Preprint of paper to appear in the Proceedings of the International Conference on Program Comprehension (ICPC'06)

How Webmining and Coupling Metrics Improve Early Program Comprehension

Andy Zaidman, Bart Du Bois, and Serge Demeyer

Lab On Reengineering (LORE)
University of Antwerp, Belgium
{Andy.Zaidman, Bart.DuBois, Serge.Demeyer}@ua.ac.be

Abstract

During initial program comprehension, software engineers could benefit from knowing the most need-to-be-understood classes in the system under study in order to kick-start their software reconnaissance.

Previously we have used webmining techniques on runtime trace data to identify these important classes. Here, we reprise this webmining technique and make a thorough comparison of its effectiveness when collecting static information of the software system under study.

Apache Ant and Jakarta JMeter, two medium-scale open source Java software systems, serve as case studies. From publicly available developers notes we conclude that the webmining technique in combination with dynamic analysis provides the best results with a level of recall of 90% when comparing with the developers' opinion.

1 Introduction

In [7] we proposed a heuristic that helped software engineers during the early phases of program comprehension by identifying need-to-be-understood classes, i.e. classes which are essential before any meaningful change-operation can take place. This heuristic is based on collecting trace information from program runs and applying a webmining technique on the resulting graph. The experiments were carried out on two medium-scale open-source case studies, Apache Ant and Jakarta JMeter. In [6], we reprised the same technique and applied it on a large-scale commercial software system. During these experiments however, we noticed that the use of dynamic analysis is often not so evident, mainly due to constraints such as the unavailability of tracing mechanisms in legacy environments (e.g. non-ANSI C on UnixWare) and the size of the event trace of larger scale systems (e.g. 90 GB).

These constraints have made us add a third leg to this research: we propose to apply webmining techniques on a static topological structure of the application. We instantiate this static topological structure with a number of static coupling metrics that we will compare with the results we have obtained from using dynamic information and with the developers' opinion. By using static information, we hope to overcome a number of constraints of dynamic analysis, namely (1) the necessity of a good execution scenario, (2) the availability of a tracing mechanism and (3) scalability issues such as the size of the trace file or the overhead from the tracing mechanism.

2 Webmining

Webmining deals with analyzing the structure of the *world wide web*. Webmining solutions see the web as a large graph, where each node represents a page and each edge represents a hyperlink. Using this graph as an input, the algorithm identifies so-called *hubs* and *authorities* [4]. Hubs are pages that refer to other pages containing information rather than being informative themselves, while a page is called an authority if it contains useful information.

Software systems can also be represented by graphs. As such, we found it worthwhile to try and reach our goal of identifying important classes in a system through the HITS webmining algorithm [4, 7].

The basis for this technique is the measurement of runtime coupling between modules of a system. Modules that have a high level of runtime *import* coupling, are often modules that contain important control structures, and request other modules to do work for them. As such, these are ideal candidates to study during early program comprehension.

Coupling measurements typically do not take into account *indirect coupling*. With the help of the iterative recursive algorithms such as HITS webmining, this indirect relationship can be taken into account.

Webmining more formally Every node i in the graph 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 calling relationship between modules i and j, and let $w[i,j]$ be the weight of the calling relationship between the modules i and j. The recursive relation between authority and hubiness is captured by:

$$h_i = \sum_{i \rightarrow j} w[i,j] \cdot a_j \quad (1)$$

$$a_j = \sum_{i \rightarrow j} w[i,j] \cdot h_i \quad (2)$$

For an example, see [7].

3 Coupling measurements

Intuitively, *coupling* refers to the degree of interdependence between parts of a software system's design, while *cohesion* refers to the internal consistency within parts of the design [3]. In object-oriented designs cohesion is desirable, while coupling is undesirable. Although coupling should be minimized, a certain degree of coupling is unavoidable as classes collaborate in certain ways in order to provide the required functionality. The definition from Wand [3, 5] captures the concept of *coupling*:

> Two things are coupled if and only if at least one of them "acts upon" the other. X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a thing traverses in time.

3.1 Dynamic coupling

Recently a framework for dynamic coupling metrics was set up by Arisholm et al [1]. This framework defines a set of 12 dynamic coupling metrics for object-oriented software.

The coupling measurement we used in our experiments in [7] can be translated into the dynamic coupling terminology of Arisholm, see Table 1. Having calculated the IC_CC' (Import Coupling, Class level, Distinct Classes [1]) metric, we can build up the so-called compacted callgraph [7]. The algorithm works as follows:

C	Set of classes in the system.
M	Set of methods in the system.
R_{MC}	$R_{MC} \subseteq M \times C$ Refers to methods being defined in classes.
IV	$IV \subseteq M \times C \times M \times C$ The set of possible method invocations.
IC_CC'(c_1, c_2) =	# {(m_2, c_1, c_2) | ($\exists$ (m_1, c_1), (m_2, c_2) $\in R_{MC}$) $\wedge$ $c_1 \neq c_2$ $\wedge$ (m_1, c_1, m_2, c_2) $\in$ IV }

Table 1. Dynamic coupling measure [1].

1. For each class that participated in the execution scenario, add a node to the graph.
2. For each pair (c_1, c_2):
   - if IC_CC'(c_1, c_2) > 0 add an edge $c_1 \rightarrow c_2$ with weight IC_CC'.
   - if IC_CC'(c_1, c_2) == 0, do nothing.

3.2 Static coupling

Previously we validated the results of the dynamic coupling metric we presented in Section 3.1 against the *Coupling Between Objects* (CBO) metric [7, 2, 3]. CBO however, is often ill-defined [2] and the tool we used for determining CBO previously, did not mention the exact metric-implementation. As such, for this experiment, we wanted to have more control over the static metric calculation. Furthermore, we use static coupling metrics that are as close as possible to the IC_CC' metric we used in the previous section.

The framework from Arisholm does not need to make a distinction between static and polymorphic calls due to the dynamic nature of its measurements. We add notations from the unified framework for object-oriented metrics from Briand et al. [2] to ensure that the notation as used by Arisholm still holds for our purposes. Some helpful definitions are:

Definition1 Methods of a Class
For each class $c \in C$ let $M(c)$ be the set of methods of class c.

Definition 2 Declared and Implemented Methods.
For each class $c \in C$, let:

- $M_D(c) \subseteq M(c)$ be the set of methods declared in c, i.e., methods that c inherits but does not override or virtual methods of c.
- $M_I \subseteq M(c)$ be the set of methods implemented in c, i.e., methods that c inherits but overrides or nonvirtual noninherited methods of c.

Definition 3 $M(C)$. The Set of all Methods.
$M(C) = \cup_{c \in C} M(c)$

Definition 4 $SIM(m)$. Set of Statically Invoked Methods of m. Let $c \in C$, $m \in M_I(c)$, and $m' \in M(C)$. Then $m' \in SIM(m)$ $\Leftrightarrow$ $\exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' is invoked for an object of static type class d.

Definition 5 $NSI(m, m')$. The Number of Static Invocations of m' by m. Let $c \in C$, $m \in M_I(c)$, and $m' \in SIM(m)$. $NSI(m, m')$ is the number of method invocations in m where m' is invoked for an object of static type class d and $m' \in M(d)$.

Definition 6 $PIM(m)$. The Set of Polymorphically Invoked Methods of m. Let $c \in C$, $m \in M_I(c)$, and $m' \in M(C)$. Then $m' \in PIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of m has a method invocation where m' may, because of polymorphism and dynamic binding, be invoked for an object of dynamic type d.

Definition 7 $NPI(m, m')$. The Number of Polymorphic Invocations of m' by m. Let $c \in C$, $m \in M_I(c)$, and $m' \in PIM(m)$. $NPI(m, m')$ is the number of method invocations in m where m' can be invoked for an object of dynamic type class d and $m' \in M(d)$.

```
1 public void foo() {
2   BaseClass base = new BaseClass();
3   base.doSomething();
4   base.doSomething();
5 }
```

Figure 1. Code snippet to explain metrics.

These notational constructs help us to write down four static coupling measures that closely resemble the measurements that were defined in Section 3.1.

The fact that one dynamic metric IC_CC' is translated into 4 static metrics can be explained by the fact that the static environment offers some degrees of choice when calculating the metrics. Consider Figure 1:

- The choice between *static calls* and *polymorphic calls*. In other words when considering Figure 1, do we only count the reference to `BaseClass` or also to all subclasses of `BaseClass`?
- Do we count duplicate calls for the same (origin, target) pairs? When considering Figure 1 do we count the `base.doSomething()` call one or twice?

Definition SM_SO Static Metric, Static calls, count every Occurrence of a call only once.

$SM\_SO(c_1, c_2) = \#\{(m_2, c_2, c_1) | \exists\, (m_1, c_1), (m_2, c_2) \in R_{MC} \wedge c1 \neq c2 \wedge\ (m_1, c_1, m_2, c_2) \in IV \wedge\ m_2 \in SIM(m1)\}$

Definition SM_SW Static Metric, Static calls, count every occurrence of a call (Weighted).

$SM\_SW(c_1, c_2) =$ identical to $SM\_SO(c_1, c_2)$, but { } should be interpreted as *bag* or *multiset*.

Definition SM_PO Static Metric, Polymorphic calls, count every Occurrence of a call only once.

$SM\_PO(c_1, c_2) = \#\{(m_2, c_2, c_1) | \exists\, (m_1, c_1), (m_2, c_2) \in R_{MC} \wedge c1 \neq c2 \wedge\ (m_1, c_1, m_2, c_2) \in IV \wedge\ m_2 \in PIM(m1)\}$

Definition SM_PW Static Metric, Polymorphic calls, count every occurrence of a call (Weighted).

$SM\_PW(c_1, c_2) =$ identical to $SM\_PO(c_1, c_2)$, but { } should be interpreted as *bag* or *multiset*.

4 Reverse engineering process

In [7] we explained that applying our webmining heuristic in combination with dynamic analysis is a 4-step process, (1) starting with the definition of an execution scenario, over (2) calculating the basic metrics and (3) applying the webmining algorithm on them, to (4) interpreting the resultset. This approach is characterized by a number of drawbacks, which we will now discuss in some more detail:

- The necessity of a *well-covering execution scenario*. This is not always evident as one needs knowledge of how to operate the application and the execution scenario can sometimes take several hours.
- The availability of a *tracing mechanism* for the language/platform. For interpreted languages such as Java, C# and Smalltalk, generic tracing mechanism exist, but in the case of e.g. Cobol or ANSI C, one often has to revert to custom-made solutions such as those presented in e.g. [6].
- *Scalability issues* of storing and analyzing the resulting trace file. Our dynamic analysis experiments with Apache Ant and Jakarta JMeter resulted in tracefiles of 2 and 0.6 GB respectively [7]. Larger-scale industrial case studies however have resulted in trace files of around 90GB [6].

When we look at the same process in this static experiment, we see the following 3-step process:

1. Calculate the metrics. In our case we import the project into Eclipse and used the JDT2MDR[1] plugin to calculate our specific metrics. This plugin transforms the project to a graph representation closely resembling the metamodel employed by Briand et al. [2], thereby enabling the calculation of the coupling measures formalized in their paper.
2. Apply the HITS webmining algorithm.
3. Interpret the resultset.

The resultset that is presented to the user is a list of classes ranked according to their relative importance according to the webmining heuristic. By default, we only present the 15% most highly ranked classes, the reasoning behind this is as follows:

- From the documentation of both Apache Ant and Jakarta JMeter we have learned that about 10% of the classes of the systems need to be understood before any meaningful change operation can take place. As we are working with a heuristical technique we took a 5% margin.
- For cognitive reasons, the size of the resultset should be kept at a minimum.
- Empirically, we found that lowering the threshold to the top 20% classes, did not result in an increase in recall. That is, we did not notice any classes mentioned in the documentation showing up in the interval [15%, 20%] [7]. We again verify this empirically with the resultset presented in the next section.

Validation As validation we use the concepts of *recall* and *precision*. Each resultset obtained will be compared to the baseline that is comprised of the developers' opinion.

Threats to validity Comparing static and dynamic analysis poses some threats to the validity of our experimental setup. When considering the 15% most highly ranked classes, the size of this 15% resultset varies according to the size of the inputset, namely the number of classes. In

[1] mailto: bart.dubois@ua.ac.be

the case of the static process, the size of the inputset equals the total number of defined classes, while in the dynamic process, this equals the number of classes that participate in the execution scenario(s).

5 Results

Apache Ant[2] and Jakarta JMeter[3] are two software projects we previously used for our experiments [7].

5.1 Apache Ant 1.6.1

Table 2 gives an overview of the results we have obtained from calculating the coupling metrics from Section 3 and applying the HITS webmining algorithm on them. Classes that belong to the 15% highest ranked classes are marked with a $\surd$, while other classes are linked to their place in the ranking.

As a general observation, we note that recall results of the dynamic experiment (90%) are noticeably higher then any other result, while precision is also higher. The fact that precision for the 4 static metrics is much lower (8% or less, compared to 60%) can be explained by the size of the inputsets, as the inputset for the static experiment was 403 classes, while for the dynamic experiment this was only 127 classes. As such, when using our 15% highest ranked classes rule of thumb for determining the final size of the resultset, we end up with 60 and 15 classes respectively.

When considering the precision of the 4 static metrics, we observe that taking into account polymorphism provides a much better result. This can be explained by the fact that (1) sometimes a base class is abstract or (2) the base class is not always the most important class in the hierarchy. The second variation point, namely whether to only count an occurrence of a call once or count every occurrence of a call (weighted) does not seem to make any difference with regard to our specific context (small variations exist, but these do not influence our resultset).

A further point to be made is that when looking at the ranking of classes that fall outside the top 15%, lowering the bar to 20% would not have resulted in a (significant) gain in recall, while precision would drop further. Furthermore, by raising the bar to 10%, recall will generally fall with 10%.

5.2 Jakarta JMeter 2.0.1

Table 3 shows the results for the JMeter case. Considering that for the static approach there are 490 classes at the core of the JMeter application, the 15% highest ranked classes are ranked $1 \rightarrow 74$. A number of observations:

- The difference in recall between the dynamic metric and the static metrics is rather big, as the dynamic IC_CC' metric is able to recall 93% percent of classes deemed important by the developers, while the static metrics cannot achieve more than 43% recall.
- A similar difference can be noted with regard to the precision: a precision of 63% for the dynamic IC_CC' metric, while statically 8% is best case.
- The resultset for SM_SW and SM_SO is identical, while only minimal differences exist between SM_PW and SM_PO. In our opinion this is due to the fact that most method calls happen only once per method (no two identical calls in one method, also see Figure 1).
- There is a sizeable dissimilarity between the results obtained when only taking into account static calls versus also adding polymorphism (43% versus 7% recall).

Also, when considering the final resultset of the top 15% ranked classes, lowering the bar to 20% would not have resulted in a (significant) gain in recall.

Class	SM_PO	SM_PW	SM_SO	SM_SW	IC_CC'	Ant docs
Project	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
UnknownElement	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Task	79	81	119	120	$\surd$	$\surd$
Main	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
IntrospectionHelper	$\surd$	$\surd$	116	105	$\surd$	$\surd$
ProjectHelper	97	99	90	190	$\surd$	$\surd$
RuntimeConfigurable	$\surd$	$\surd$	63	63	$\surd$	$\surd$
Target	89	93	100	100	$\surd$	$\surd$
ElementHandler	192	198	125	125	$\surd$	$\surd$
TaskContainer	398	403	381	383	N/A	$\surd$
→ **recall** (%)	50	50	30	30	90	-
→ **precision** (%)	8	8	5	5	60	-

Table 2. Ant metric data overview.

Class	SM_PO	SM_PW	SM_SO	SM_SW	IC_CC'	JMeter docs
AbstractAction	275	275	336	336	$\surd$	$\surd$
JMeterEngine	$\surd$	$\surd$	484	484	$\surd$	$\surd$
JMeterTreeModel	$\surd$	$\surd$	150	150	$\surd$	$\surd$
JMeterThread	$\surd$	$\surd$	147	147	$\surd$	$\surd$
JMeterGuiComponent	$\surd$	$\surd$	475	475		$\surd$
PreCompiler	362	362	293	293	$\surd$	$\surd$
Sampler	457	478	454	454	$\surd$	$\surd$
SampleResult	119	119	209	209	$\surd$	$\surd$
TestCompiler	$\surd$	$\surd$	145	145	$\surd$	$\surd$
TestElement	$\surd$	$\surd$	451	451	$\surd$	$\surd$
TestListener	450	443	449	449	$\surd$	$\surd$
TestPlan	113	113	234	234	$\surd$	$\surd$
TestPlanGui	93	93	$\surd$	$\surd$	$\surd$	$\surd$
ThreadGroup	140	140	157	157	$\surd$	$\surd$
→ **recall** (%)	43	43	7	7	93	-
→ **precision** (%)	8	8	1.4	1.4	62	-

Table 3. JMeter metric data overview.

[2]For more information, see: http://ant.apache.org

[3]For more information, see: http://jakarta.apache.org/jmeter/

6 Trade-off analysis

Although we are satisfied with the results from the dynamic approach in terms of recall and precision, there are a number of downsides to the dynamic analysis process. These three drawbacks, added with the quality of the resultset, will form the criteria during the trade-off analysis:

1. *The necessity of a good execution scenario.* When performing static analysis, having an execution scenario is no issue. However, access to the source code remains a necessity[4]. Static analysis is to be favored here.
2. *The availability of a tracing mechanism.* Although a tracing mechanism is no longer an issue, having a metrics engine remains a necessity. To implement such an engine, either open source tools need to be available or a parser needs to be constructed. Because a similar precondition exists for both processes, neither of the two approaches has an advantage here.
3. *Scalability issues.* In terms of scalability the dynamic process is plagued by the possibly huge size of the tracefile. This has consequences on multiple levels:
   - The I/O overhead on the traced application (e.g. for Ant: execution of 23 seconds without tracing versus just under one hour with tracing).
   - The size of the trace (2 GB in the case of Ant).
   - The time it takes to calculate the IC_CC' metric and perform the HITS webmining algorithm on this 2 GB of data. In the case of Ant this takes around 45 minutes.

   As such, the time-efficiency of the dynamic process is not optimal. Statically, however, our prototype metrics engine took one hour to calculate the metrics for Ant and slightly over one hour for JMeter. Applying the HITS algorithm takes less than one minute, so the total round trip time is around one hour for both projects. While these times are not so different from the dynamic process, the dynamic process still needs the tracing step, which makes that the round trip time for the dynamic process is significantly larger and in the case of Ant takes around two hours.
4. *Quality of the resultset in terms of recall and precision.* The dynamic approach has shown to be able to attain the best level of recall at 90%. Recall for the static metrics that take into account polymorphism (SM_P* ranges from 43 to 50% recall) is generally better than that of their counterparts that do not take into account polymorphism (SM_S* ranges from 7 to 30%).
   For precision, the situation is similar. Dynamically, a satisfactory 60% precision is reached. Statically however, precision remains a lowly 8%, due to the fact that the resultset, which we hold at the 15% highest ranked classes by the webmining algorithm, is larger because of the larger input set.

The balance clearly swings towards the dynamic approach. Although the static process has the advantage of (1) eliminating the need for defining an execution scenario and (2) having a shorter round trip time, it is severely let down by its results, mainly its lack of recall is worrying, which is one of the fortes of the technique when applied dynamically.

[4] Reverse engineering from binaries is sometimes possible.

7 Conclusion

This paper describes an experiment in which we try to detect "important classes", which are of use during initial program comprehension phases, because they are responsible for a reasonable amount of high level actions or services in at least one use case. The core of the experiment lies in the fact that we introduce four static metrics and compare them with an already existing dynamic coupling metric, that has already been used to identify important classes [7, 6]. The rationale behind this experiment was to try and overcome three drawbacks of the dynamic analysis approach, namely (1) the necessity of having a good execution scenario, (2) scalability issues inherent to dynamic analysis and (3) the necessity of having a tracing mechanism available.

Although static analysis resolves issues (1) and (2), issue (3) has a static counterpart, namely the availability of a (customizable) metrics engine (or at the very least the availability of a parser for the language). Realistically, static analysis is to be favored when considering the fact that it incurs much less overhead on the reverse engineering process and has shorter round trip time. However, when taking the results into account, specifically the level of recall, the balance clearly favors the dynamic solution. In our experiment with dynamic analysis, the HITS webmining algorithm is able to recall around 90%of the classes. Statically, recall remains at 50% or less.

References

[1] E. Arisholm, L. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE TSE*, 30(7):491–506, 2004.

[2] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE TSE*, 25(1):91–121, 1999.

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

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

[5] Y. Wand and R. Weber. An ontological model of an information system. *IEEE TSE*, 16(11):1282–1292, 1990.

[6] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and aspect orientation — an industrial experience report. In *CSMR*, pages 91–102. IEEE, 2006.

[7] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR*, pages 134–142. IEEE, 2005.