# Selective Introduction of Aspects for Program Comprehension

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

*We propose a technique that uses webmining principles on event traces for uncovering important classes in a system's architecture. These classes can form starting points for the program comprehension process. Furthermore, we argue that these important classes can be used to define pointcuts for the introduction of aspects. Based on a medium-scale case study – Apache Ant – and detailed architectural information from its developers, we show that the important classes found by our technique are prime candidates for the introduction of aspects.*

## 1 Introduction

Program comprehension is the process of understanding a system through feature and documentation analysis [11]. Gaining understanding of a program is a time-consuming task taking up to 40% of the time-budget of a maintenance operation [15]. 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, ... [10]

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. 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 [12]. 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 [4]. To comprehend the application, they build mental models by evaluating program code against their general programming knowledge [11].

For large industrial-scale systems, the program comprehension process requires the inspection and study of a significant number of packages, classes and code. As such, a semi-automated process in which an analysis tool supports the identification of key classes in a system's architecture and presents these to the user suits the hybrid cognitive model that is frequently used in large-scale systems [11].

Program understanding can be attained by using one of several 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. 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 (1) build up understanding of the software, and (2) locate candidate introduction points for aspects.

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 [9]. Hence, the Internet is viewed as a large graph. We verify that important classes in the pro-

gram correspond to the hubs in the dynamic call-graph of a program trace.

We apply the proposed technique to a medium-scale case study, namely Apache Ant. The results show that the *hubiness* is indeed a good measure for finding important classes in the system's architecture. Furthermore, based on these results we verify the hypothesis that these classes are good candidates for aspect introduction.

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 explains the datamining algorithm in detail, while in Section 4 the results of applying our technique on the case study are discussed. Section 5 explores related work, while Section 6 points to future research and concludes the paper.

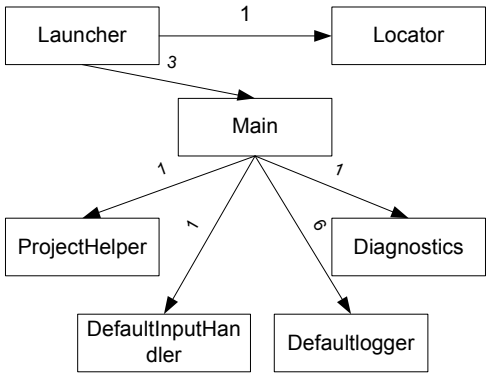


**Figure 1. A compacted call graph.**

## 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. Furthermore, by keeping the execution scenario specific, i.e., only calculating the interest rate, the final results will be more precise.

**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 (for our case study the trace consisted of 24 270 064 events for an execution time of 23s).

**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 are classes that call upon 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 unique messages that are sent between instances of both classes.
This compacted call graph is the input to the datamining algorithm that is presented in detail in section 3.

**Selective introduction of aspects.** The goal we wish to attain is guiding the software engineer through the software in order to help 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 program understanding purposes.
To the user, these starting points can be:

- Pointers to classes: the user should begin his/her investigation from these classes and analyze them and their collaborating classes manually.
- A visualization, e.g. an interaction diagram, of the classes deemed important and their immediate collaborators. This set of classes can e.g. be found by introducing aspects with the `cflow` pointcut designator [8] on all classes deemed important.

As a side effect of this heuristical detection of important classes, we expect to find opportunities for aspect refactorings [14, 13].

As validation we propose to verify whether the classes our technique marks as important are also deemed important

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

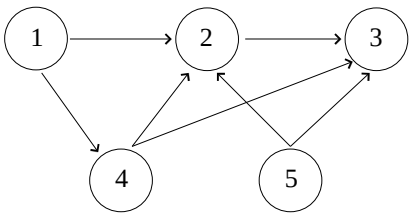


**Figure 2. Example web-graph**

by the developers. Furthermore, we will also compare the importance of these classes with the *Coupling Between Objects* (CBO) metric [3]. CBO can be seen as a typical static coupling measure which can help in identifying classes with a coordinating role.

# 3 Applying webmining techniques for program comprehension

In datamining, many successful techniques have been developed to analyze the structure of the web [2, 5, 9]. 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 [9] 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 the introduction of aspects as well.

## 3.1 Identifying hubs in large webgraphs

In [9], 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 $1 \rightarrow 2$ and $2 \rightarrow 3$, and 1 is a better hub than 2. 4 and 5 are better hubs than 1, as they point to better authorities.

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 $i \rightarrow j$ 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.

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

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

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 [9].

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 $h_i = \sum_{i \rightarrow j} w[i,j] \cdot a_j$ and $a_j = \sum_{i \rightarrow j} w[i,j] \cdot h_i$.

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

| | | | | | |
|---|---|---|---|---|---|
| $h_1$ | = | 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 |

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.

## 3.2 Identifying aspect candidates

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 the introduction of aspects to monitor the runtime behavior of these coordinators.

Furthermore, by using the `cflow` pointcut designator, we are not only able to monitor these coordinating classes, but also the classes that get their orders from these coordinators. This strategy can furthermore be used for efficient dynamic slicing.

## 4 Case study – Apache Ant

Ant is an XML based Java build tool. We chose Apache Ant 1.6.1 because we consider it to be a medium-size program (98 681 LOC, 127 classes) 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[2]. This knowledge will help us in validating our technique.

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.

We applied our technique two times on our case study. The first time, we set the weights of the compacted call graph all to 1, for the second experiment we used as weights the number of methods called upon from another class; see also Section 2.

In Table 1 we list the result of the first experiment. We show the highest 15% of classes according to their hubiness. We compare these classes with the CBO metric and with the opinion of the Ant development team.

Figure 1 shows that:

- The number of *false positives*, i.e. classes reported but not considered important by the developers, is 6 out of 15 (40%). In the case of the CBO metric this amounts to 7/12 (58%).
- *False negatives* on the other hand remain limited to just 1 out of 10. For the CBO metric this number equals 5 out of 10.

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 be attributed to several facts:

1. the developers opinion is *subjective* and only mentions those classes (or constructions) they are most proud of or they themselves find most interesting.
2. the classes our technique finds should also be considered important, albeit less important than those mentioned in the design documents.

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

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

**Table 1. Correlation between hubiness, static coupling, and expert opinion.**

Close inspection of the project's source code reveals that the results can be explained by a mixture of the above reasons. All 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 1 shows there is a big difference in precision with regard to the CBO metric.

The results of the second experiment, where we used the real weights calculated during the transformation from a call graph to a compacted call graph, are very similar. The important classes are now however not strictly in the upper 15%, but more in the upper 25%. Furthermore, a number of helper classes to the classes deemed important, now also have a high degree of hubiness. This comes from the fact that many of these helper classes make use of only a limited number of classes, but do use a lot of different methods. Hence, these helper classes do not use many other classes, but the ones they do use, are used very intensively. This intensity results in a large weight, which, on its turn, increases the relative hubiness.

Keeping this in mind, we advocate the use of the `cflow` pointcut on the important classes of the experiment with the weights set to 1. This way, the helper classes will also be

touched by the pointcut.

## 5 Related work

Tourwé and Mens [13] 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.

Breu and Krinke experimented with finding sets of methods that are always executed in the same sequence [1]. They argue that the found sets of classes are candidates for aspect refactoring.

## 6 Conclusion and future work

In this paper, we proposed a technique that uses webmining 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 the identification of aspects candidates. A case study showed that the approach is promising.

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 [2] 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.

- The event trace is in fact a large call tree. There exist tree mining algorithms that search for frequent occurring subtrees. The identification of such subtrees allows for compacting the presentation of the event trace [6].

- 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 [7]. We believe this approach is very promising, and therefore think that it can become an important research direction.

## References


[1] S. Breu and J. Krinke. Aspect mining using dynamic analysis, 2003.

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

[3] 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.

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

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

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

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

[8] 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.

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

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

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

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

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

[14] A. Van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of REFACE03*, 2003.

[15] 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.