiness is captured by the following formula's.

$$h_i = \sum_{i \to j} a_j \quad (1)$$

$$a_j = \sum_{i \to j} h_i \quad (2)$$

The HITS algorithm starts with initializing all h's and a's to 1, and repeatedly updates the values for all pages, using the formula's (1) and (2). If after each update the values are normalized, this process converges to stable sets of authority and hub weights [16].

It is also possible to add weights to the edges in the graph. Adding weights to the graph can be interesting to capture the fact that some edges are more important than others. This extension only requires a small modification to the update rules. Let $w[i,j]$ be the weight of the edge from page i to page j. The update rules become $h_i = \sum_{i \to j} w[i,j] \cdot a_j$ and $a_j = \sum_{i \to j} w[i,j] \cdot h_i$.

Example For the graph given in 2, the hub and authority weights converge to the following (normalized) values:

$$\begin{array}{lcl@{\qquad}lcl} h_1 &=& 64 & a_1 &=& 0 \\ h_2 &=& 48 & a_2 &=& 100 \\ h_3 &=& 0 & a_3 &=& 94 \\ h_4 &=& 100 & a_4 &=& 24 \\ h_5 &=& 100 & a_5 &=& 0 \end{array}$$

In the context of webmining, the identification of hubs and authorities by the HITS algorithm has turned out to be very useful. Because HITS only uses the links between webpages, and not the actual content, it can be used on arbitrary graphs to identify important hubs and authorities.

4.2 Identifying key classes

Within our problem domain, hubs can be considered *coordinating classes*, while authorities correspond to classes providing small functionalities that are used by many other classes. As such, the hub classes play a pivotal role in a system's architecture. Therefore, hubs are excellent candidates for beginning the program comprehension process or for gaining quick and initial program understanding.

5 Case studies

5.1 Apache Ant

Introduction. Ant is an XML based Java build tool. We chose Apache Ant 1.6.1[2] because we consider it to be a medium-size program (see also Table 1) and because of the extensive design information that is publicly made available by the developers. As such we have clear evidence about the classes the developers consider to be important[3]. This knowledge will help us in validating our technique.

Execution scenario. As execution scenario we have chosen to let Ant build itself, i.e., we supplied the XML build file that comes with the Apache Ant 1.6.1 source code edition. This scenario was chosen because (1) the Ant build file is representative for typical Ant functionality and (2) it allows for easy verification of the results presented in this paper.

Architectural overview. Now, with the help of the design documentation, we will discuss the role the five classes that are considered important by the architects, play in the execution of a build.xml file:

1. `Project`: Ant starts in the Main class and immediately creates a `Project` instance. With the help of subsidiary objects, the `Project` instance parses the build.xml file. The xml file contains *targets* and *elements*.

2. `Target`: this class acts as a placeholder for the *targets* specified in the build.xml file. Once parsing finishes, the build model consists of a project, containing multiple targets – at least one, which is the implicit target for top-level events.

3. `UnknownElement`: all the elements that get parsed are temporarily stored in instances of `UnknownElement`. During parsing the `UnknownElements` objects are stored in a tree-like datastructure in the `Target` to which they belong. When the parsing phase is over and all dependencies have been determined, the `makeObject()` method of `UnknownElement` gets called, which instantiates the right kind of object for the data that was kept in the placeholder UnknownElement object.

4. `RuntimeConfigurable`: each `UnknownElement` has a corresponding `RuntimeConfigurable`, that contains the element's configuration information. The `RuntimeConfigurable` objects are also stored in trees in the `Target` object they belong to.

5. `Task` is the superclass of `UnknownElement` and is also the baseclass for all types of tasks that are created by calling the `makeObject()` method of `UnknownElement`.

We tried to catch the relationship between those 5 classes in Figure 3. Besides these 5 key classes, the design documentation also mentions five other (helper)classes:
`IntrospectionHelper`
`ProjectHelper2`
`ProjectHelperImpl`
`ElementHandler`
`Main`

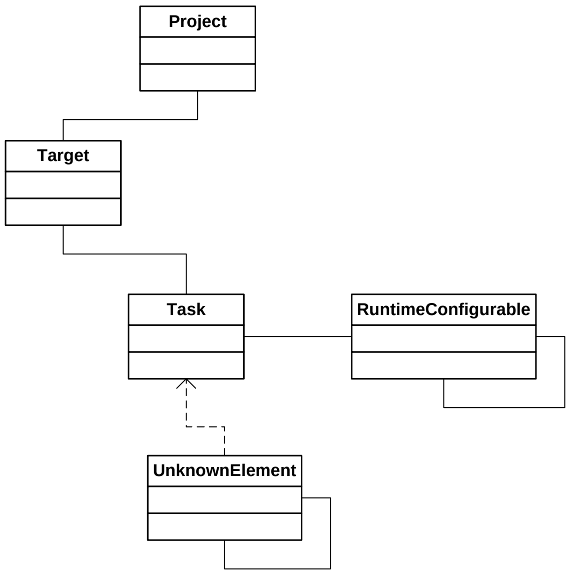

Figure 3. Simplified class diagram of Apache Ant.

[2] http://ant.apache.org/

[3] The design documentation of Ant can be found at: http://codefeed.com/tutorial/ant_config.html

Discussion of results. We applied our technique to the Apache Ant case study. Table 2 lists the results of this experiment. It has the following 3 columns with results:

- Column 1: shows the 15% highest ranked classes according to their hubiness.
- Column 2: shows the 15% highest scoring classes according to the CBO metric.
- Column 3: shows *all* the classes deemed important by the Ant development team. According to the developers these classes and their relationships need to be understood before beginning any (re)engineering operation on the project.

Class	Proposed algorithm	CBO	Ant docs
Project	√	√	√
UnknownElement	√		√
AntTypeDefinition	√		
Task	√	√	√
ComponentHelper	√	√	
Main	√	√	√
IntrospectionHelper	√	√	√
AbstractFileSet	√	√	
ProjectHelper	√	√	√
RuntimeConfigurable	√	√	√
SelectSelector	√		
DirectoryScanner	√		
Target	√		√
TaskAdapter	√		
ElementHandler	√		√
FileUtils		√	
BaseSelectorContainer		√	
XMLCatalog		√	
AntClassLoader		√	
FilterChain		√	
ChainReaderHelper		√	
Path		√	
TaskContainer			√

Table 2. Correlation between hubiness, static coupling, and expert opinion for Apache Ant.

Table 2 shows that:

- The number of *false positives*, i.e. classes reported but not considered important by the developers, is 6 out of 15 (40%) four our proposed technique. In the case of the CBO metric this amounts to 9/15 (60%).
- *False negatives*, i.e. classes mentioned in the documentation but not identified by our technique, on the other hand remain limited to just 1 out of 10 (10%) for the webmining approach. For the CBO metric this number equals 4 out of 10 (40%).

The number of false negatives can be considered very low and shows the value of using our technique. The number of false positives however is – at first sight – alarmingly high. This can amongst others be attributed to the fact that the classes reported by our technique should still be considered important, albeit less important than those mentioned in the design documents.

Close inspection of the project's source code reveals that the results of our technique can in fact be explained by this. Most classes that are highly-ranked through their hubiness are in fact classes that have a *coordinating role* in the system and as such make them interesting for program comprehension purposes.

Furthermore, Table 2 shows there is a big difference in precision with regard to the CBO metric.

5.2 Jakarta JMeter

Introduction. Our second case study involved Apache Jakarta JMeter 2.0.1[4]. JMeter is a Java application designed to load-test functional behavior and measure performance. It is frequently used for testing webapplications, but it can also handle SQL queries through JBDC and plugins can be written for other network protocols.

JMeter is part of this study because of its extensive documentation. On top of the general-purpose *javadoc* documentation, there are also documents that contain information on architectural design-choices, architectural evolution, etc.[5] This kind of documentation is helpful for (1) validating the results and (2) making the experiment repeatable.

Execution scenario. The execution scenario for this case study consisted of testing a HTTP (HyperText Transfer Protocol) connection to *Amazon.com*, a well-known online shop. More precisely, we configured JMeter to test the aforementioned connection 100 times and visualize the results in a simple graph. Running this scenario took 82 seconds.

Architectural overview. What follows is a brief description of the innerworkings of JMeter:
The `TestPlanGUI` is the component of the user-interface that lets the end user add and customize tests. Each added test resides in a `JMeterGUIComponent` class. When the user has finished creating his or her `TestPlan`, the information from the `JMeterGUIComponents` is

[4]http://jakarta.apache.org/jmeter/index.html

[5]The design documentation can be found on the Wiki pages of the Jakarta JMeter project: http://wiki.apache.org/jakarta-jmeter

extracted and put into `TestElement` classes.
These `TestElement` classes are stored in a tree-like datastructure `JMeterTreeModel`. This datastructure is then passed onto the `JMeterEngine` which, with the help os the `TestCompiler` creates `JMeterThread`(s) for each individual test. These `JMeterThread`s are grouped into logical `ThreadGroups`. Furthermore, for each test a `TestListener` is created: these catch the results of the threads carrying out the actual tests.

As such, we've identified nine key classes from the JMeter documentation. The design documentation also mentions a number of important helper-classes, being:
`AbstractAction`
`PreCompiler`
`Sampler`
`SampleResult`
`TestPlanGui`

Discussion of results. For the JMeter case study we calculated the *hubiness* of the compacted call graph taking into account the weights on the edges of the compacted call graph. The results of this effort are depicted in Table 3.

Just as we did for the previous case study, we count the number of false positives and false negatives:

- The number of *false positives*, i.e. classes reported but not considered important by the developers, is 8 out of 21 (38%) four our proposed technique.. In the case of the CBO metric this amounts to 16/21 (76%).

- *False negatives*, i.e. classes mentioned in the documentation but not identified by our technique, on the other hand remain limited to just 1 out of 14 (7%) for the webmining approach. For the CBO metric this number equals 9 out of 14 (64%).

As is the case for the Apache Ant case study, the number of false positives for Jakarta JMeter can be considered very low. The number of false positives is slightly lower than for our previous case study, but is still considered high. The reasons for this high number of false positives can again be attributed to the subjectiveness of what should be considered an important class (see also Section 5.1).

These results support our findings from the Apache Ant case study.

5.3 Interpretation of results

The results of our two case studies are very similar: the percentages of false positives and false negatives are around 40% and 10% respectively. As such, both our case studies support our initial hypothesis that important classes in a system are the classes that exhibit a high degree of

Class	Proposed algorithm	CBO	JMeter design docs
ArgumentsPanel		√	
CompoundVariable		√	
FunctionHelper		√	
FilePanel		√	
FileReporter		√	
GuiPackage		√	
JMeter		√	
JMeterMenuBar		√	
JMeterTest		√	
JMeterTreeListener		√	
MainFrame		√	
MenuFactory		√	
NamePanel		√	
SimpleConfigGui		√	
ValueReplacer		√	
AbstractAction	√	√	√
JMeterEngine	√	√	√
JMeterTreeModel	√	√	√
TestPlanGui	√	√	√
ResultCollector	√	√	
GenericController	√		
Graph	√		
ListenerNotifier	√		
MonitorPerformancePanel	√		
MultiProperty	√		
TestElementProperty	√		
Visualizer	√		
TestElement	√		√
JMeterThread	√		√
PreCompiler	√		√
Sampler	√		√
SampleResult	√		√
TestCompiler	√		√
TestElement	√		√
TestPlan	√		√
ThreadGroup	√		√
JMeterGuiComponent		√	√

Table 3. Correlation between hubiness, static coupling, and expert opinion for Jakarta JMeter.

hubiness in the compacted call graph.

Within our heuristical approach we accept that the number of false positives is fairly high. The number of false negatives, however, is very low and as such, not many important classes are missed by our heuristic. As such, we believe that our heuristic is well-suited for providing a quick answer to the software engineer as to which classes should be looked first at when trying to understand a program.

Furthermore, we believe that the heuristic we present here, offers an opportunity to software (re)engineers to become familiar with the software they need to understand in a more efficient way.

6 Related work

Tourwé and Mens [24] describe an experiment in which formal concept analysis is used to mine for *aspectual views*. An aspectual view is a set of source code entities, such as class hierarchies, classes and methods, that are structurally related in some way, and often crosscut a particular application. These aspectual views are used for aspect mining, but also for program comprehension purposes.

El-Ramly Et Al [7] describe a datamining technique for detecting interaction patterns in run-time behavior. Their initial focus is mainly on finding interaction patterns between graphical user interface components as their reengineering mission is a migration from a classical GUI to a web-based interface.

A novel solution has been formulated by Hamou-Lhadj and Lethbridge [11]. They represent the event trace as a tree in which they search neighbouring isomorphic subtrees. Identical neighbouring subtrees are pruned and replaced with a single occurrence which gets annotated with the total number of occurrences of the subtree. As such, they are able to present the user with a compacted trace, undone of loops.

7 Conclusion and future work

In this paper, we proposed a technique that uses web-mining principles for uncovering important classes in a system's architecture. We believe that the automatic classification of classes w.r.t. their importance is a critical step in alleviating the software engineer's program comprehension task. By allowing him/her to start his/her reconnaissance of the software from important classes can result in a tangible time-efficiency increase.

In the future, we will pursue the idea of applying datamining techniques to uncover important trends and relations in dynamic traces. First of all, we will continue the work on the identification of uncovering important classes. In the future we want to explore the connections and differences with other, dynamic or static, coupling metrics.

Besides the application of the HITS algorithm, there are many other datamining techniques that might help the analysis of large event traces. Especially because of the potentially large scale of event traces, the use of scalable datamining techniques seems very promising. The following datamining techniques are good candidates for helping the analysis of large event traces:

- Besides the hubs and authorities framework, there are many other graph mining concepts that can be interesting in the context of event traces. For example, Pagerank [1] is another method for ranking pages according to importance. Also the identification of web communities might prove useful in identifying classes or methods that are intimately connected.
- It can be interesting to find frequently occurring sequences in event traces. This problem might be solved by applying episode mining algorithms.

As can be seen from this list of candidates, the possibilities for applying datamining for automating program understanding are numerous. For an overview of the datamining techniques, see [12]. We believe this approach is very promising, and therefore think it should be explored further.

8 Acknowledgments

This work has been sponsored by the Belgian National Fund for Scientific Research (FWO).

Further sponsorship comes from the Institute for the Promotion of Innovation by Science and Technology in Flanders under grants Architectural Resources for the Restructuring and Integration of Business Applications (ARRIBA).

References

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactons on Software Engineering*, 20(6):476–493, 6 1994.

[3] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, Jan. 1990.

[4] T. Corbi. Program understanding: Challenge for the 90s. *IBM Systems Journal*, 28(2):294–306, 1990.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM*, pages 602–611, 2001.

[7] M. El-Ramly, E. Stroulia, and P. Sorenson. From run-time behavior to usage scenarios: an interaction-pattern mining approach. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 315–324. ACM Press, 2002.

[8] H. Erdogmus and O. Tanir, editors. *Advances in Software Engineering: Comprehension, Evaluation, and Evolution*, chapter 3. Studies of the Work Practices of Software Engineers, pages 50–74. Springer-Verlag, 2001. (Authors of Chapter 3: T. Lethbridge and J. Singer).

[9] M. A. Foltz. Dr. jones: A software archaeologist's magic lens. http://citeseer.nj.nec.com/457040.html.

[10] D. Gibson, J. M. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *UK Conference on Hypertext*, pages 225–234, 1998.

[11] A. Hamoe-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls, 2003. Workshop on Dynamic Analysis.

[12] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[13] I. Jacobson. *Object-Oriented Software Engineering: a Use Case driven Approach*. Addison–Wesley, Wokingham, England, 1995.

[14] J. H. Jahnke and A. Walenstein. Reverse engineering tools as media for imperfect knowledge. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 22–31. IEEE, 2000.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[16] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[17] A. Lakhotia. Understanding someone else's code: Analysis of experiences. *Journal of Systems and Software*, pages 269–275, Dec. 1993.

[18] D. Ng, D. R. Kaeli, S. Kojarski, and D. H. Lorenz. Program comprehension using aspects. In *ICSE 2004 Workshop WoDiSEE'2004*, 2004.

[19] N. Pennington. Comprehension strategies in programming. In *Empirical studies of programmers: second workshop*, pages 100–113. Ablex Publishing Corp., 1987.

[20] R. Smith and B. Korel. Slicing event traces of large software systems. In *Automated and Algorithmic Debugging*, 2000.

[21] D. Spinellis. *Code Reading: The Open Source Perspective*. Addison-Wesley, 2003.

[22] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

[23] E. Stroulia and T. Systä. Dynamic analysis for reverse engineering and program understanding. *SIGAPP Appl. Comput. Rev.*, 10(1):8–17, 2002.

[24] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *Proceedings of SCAM Workshop*. IEEE, 2004.

[25] A. von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 10(8):44–55, Aug. 1995.

[26] N. Wilde. Faster reuse and maintenance using software reconnaissance, 1994. Technical Report SERC-TR-75F, Software Engineering Research Center, CSE-301, University of Florida, CIS Department, Gainesville, FL.

[27] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, 1992.

[28] I. Zayour and T. C. Lethbridge. Adoption of reverse engineering tools: a cognitive perspective and methodology. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 245–255, 2001.