JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

Automatic Identification of Key Classes in a Software System Using Webmining Techniques

Andy Zaidman[1,*], Serge Demeyer[2]

[1] *Delft University of Technology, The Netherlands*
[2] *University of Antwerp, Belgium*

SUMMARY

Software engineers new to a project are often stuck sorting through hundreds of classes in order to find those few classes that offer a significant insight into the inner workings of the software project. To help stimulate this process, we propose a technique which can identify the most important classes in a system, or the key classes of that system. Software engineers can use these classes to focus their understanding efforts when starting to work on a new software project. Those *key classes* are typically characterized with having a lot of "control" within the application. In order to find these controlling classes, we present a detection approach that is based on dynamic coupling and webmining. We demonstrate the potential of our technique using two open source software systems, which have a rich documentation set. During the case studies we use dynamically gathered coupling information and vary between a number of coupling metrics. The case studies show that we are able to retrieve 90% of the classes deemed important by the orginal maintainers of the systems, while maintaining a level of precision of around 50%.

KEY WORDS: program comprehension, dynamic analysis, webmining, coupling

1. Introduction

Most successful software system are in a state of constant flux, evolving towards new business needs, higher performance, better reliability and perhaps even a better internal structure [1]. When this evolution is applied to a system, a software engineer who is not completely familiar

*Correspondence to: a.e.zaidman@tudelft.nl

Received 27 April 2007
Revised 29 April 2008

with the system that needs to be evolved, first needs to go through a process of acquiring enough knowledge about the system before making alterations [2, 3]. This process, which is termed the program comprehension process [4, 3], is known to take up between 30 and 60% of a software engineer's total allocated time [5, 6, 3]. When it comes to a definition of what program comprehension means, we adhere to the definition introduced by Biggerstaff et al. [7]:

> *"A person understands a program when able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."*

Although the manner in which a programmer builds up his understanding of a software system varies greatly [8, 9, 10, 3], we do realize that for large-scale software systems building up knowledge of that system is a daunting task. Just think of how difficult it can be to find your way in an unknown software system containing hundreds or thousands of classes: where do you need to start looking in order to understand part of the system? Knowing where to start looking, i.e., which classes are important, and from there on following links to other classes in order to understand the inner workings of an application, is certainly more time-efficient.

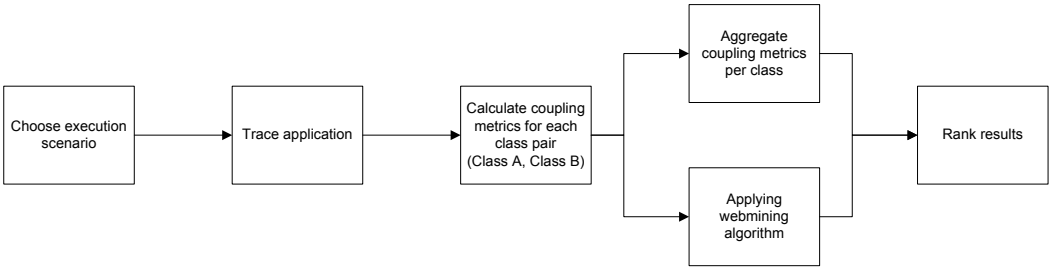
These starting-point classes often have a *controlling function* within the software system and they are typically characterized by the fact that they use a large amount of other classes to implement their functionality. However, the identification of these classes with a controlling function, so-called *key classes* is not so straightforward when working with an unfamiliar system. Other resources like documentation might be outdated and fellow software engineers might not know much about a specific application. In this light, we developed a heuristic approach that automatically identifies a set of candidate classes within a system that are prime candidates to be studied during initial program understanding.

In her research about design flaws, Tahvildari has also noticed these *key classes* [11]:

> *"These key classes are described as the classes that implement the key concepts of a system. Usually, these most important concepts of a system are implemented by very few key classes, which can be characterized by a number of properties. These classes which we called key classes manage a large amount of other classes or use them in order to implement their functionality. The key classes are tightly coupled with other parts of the system. Additionally, they tend to be rather complex, since they implement much of the legacy system's functionality."*

It is our goal to automatically detect these key classes. The observation from Tahvildari that these classes are characterized by being *tightly coupled*, made us build our key class identification technique around detecting tight coupling. Our specific angle is to focus on dynamic coupling, i.e., coupling information that was gathered from a running system. Two reasons instigate our choice for dynamic coupling, namely: (1) we expect that a higher level of precision can be obtained in the light of the abundant presence of polymorphism in object-oriented software systems and (2) by actually only collecting coupling metrics of specific execution scenarios we are able to follow a goal-oriented comprehension strategy, which will focus the comprehension process even more. Another important ingredient of our approach is

Figure 1. Overview of the approach.



the addition of *indirect coupling*, or coupling between two classes via a finite number of other classes. We add this notion of indirect coupling through the use of a webmining algorithm.

The contributions of this paper are:

- We propose a technique to automatically identify so-called key classes of a software system that can be useful for a software engineer who is trying to get a high-level overview of system that he is unfamiliar with.
- Our technique is based on the identification of tightly coupled classes, where we also take into account indirect coupling through the application of a webmining algorithm.
- The comparison of a number of dynamic and static coupling metrics for the purpose of identifying the coupling metric that is best suited for our purposes.
- A demonstration of our technique using two open source case studies. For both case studies we have extensive design documents from the original developers and maintainers of the software projects, which helps us in establishing a program comprehension baseline with which we are able to evaluate our retrieval technique.

The structure of this paper is as follows: Section 2 introduces our approach and provides detailed information on dynamic coupling metrics and webmining. Section 3 explains our experimental setup and talks about our case studies and research questions. Section 4 presents the results of applying dynamic coupling to our case studies, while Section 5 continues with static coupling. In Section 6 we discuss the overall results of our case studies, including threats to validity to our experimental setup. Section 7 contains related work, while Section 8 presents our conclusions and future work.

2. Approach

Our technique of automatically detecting the key classes of a software system is based on the combination of two principles, namely (1) the identification of tightly coupled classes and (2) also taking into account indirect coupling with the help of a webmining algorithm. Figure 1 shows an overview of the process of our approach. After defining an execution scenario, we

trace the application. Post-mortem we use the trace to calculate coupling metrics between individual classes. The next step has two alternatives, namely: (1) we simply aggregate the coupling metric values that we have calculated between individual classes on a per class basis or (2) if we want to take indirect coupling into consideration, we directly provide the metric values between individual classes as input to the webmining algorithm. A final step consists of ranking the results from strong coupling to weak coupling for each class in the result set, where the actual rank of the class serves as an indicator for its importance during initial program comprehension.

Sections 2.1 and 2.4 will elaborate on these techniques, while Section 2.5 will discuss how we combine both mechanisms.

2.1. Coupling

This section introduces coupling and reasons on the usefulness of coupling when trying to detect the key classes of a software system. We first introduce dynamic coupling metrics, after which we discuss static coupling metrics.

2.1.1. Introduction to coupling

Software systems are typically composed from several software entities — be it modules, classes, components, aspects,... These entities work together to reach their goal(s) and the collaborations that exist between these entities give rise to the notion of coupling. Wand defines coupling as [12]:

> 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.

Although software engineers are constantly striving to minimize coupling in order to improve, e.g., the understandability and reusability of software components [13], we intuitively understand that coupling will always exist within software systems, as classes need to work together to deliver the desired functionality [14].

2.1.2. Static and dynamic coupling

Coupling metrics have for some time now been subject of research, e.g., in the context of quality measurements [15]. These metrics have mostly been determined *statically*, i.e., based upon structural properties of the source code (or models thereof). However, with the wide-spread use of object oriented programming languages, these static coupling metrics lose precision as more intensive use of inheritance and dynamic binding occurs [16]. Another factor that possibly negatively influences the measurements is the presence of dead code, which can be difficult to detect statically in the presence of polymorphism.

This has led us to start looking at dynamic coupling metrics, a branch of software engineering research that has only recently been developing [16]. We propose the following working definition for dynamic coupling metrics:

Table I. Dynamic coupling classification [16].

Entity	Granularity (Aggregation Level)	Scope (Include/Exclude)	Direction
Object	Object Class (set of) Scenario(s) (set of) Use case(s) System	Library objects Framework objects Exceptional use cases	Import/Export
Class	Class Inheritance Hierarchy (set of) Subsystem(s) System	Library classes Framework classes	Import/Export

> Dynamic coupling metrics are defined based upon an analysis of interactions of runtime objects. We say that two objects are dynamically coupled when one object acts upon the other. Object x is said to act upon object y, when there is evidence in the execution trace that there is a calling relationship between objects x and y, originating from x. Furthermore, two classes are dynamically coupled if there is at least one instance of each class for which holds that they are dynamically coupled.

The basic framework we use when considering dynamic coupling metrics was first introduced by Arisholm et al. [16].

2.1.3. Classification of dynamic coupling metrics

Dynamic coupling can be measured in different ways. Each of the measures can be justified, depending on the application context where such measures are to be used [16]. Table I gives an overview of the variations. Each of the variations will also be discussed in this section.

1. **Entity of measurement**. Since dynamic coupling is calculated from dynamic data stored in the event trace, we can calculate coupling at the *object*-level or at the *class*-level.
2. **Granularity**. Orthogonal to the entity of measurement, dynamic coupling measures can be aggregated at different levels of *granularity*. Different kinds of aggregations can be made depending on the entity of measurement. Aggregations that can be made include: at the *(sub)system*, *inheritance hierarchy*, *use case* or *scenario* level.
3. **Scope**. Another variation can be the classes we want to consider when calculating the metric(s). For example, instances of library or framework classes can sometimes be of no special interest and as such they can be excluded.
4. **Direction (import or export)**. Consider two classes c and d being coupled by the invocation of a method m_2 of d in a method m_1 in class c. This relationship can be described as a client-server relationship between the classes: the client class c uses

(*imports* services), the server class d is being used (*exports* services). This situation gives rise to the concepts of import and export coupling.

2.1.4. Dynamic coupling for program comprehension

Based on the classification schema presented in Section 2.1.3 we will now discuss which properties we expect from a coupling metric in order to be useful for program comprehension purposes. Based on these properties, we will find those dynamic coupling metrics that suit our intentions best.

1. At a cognitive level, the software engineer trying to get a first impression of a piece of software, will try to comprehend the software at the *class-level*, as these are the concepts he/she can recognize in the source code, the documentation and the application domain.
2. As such we advocate either the use of classes as level of granularity or a further aggregation up to the (sub)component (or in other terms package) level.
3. A general purpose tracing mechanism usually traces everything, also low-level calls to libraries. In order to keep focus, we discard all classes foreign to the actual project (e.g., libraries), as they are not the target of the comprehension process. Furthermore, choosing a well-defined execution scenario of the software that involves exactly those features that the end-user wants to understand, is essential.
4. In Section 1 we already stated that we are looking for classes that have a prominent role within the system's architecture. We expect these classes to *give orders* to other classes, i.e., tell them what to do and what to give in return. In terms of the *direction* of coupling, this means that we are looking at *import* coupling. Vice versa, classes with strong export coupling are classes that provide services to other classes.

Arisholm et al. defined twelve dynamic coupling metrics; two of these adhere to the criteria we set out, namely: working at the class-level and measuring import coupling [16]. We will now discuss these two metrics.

1. *Distinct method invocations.* This measure counts the number of *distinct* methods invoked by each method in each object. This information is then aggregated for all the objects of each class. Arisholm et al. call this metric IC_CM (Import Coupling, Class level, Distinct Methods). Calls to methods from the same object or class (cohesion) are excluded.
2. *Distinct classes.* This measure counts the number of distinct server classes that a method in a given object uses. That information is than aggregated for all the objects of each class. Arisholm et al. call this metric IC_CC (Import Coupling, Class level, Distinct Classes). Calls to methods from the same object or class (cohesion) are excluded.

Consider the formal definitions of IC_CC and IC_CM in Table II.

Reconsider the IC_CC metric. When we are looking for a metric that points to classes that import a lot of services from other classes, we see that IC_CC has a limited range. IC_CC counts the number of (m_1, c_1, c_2) triples. Because the first component in this triple is m_1, the maximum metric value is the product of the number of methods in the definition of c_1 and the number of classes c_1 interacts with. Because the number of methods defined in c_1 plays a vital role in the calculation of this metric, this can become a limiting factor. Furthermore, it does not give a true reflection as to how many other classes and in particular methods in other classes are used.

Table II. Dynamic coupling measures [16].

Helper definitions

C	Set of classes in the system.
M	Set of methods in the system.
R_{MC}	$R_{MC} \subseteq M \times C$ The set of all methods that are actually defined in a class.
IV	$IV \subseteq M \times C \times M \times C$ The set of all possible method invocations.

Metric definitions

$IC\_CM(c_1) =$	$\mid \{(m_1, c_1, m_2, c_2) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC}\ ) \wedge c_1 \neq c_2 \wedge (m_1, c_1, m_2, c_2) \in IV\ \} \mid$
$IC\_CC(c_1) =$	$\mid \{(m_1, c_1, c_2) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC}\ ) \wedge c_1 \neq c_2 \wedge (m_1, c_1, m_2, c_2) \in IV\ \} \mid$
$IC\_CC'(c_1) =$	$\mid \{(m_2, c_1, c_2) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC}\ ) \wedge c_1 \neq c_2 \wedge (m_1, c_1, m_2, c_2) \in IV\ \} \mid$

Therefore, we made a variation on the IC_CC metric, called IC_CC′. This variation does not count the number of calling methods, but the number of called methods. In other words, triples of the form (m_2, c_1, c_2) are counted. This metric gives a more accurate reflection of the number of "services", i.e., distinct methods, that a class requests. A formal definition of IC_CC′ can be found in Table II.

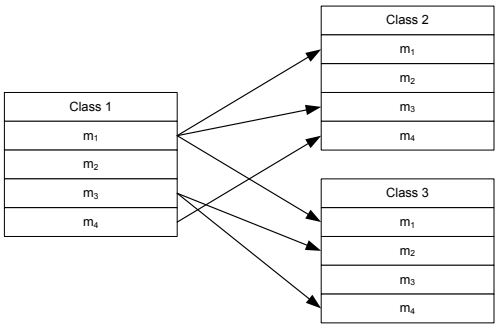
Example. Consider the three classes depicted in Figure 2. The IC_CC metric would yield a score of 4 for *class 1*, as the number of unique (m_1, c_1, c_2) triples is 4. For IC_CC′ on the other hand, the metric value for *class 1* is 6, which corresponds with the number of unique methods called in foreign classes (i.e., no cohesion). This example also shows that when a class contains only one or a limited number of very long methods (which is typical for "god classes"), that the IC_CC metric value is limited in its range, while the IC_CC′ metric's range is not influenced.

In the first phase of our case studies (see Section 4) we will make a thorough comparison of the effectiveness of the three aforementioned metrics.

2.2. Static coupling

In a previous experiment, we have compared these three dynamic coupling metrics for their effectiveness in detecting the key classes of a software system. In that comparison, we have also included the static *Coupling Between Objects* (CBO) metric [15, 17]. CBO however, proved to perform poorly against the dynamic coupling metrics, which instigated us to research static coupling metrics that are very close to the dynamic coupling metrics defined in the previous sections.

Figure 2. Comparison of IC_CC and IC_CC′



After performing the first phase of our case study in which we compare the IC_CM, IC_CC and IC_CC′ metrics, we take the best performing of these three metrics and define one or more static coupling metrics that are close to it. Sections 3.2 and 5 elaborate on the exact process that we follow.

2.3. Indirect coupling

Up until now we have talked about *direct* coupling. Direct coupling is a relationship between two entities. However, when considering large-scale software systems it is far from inconceivable that more than 2 entities influence each other. Reconsider the coupling definition from Wand (see Section 2.1.1) and let X, Y and Z be 3 entities where, respectively (X, Y) and (Y, Z) are directly coupled, i.e., X acts upon Y and Y acts upon Z. Intuitively, it is easy to understand that it is possible that X also (indirectly) acts upon Z, e.g., through parameter-passing and/or polymorphism (e.g., double-dispatch).

Based upon this observation, we investigate the notion of *indirect* coupling [18]. Briand et al. use the following definition [19]:

> Direct coupling describes a relation on a set of elements (e.g., a relation "invokes" on the set of all methods of the system, or a relation "uses" on the set of all classes of the system). To account for indirect coupling, we need only use the transitive closure of that relation.

The next section introduces the HITS webmining algorithm, which we will use for taking into account indirect coupling.

2.4. The HITS webmining algorithm

2.4.1. Introduction

Webmining, a branch of datamining research, deals with analyzing the structure of the world wide web [20, 21, 22]. Typically, webmining algorithms see the internet as a large graph, where each node represents a webpage and each edge represents a hyperlink between two webpages. Using this graph as an input, the algorithm allows us to identify so-called *hubs* and *authorities* [22]. Intuitively, on the one hand, hubs are pages that refer to other pages containing information rather than 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 and is referenced by others (e.g., web pages containing definitions, personal information, ...).

Software systems can also be represented by graphs, where classes are nodes and calling relationships between classes are edges. Furthermore, there is a "natural" extension to the concepts of *hubs* and *authorities* in the context of (object-oriented) software systems. Classes that exhibit a large level of import coupling call upon a number of other classes that do the groundwork. In order for them to control these assisting classes, they often contain important control structures. As such, they have a considerable amount of influence on the data and control flow within the application. Conceptually, the classes that have a high level of import coupling are similar to the hubs in web-graphs.

Export coupling on the other hand is a sign of very specific functionality, often frequently reused throughout the system. Because of their specificity, they are conceptually similar to authorities in web-graphs.

Because of this conceptual similarity, we found it worthwhile to try and reach our goal of identifying important classes in a system through the HITS webmining algorithm [17], which also explains why we focus on retrieving hubs for our technique.

2.4.2. HITS algorithm

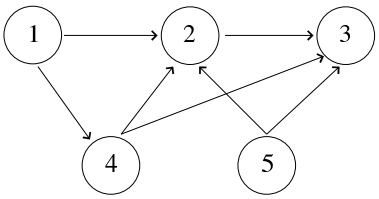
The HITS algorithm works as follows. Every node i gets assigned to it two numbers; a_i denotes the authority of the node, while h_i denotes the hubiness. Let $i \rightarrow j$ denote that there is a link from node i to node j. The recursive relation between authority and hubiness is captured by the following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

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

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

The HITS algorithm starts with initializing all h's and a's to 1. In a number of iterations, the values are updated for all nodes, using the previous iteration's values as input for the current iteration. Within each iteration, the h and a values for each node are updated according to the

Figure 3. Example graph and the accompanying first iterations of the HITS webmining algorithm.



		Nodes				
		1	2	3	4	5
Iterations	1	(1,1)	(1,1)	(1,1)	(1,1)	(1,1)
	2	(2,0)	(1,3)	(0,3)	(2,1)	(2,0)
	3	(4,0)	(3,6)	(0,5)	(6,2)	(6,0)
	4	(8,0)	(5,16)	(0,15)	(11,4)	(11,0)
	5	...	...	...	...	...

formulas (1) and (2). If after each update the values are normalized, this process converges to stable sets of authority and hub weights [22].

Adding weights to the edges of the graph is also possible and can capture the notion of relative importance of edges. This extension requires only a small modification to the update rules. Let $w[i, j]$ be the weight of the edge from node i to node j. The update rules become:

$$h_i = \sum_{i \to j} w[i, j] \cdot a_j \tag{3}$$

$$a_j = \sum_{i \to j} w[i, j] \cdot h_i \tag{4}$$

Example. Consider the example graph of Figure 3. The accompanying table, shows the first iteration steps of the hub and authority scores (represented by tuples *(H, A)*) for each of the five nodes in the example graph. Even after only 3 iterations steps, it becomes clear that 2 and 3 will be good authorities, as can be seen from their high A scores. Looking at the H values, 4 and 5 will be good hubs, while 1 will be a less good one. The algorithm generally stabilizes after around 11 iterations [22].

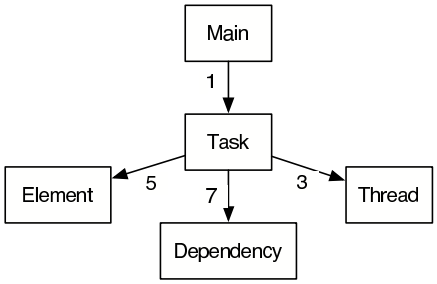
2.5. How it works in practice

We will now describe how each of the steps in our process are combined.

Step 1. Once the execution trace has been obtained, we start by calculating the metrics. For each type of metric, i.e., IC_CM, IC_CC or IC_CC′, we first calculate the individual coupling strengths that are present between individual class pairs. An example of this can be see in the listing below.

```
Main    Task        1
Task    Element     5
Task    Dependency  7
Task    Thread      3
```

Figure 4. Indirect coupling example.



An alternative representation is the compacted call graph (see Figure 4), which shows the exact same metric data, but in graphical form [23, 17]. This graph is constructed by creating a node for each class that is present in the execution scenario and by labeling the edges with the coupling strength (as determined by either the IC_CM, IC_CC or IC_CC′ metric).

Step 2. When we are interested in determining the direct coupling that exists within an application, we simply aggregate the coupling per class, which, e.g., in the case of the `Task` class would give an import coupling strength – only considering outgoing edges – of 15 in the above example.

Step 3. The compacted call graph is used as input for the HITS webmining algorithm so that the algorithm can reason over it and determine those classes that request a lot of services from other classes, i.e., import functionality. Because the HITS algorithm is iterative in nature, it not only takes into account direct links between classes, but also classes that are indirectly coupled to each other. When we reconsider the example in Figure 3 on page 10 we see that the hubiness score for node 1 benefits from the fact that node 4 is a good hub (because it is connected to good authorities). Because the relationship between hubs and authorities is mutually reinforcing (see formulas 1 and 2 on page 9), there is also no danger that the hubiness (or authority) scores keep reinforcing themselves, which would result in every node becoming strongly coupled to every other node [22].

Step 4. Rank the results from Step 2 and/or Step 3 according to respectively coupling strength and *hubiness* from high to low.

3. Case study setup

This section elaborates on the hypothesis, the research questions and the experimental setup that we have created for answering the research questions.

3.1. Hypothesis

Our hypothesis is that *dynamic coupling, which is very precise in its measurements with regard to polymorphism in object-oriented software, is a good indicator of whether a class in a system is actually a key class*. In order to validate this hypothesis, we refine it into a number of research questions:

1. Can dynamic coupling metrics provide an indication whether classes are "key classes"?
2. Which of the proposed coupling metrics — IC_CM, IC_CC and IC_CC′ — performs the best when retrieving key classes?
3. Can we improve our key class indicator by also taking into account indirect coupling?
4. As we know that dynamic analysis is typically an expensive operation due to the massive amount of data involved [24], we want to know whether static coupling is able to match the retrieval quality of dynamic coupling, while improving time-efficiency.

.

3.2. Case study setup

Our experimental setup is such that we use two open source software systems as case studies for answering the above research questions. Section 3.3 elaborates on the choice of case studies.

We perform these case studies in two rounds: a first round that deals with dynamic analysis and a second round that deals with static analysis. We now explain the rationale behind those two rounds.

Round 1. In the first round we solely work with dynamic coupling metrics. As dynamic coupling metrics have shown to be more precise in measuring coupling in object-oriented software due to the presence of polymorphism [16], it is our expectation that these metrics will perform best. An added benefit of using dynamic analysis in this context is that it becomes possible to employ a goal-driven strategy, wherein the program comprehension process can be steered by the definition of the execution scenario in such a way that only the features of interest of a software system are exercised [23].

Round 2. The second round deals with static analysis, as we want to make a trade-off analysis of computational cost versus recall of a dynamic analysis based solution versus a static analysis based solution. For this second round we take the best performing dynamic coupling metric and define a static counterpart for it.

3.3. Open source software systems

When selecting case studies, there are three requirements that we keep in mind due to our program comprehension context:

- The case studies should be public in nature in order to ensure repeatability of this (or similar) experiments within the research community.

Table III. Size-related information of the two case studies.

	Ant 1.6.1	JMeter 2.0.1
Classes (traced)	127	189
Classes (total)	1 216	245
Lines of code (LOC)	98 681	22 234

- The case studies should have extensive design documentation available that lets us verify whether we have actually detected all the classes that need to be understood early on (i.e., the so-called key classes).
- Ideally, the design documentation is also freely available, which is a further bonus with respect to the guarantee of repeatability of the experiment.

During our search, we found *Apache Ant 1.6.1* and *Jakarta JMeter 2.0.1* to adhere best to these criteria. An added benefit of these two software systems is that they are completely different kinds of applications: Ant is a command-line batch application, while JMeter features a highly interactive graphical user interface.

Some metric-related data of both projects can be found in Table III. Please note here that we mention both the total number of classes that are in the "source distribution" and the total number of classes within that source distribution that solely belong to the project itself, i.e., we removed classes that, e.g., belong to the Xerces XML parser, log4j, etc. The removal of library classes was done on the basis of the package structure, which, for both software projects, adhered to a clear naming convention making identification of library classes straightforward.

Apache Ant

Apache Ant 1.6.1[†] is a well-known build tool, mainly used in Java environments. It is a command-line tool, has no GUI and is single-threaded. It has a relatively small footprint, but it does however use a lot of external libraries (e.g., the Xerces XML library) and is user-extensible. Ant relies heavily on XML, as the build files that Ant processes are written entirely in XML. Ant is used in both open-source and industrial settings and it has been integrated in numerous (Java) Integrated Development Environments (IDE's) (e.g., Eclipse, IntelliJ IDEA, ...).

The source-file distribution of Apache Ant 1.6.1 contains 1216 Java classes. Only 403 of these classes (around 83 KLOC) are Ant-specific, as most of the classes in the distribution belong to general purpose libraries or frameworks, such as Apache ORO (for regular expressions) or Apache Xerces (XML parser). These libraries could easily be recognized through their package structure and package name and were omitted from the tracing operation.

[†]For more information, see: http://ant.apache.org. For the design documentation, see: http://codefeed.com/tutorial/ant_config.html

Jakarta JMeter

Jakarta JMeter 2.0.1[‡] is a Java application designed to test webapplications. It allows to verify the application (functionally), but it also allows to perform load-testing (e.g., to measure performance or stability of the software system). It is frequently used to test webapplications, but it can also handle SQL queries through JBDC. Furthermore, due to its architecture, plugins can be written for other (network) protocols. Results of performance measuring can be presented in a variety of graphs. JMeter is a tool which relies on a feature-rich GUI, uses threads abundantly and relies mostly on the functionality provided by the Java standard API (e.g., for network-related functionality).

The source-file distribution of Jakarta JMeter 2.0.1 consists of around 700 classes, while the core JMeter application is built up from 490 classes (23 KLOC).

3.4. Program comprehension baseline

When performing case studies with new reverse engineering techniques aimed at understanding a software system, there basically exist two paths to follow when trying to validate the results. A first path is the *intrinsic evaluation*, where the original developers and maintainers serve as an "oracle". Another possibility is to perform an extrinsic evaluation, where, e.g., a controlled experiment would serve as evaluator.

For this study we have chosen to follow the first route, namely to perform an intrinsic validation with the help of design documents of open source software systems that were left behind by the original developers and maintainers of the software projects.

Both software projects that we use in this paper have a particular type of documentation that is aimed at developers who want to start contributing to the project, but are unfamiliar with it. This documentation contains a high-level view of the control-flow of the system and for each class involved in this high-level view a short description is given. The program comprehension baseline is distilled from these design documents in such a way that each and every class mentioned in this high-level overview is contained in this baseline.

Understanding the classes involved in this program comprehension baseline would thus give the novice developer a general knowledge of the system. This "generality" should also be reflected in the choice of execution scenario when using dynamic analysis. Details of the specific execution scenarios that we use for both our case studies are explained in Sections 4.1 and 4.2.

3.5. Evaluation and validation

Typical in the field of information retrieval is the use of the concepts of precision and recall for determining the retrieval power of a technique. As we have taken great care during our case study selection process to have extensive design documentation available for our software systems, we are able to define a program comprehension baseline, which in turn allows us to

[‡]For more information, see: http://jakarta.apache.org/jmeter/. The design documentation can be found on the Wiki pages of the Jakarta JMeter project: http://wiki.apache.org/jakarta-jmeter

evaluate our approach in terms of *recall* and *precision*. A third evaluation criterion, namely the time it takes to run the analysis from start to finish, rounds out the evaluation criteria:

1. The *recall* of the result set, or in other words, the technique's retrieval power (the percentage of key classes retrieved by the technique versus the total number of key classes present in the baseline).
2. The *precision* of the result set, or in other words, the technique's retrieval quality (the percentage of key classes retrieved versus the total size of the result set).
3. The *time* it takes to perform the complete analysis, i.e., the time it takes to run the analysis from start to finish.

The first two criteria will serve as deciding factors for determining (1) which of the considered metrics performs best, (2) whether taking into account indirect coupling serves its purpose and (3) last but certainly not least whether the overall approach is indeed capable of detecting the key classes in a system. The third criterion, the time it takes to perform the analysis, will be used to perform a trade-off analysis and can also serve as a deciding factor when a number of variations perform equally well on the first two criteria.

3.6. Evaluation of the results

In Step 4 of our approach (see Section 2.5) we mentioned that we ranked the results of our approach (according to either their metric value or hubiness score, depending on the technique used). The resulting list gives an indication of classes that are important (top-ranked) to less important (low-ranked). However, for evaluation purposes we have to somehow draw the line, as to what the most important classes are that we want to compare with the baseline.

For this purpose we set the mark of classes to be compared with the baseline at the top 15% highest ranked classes in the result set. The rationale behind choosing this 15% marker is, firstly, the documentation from which we created the baseline mentioned around 10% of the total number of classes that we considered. Because we still wanted to maintain a small margin, we extended the set of classes to be evaluated to 15%. Secondly, because in practice we would ideally want to have a concise set of key classes for starting to understand the software, we did not want to extend the set of classes to be evaluated too much.

4. Case studies: 1st phase

In a first phase we will compare how the dynamic coupling metrics that were defined in Section 2.1.4 perform in retrieving the key classes of both of our case studies. We vary between using direct and indirect coupling. Section 4.1 discusses the results of Ant, while JMeter's results are discussed in Section 4.2.

4.1. Apache Ant

Execution scenario

We chose to let Ant build itself as the execution scenario of choice for our experiment. This scenario involved 127 classes. At first sight this may seem rather low, considering that Ant is built from 403 classes in total, however, this can be explained by the fact that the Ant architecture contains some very broad (and sometimes deep) inheritance hierarchies. For example the number of direct subclasses from the class *Task* is 104. Each of these 104 classes stands for a typical command line task, such as *mkdir*, *cvs*, ... As typical execution scenarios do not contain all of these commands (some are even conflicting, e.g., different versioning system or different platform-specific commands, e.g., ls versus dir), the execution scenario containing 127 classes covers all basic functionality of Ant.

The two main reasons why we chose this particular execution scenario are:

- From a post-mortem inspection of the trace, we know that this scenario offers a good balance of features that get exercised. As such, this scenario activates the most common features that are used to build a typical java project, including those for compiling, copying files into different directories, generating jar (archive) files, etc. Because this scenario activates the most common features, it serves our purpose of building up a general knowledge of the software system, even though the class coverage of our scenario is only 32%. We are aware that dynamic analysis techniques in general often use more than one execution scenario, but as we are looking for general knowledge, we preferred one general execution scenario, with the option of refining our results later on with more specialized execution scenarios.
- Every source file distribution of Ant contains this specific execution scenario, through the build.xml file that is included in the distribution, making replication of the experiment straightforward.

Results

Table IV presents the metric-results for Apache Ant. We present the results for each of the 3 basic metrics, i.e., IC_CM, IC_CC and IC_CC′, both with and without the webmining algorithm applied in columns 1 through 6. Column 7 contains the program comprehension baseline.

The IC_CM metric for a class c_1, which counts quadruples of the form (m_1, c_1, m_2, c_2) [§], exhibits the lowest recall of all dynamic analysis solutions: 40%. The IC_CM metric counts distinct method invocations originating from the same source (m_1, c_1) combination. As such, a class c_1 using low-level functionality from c_2 in each of its methods m_i, will get a high metric value. This causes noise in the result set, because we are actually looking for classes that use other (high-level) classes. This explains its relatively low recall when compared to the baseline.

[§] A tuple of the form (m_1, c_1, m_2, c_2) is the combination of a method m_1 from a class c_1 that calls a method m_2 from class m_2. The exact definition can be found in Table II on page 7.

Table IV. Ant dynamic metric data overview.

Class	1: IC_CM	2: IC_CC	3: IC_CC′	4: IC_CM + webmining	5: IC_CC + webmining	6: IC_CC′ + webmining	7: Ant docs
Project		√		√	√	√	√
UnknownElement	√	√	√	√	√	√	√
Task	√	√	√		√	√	√
Main						√	√
IntrospectionHelper		√	√	√	√	√	√
ProjectHelper		√	√	√	√	√	√
RuntimeConfigurable	√	√	√	√	√	√	√
Target	√	√	√	√	√	√	√
ElementHandler			√		√	√	√
TaskContainer	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	√
→ **recall** (%)	40	70	70	60	80	90	-
→ **precision** (%)	21	37	37	32	42	47	-
Trace collection	1h			1h			
Metric calculation	45 min			45 min			
HITS algorithm				30 sec			
→ **total time**	1h45			1h45:30			

The IC_CC and IC_CC′ metrics, which count (m_1, c_1, c_2) and (m_2, c_1, c_2) respectively, exhibit a similar recall of 70%. Although at this point, we would have expected IC_CC′ to perform considerably better, there is no noticeable difference with regard to the recall. Our expectation for a better performance from IC_CC′ stems from the fact that, just as is the case for IC_CM, IC_CC focusses on counting the originating class/method pair, while IC_CC′ shifts focus towards the target class/method pair.

When we apply the HITS webmining algorithm on the obtained results (columns 4 through 6), we see that the retrieval power of each of the metrics improves. IC_CM now retrieves 60% of the program comprehension baseline, IC_CC goes from 70% to 80%, while IC_CC′ improves to a recall of 90%.

The one class that none of the metrics detect is the `TaskContainer` class. Upon closer inspection, we noticed that this class is no longer part of the Ant distribution in version 1.6.1 and hence, we put N/A in Table IV. We decided to explicitly mention the `TaskContainer` class, because it is a good example that the documentation is often outdated. Table IV only shows the scores for the 10 classes that are mentioned in the baseline, while each of our metric-variations detect more than 10 classes, 19 to be exact for this experiment (we have taken 15% of the 107 traced classes). For completeness sake, we add that the IC_CC′ metric has also detected the following classes: `ComponentHelper`, `AbstractFileset`, `SelectSelector`, `DirectoryScanner`, `TaskAdapter`. Altough these classes are not mentioned in the baseline,

further inspection indicated that these classes also have a controlling function, meaning that they are also potentially useful to study early on.

Considering precision, applying the webmining algorithm allows to improve precision for all of the considered metrics. In the case of IC_CC′ it is able to bring precision to a level of 47%, which is a very satisfying result, given the fact that other than an execution scenario no domain knowledge is required for our key class detection technique. Nevertheless, we should keep in mind that around 50% of the program comprehension "pointers" returned are potentially of lesser value to the user.

On a final note, we also want to add that we have experimented with changing our retrieval rate to the top 20% ranked classes. By doing so, we have seen that recall did not significantly increase, to be more precise recall for IC_CM increased by 10% with all others remaining stable. Precision dropped for each of the metric variations. Lowering the retrieval rate to 10% of the highest ranked classes made recall drop significantly all over the line.

Time-effort analysis

When we run Ant according to the previously defined execution scenario, the execution takes 23 seconds without collecting trace-information. Table IV shows that when we enable trace collection, this scenario now takes slightly under 1 hour[¶], generating a trace of roughly 2 GB of data. Metric-calculation takes 45 minutes (the three metrics were calculated in parallel, only calculating one of these at a time lowers the time needed by only a fraction), while applying the HITS webmining algorithm on the metric data takes less than 30 seconds.

Discussion

For our first case study, we see that the IC_CC′ metric in combination with the HITS webmining algorithm outperforms the other metric-variations: it is able to retrieve 90% of the classes in the program comprehension baseline, with a precision of 47%. This kind of result makes the technique extremely useful for getting an initial, high-level view of the software component under study.

With regard to the time-effort, the complete analysis takes roughly 1 hour 45 minutes. This seems long, but we expect to be able to improve our tools, which are currently in a prototype state, and the algorithm can also be parallelized.

4.2. Jakarta JMeter

Execution scenario

The execution scenario for this experiment consists of testing a HTTP (HyperText Transfer Protocol) connection to a large online store. More precisely, we configured JMeter to test the

[¶]Experiment conducted on an AMD Athlon 800 with 512MB memory running Fedora Core 3 Linux.

Table V. JMeter dynamic metric data overview.

Class	1: IC_CM	2: IC_CC	3: IC_CC′	4: IC_CM + webmining	5: IC_CC + webmining	6: IC_CC′ + webmining	7: JMeter docs
AbstractAction	√		√	√	√	√	√
JMeterEngine			√	√	√	√	√
JMeterTreeModel						√	√
JMeterThread			√		√	√	√
JMeterGuiComponent	√	√	√				√
PreCompiler						√	√
Sampler		√	√	√	√	√	√
SampleResult		√		√	√	√	√
TestCompiler			√			√	√
TestElement			√		√	√	√
TestListener			√			√	√
TestPlan			√	√	√	√	√
TestPlanGui			√			√	√
ThreadGroup						√	√
→ **recall** (%)	14	21	71	36	50	93	-
→ **precision** (%)	7	11	36	18	25	46	-
Trace collection	45 min			45 min			
Metric calculation	30 min			30 min			
HITS algorithm				30 sec			
→ **total time**	1h15			1h15:30			

aforementioned connection 100 times and visualize the results in a simple graph. Running this scenario took 82 seconds. The scenario is representative for JMeter, because many of the possible variation points in the execution scenario lie in (1) the usage of a different protocol (e.g., FTP) or (2) in the output format of the data (e.g., different type of graph or plain-text). Also of importance to note here is that these 100 connections are initiated by a number of different threads, in order to simulate concurrent access to the web application. This entails that this particular experiment is an example of a multi-threaded application.

Results

Table V provides an overview of the results of the Jakarta JMeter case study, taking into account that the baseline contains 14 classes. For determining recall and precision, we again looked at the highest 15% ranked classes, i.e., 28 classes (15% of 189 classes).

The IC_CM metric clearly lags behind the other dynamic metrics proposed with a recall of 14% and a precision of 10%. The explanation for this relatively bad result is identical to the reasoning given for Ant.

In contrast with the previous experiment, there is a notable difference between the most tightly coupled classes as reported by IC_CC versus IC_CC′.
Although not immediately visible from Table V, this phenomenon is related to the feature-rich graphical user interface (GUI). Even though there is evidence of an attempt of a model-view-controller (MVC) pattern implementation [25] (both from source code and from design documents), there still is a high degree of coupling from the view to the model in the MVC scheme. Furthermore, a high degree of coupling exists within the GUI layer.
Because certain classes in the GUI layer of JMeter can be catalogued as god classes (many methods, large methods), the IC_CC metric falsely registers these classes as important, due to the high method count of these classes. IC_CC′ however does not suffer from this because its measure is not dependent on the number of methods defined within the class.

When we apply the HITS webmining algorithm to the previously discussed metrics, we see that taking into account indirect coupling does help to identify the key classes of a system. The IC_CC′, which already was the best performer without taking into account indirect coupling, comes out on top, attaining a level of recall of 93% with a level of precision of 46%.

Again, we have experimented with a different retrieval rate. When retrieving the highest 20% classes recall of IC_CM, IC_CC and IC_CC + webmining increased by respectively 14, 14 and 7%. Precision dropped for all metric variations. Lowering the retrieval rate to 10% leads to significant changes in both recall and precision, with no technique being able to recall more than 65% of the classes defined in the baseline.

Time-effort analysis

The original scenario that we studied during this experiment takes 82 seconds to run. With the added overhead of tracing JMeter, it now takes around 45 minutes; the final trace was roughly 600 MB in size. Notice the difference with the Ant experiment, where we collected 2 GB of trace data (for a time-wise shorter execution). This difference in size can mainly be attributed to the fact that JMeter heavily relies on library functions, which are excluded from the trace. This exclusion process however, also comes at an additional cost because for each call made, an exclusion-filter needs to be consulted before deciding whether to output a call to the tracefile or not.

Table V shows that calculating the metrics takes slightly under 30 minutes and applying the HITS webmining algorithm takes around 30 seconds.

Discussion

In terms of retrieval performance, we see a very similar situation to the one we encountered with Ant. IC_CC′ combined with the HITS webmining algorithm performs very strongly and recall and precision results are similar. Again, the time-effort proves worrisome.

4.3. Discussion

Table VI provides an overview of the results of the first phase of our case studies, the phase in which we compare the dynamic coupling metrics.

Table VI. Summary of the first phase of the case studies.

	Recall		Precision		Time-effort	
	Ant	JMeter	Ant	JMeter	Ant	JMeter
IC_CM	40%	14%	21%	7%	1h45	1h15
IC_CC	70%	21%	37%	11%	1h45	1h15
IC_CC′	70%	71%	37%	36%	1h45	1h15
IC_CM + webmining	60%	36%	32%	18%	1h45:30	1h15:30
IC_CC + webmining	80%	50%	42%	25%	1h45:30	1h15:30
IC_CC′ + webmining	90%	93%	47%	46%	1h45:30	1h15:30

For both our case studies, we see that applying the HITS algorithm to the dynamic coupling metrics improves their ability to retrieve the key classes of a system. In particular, the IC_CC′ metric in combination with the HITS webmining algorithm delivers convincing results for identifying the key classes: respectively 90% and 93% of the key classes as defined in the baseline are identified. Meanwhile, precision hovers slightly under 50%.

Considering the time-effort we see that for both our case studies the approach takes a long time. With this in mind, the applicability of the approach for large-scale software projects becomes questionable, even though the benefit of the approach is clear. Considering that a lot of this time-effort is spent in collecting the trace information, our subsequent question becomes: can we reach similar levels of retrieval performance, when considering only static information? This question is answered in Section 5.

5. Case studies: phase 2

During the first phase of both of our case studies we noticed that using dynamic analysis brought with it a number of constraints, namely:

- The need for a good execution scenario.
- The availability of a tracing mechanism.
- Scalability issues (the size of the trace file, run-time overhead introduced by the tracing mechanism, etc.).

Because of these constraints, we initiated a second phase of our case studies in which we validate that the good results that we have obtained through dynamic analysis, indeed warrant the time-effort [26]. In this second phase we apply the same webmining technique on a static topological structure of the application and investigate whether we can get a similar level of recall and precision as we found for the dynamic approach (see Section 4), with a significantly diminished time-effort.

In this new step we compare the best-performing dynamic coupling metric from the first phase, namely the combination of the IC_CC′ metric with the HITS webmining technique, and compare it with a static coupling metric that is modeled after the IC_CC′ metric, also combined with the HITS webmining technique.

Furthermore, because we want to make the comparison as objective as possible, the next section defines static coupling metrics that are as close as possible to the IC_CC′ metric used in the first phase of the case studies.

5.1. A static coupling metrics framework

The framework from Arisholm [16] does not have to make a distinction between static and polymorphic calls due to the dynamic nature of its measurements. We add notational constructs from the unified framework for (static) object-oriented metrics from Briand et al [19] to the definitions that we previously used from Arisholm. That way, we can still use the basic notation from Arisholm we have used in the previous chapters. For that purpose, some helpful definitions are:

Definition 1 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.
- $M_I(c) \subseteq M(c)$ 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 $SIM(m)$. The 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 $m' \in M(d)$.

5.2. Expressing IC_CC′ statically

With these added notational constructs, we are now able to write down four static coupling measures that closely resemble the measurements that were defined in Section 2.1.4.

Figure 5. Piece of Java code to help explain metrics.

```
1 public void foo() {
2   BaseClass base = new BaseClass();
3   base.doSomething();
4   // some other functionality
5   base.doSomething();
6 }
```

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 the Java code snippet in Figure 5:

- The choice between *static calls* and *polymorphic calls*. In other words when considering Figure 5, 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 5 do we count the `base.doSomething()` call once or twice (lines 3 and 5, Figure 5).

For the purpose of our research we have defined 4 metrics that vary over the characteristics described above.

Definition SM_SO Static Metric, Static calls, count every Occurrence of a call only once.
$SM\_SO(c_1, c_2) = |\{(m_2, c_2, c_1) | \exists\ (m_1, c_1), (m_2, c_2) \in R_{MC}$
$\wedge\ c1 \neq c2 \wedge\ (m_1, c_1, m_2, c_1) \in IV$
$\wedge\ m_2 \in SIM(m1)\}|$

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

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*.

To calculate these metrics, we used the JDT2MDR Eclipse plugin developed at the University of Antwerp [26]. JDT2MDR transforms a Java project to a graph representation closely resembling the metamodel employed by Briand et al. in their unified framework for coupling measurements in object-oriented software [19], thereby enabling the calculation of the coupling and cohesion measures formalized in their paper.

Table VII. Ant metric data overview.

Class	1: IC_CC′ + webmining	2: SM_PO + webmining	3: SM_PW + webmining	4: SM_SO + webmining	5: SM_SW + webmining	6: Ant docs
Project	√	√	√	√	√	√
UnknownElement	√	√	√	√	√	√
Task	√					√
Main	√	√	√	√	√	√
IntrospectionHelper	√	√	√			√
ProjectHelper	√					√
RuntimeConfigurable	√	√	√			√
Target	√					√
ElementHandler	√					√
TaskContainer	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	√
→ **recall** (%)	90	50	50	30	30	-
→ **precision** (%)	47	8	8	5	5	-
Trace collection	1h					
Metric calculation	45 min	1h				
HITS algorithm	30 sec	1 min				
→ **total time**	1h45:30	1h01				

5.3. The continuation of the case studies

This section compares and discusses the statically obtained results with (1) the best-performing dynamic analysis approach and (2) the program comprehension baseline that we have defined. Besides comparing recall and precision, we also keep a close eye on time-effort, as this is a factor where we expect the static approach to be able to significantly outperform the dynamic approach.

5.3.1. Ant

Based on the results shown in Table VII, two categories are formed, namely the category of metrics that takes polymorphism into account (SM_P*) and the category that does not take polymorphism into account (SM_S*). The former category exhibits a recall level of 50%, while the latter recalls 30%. Although interesting from the point of view that polymorphism does indeed play an important role when considering program comprehension, from a practical perspective, these results are disappointing when compared to the results obtained with the dynamic approach. The observation regarding polymorphism 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 inheritance hierarchy. The second variation point for the static metrics, namely whether to only count an occurrence of a particular call once or to 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 the result set).

The fact that precision for the 4 static metrics in columns 2 through 5 is much lower (8% or less) than what we experienced with the dynamic approach, 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. When using our rule-of-thumb of presenting the 15% highest ranked classes in the final result set, we end up with 60 and 19 classes respectively.

A further point to be made regarding this rule-of-thumb 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. We can also add, that by raising the bar to 10%, recall would fall significantly.

Considering the round-trip-time, we measured that the prototype (static) metrics engine took one hour to calculate the metrics for Ant. Applying the HITS algorithm takes less than one minute.

5.3.2. JMeter

Similar to what we saw with Ant, two groups can be identified within the JMeter result set presented in Table VIII, namely one group consisting out of SM_PO and SM_PW, and one group formed by SM_SO and SM_SW. Within these two groups, recall and precision are identical, although minimal differences exist when looking at the ranking of some classes. In contrast with the results for Ant, these differences are much more pronounced. It is our opinion that this is probably due to the fact that most method calls happen only once in each unique method, as opposed to multiple occurrences of a method call in a unique method, where the *weighted* approach (of SM_PW and SM_SW) would make the difference more pronounced.

Also to be noted is the sizeable dissimilarity between the results obtained while only taking into account static calls versus also taking polymorphic calls into account. As Table VIII shows, the SM_P* metrics have a recall of 43%, while the SM_S metrics only recall 7%.

For what the round-trip-time is concerned, the metrics engine took almost 1 $\frac{1}{2}$ hours to calculate the metrics for JMeter. This is a considerable increase from what we saw with Ant. This increase can be attributed to the fact that JMeter has (1) a larger codebase and (2) uses more libraries, which also need to be parsed. Applying the HITS algorithm takes slightly over one minute.

5.4. Discussion

We began this section by stating that there are three major drawbacks to the dynamic approach that we presented. Now that we have performed the second phase of our case study in which we tried out a static variant of our approach, we come back to each of these drawbacks in order to verify whether the static variant of our approach was able to solve them:

1. *The necessity of a good execution scenario.*
   When performing static analysis, having an execution scenario is not an issue. However,

Table VIII. JMeter metric data overview.

Class	1: IC_CC′ + webmining	2: SM_PO + webmining	3: SM_PW + webmining	4: SM_SO + webmining	5: SM_SW + webmining	6: JMeter docs
AbstractAction	√					√
JMeterEngine	√	√	√			√
JMeterTreeModel	√	√	√			√
JMeterThread	√	√	√			√
JMeterGuiComponent		√	√			√
PreCompiler	√					√
Sampler	√					√
SampleResult	√					√
TestCompiler	√	√	√			√
TestElement	√	√	√			√
TestListener	√					√
TestPlan	√					√
TestPlanGui	√			√	√	√
ThreadGroup	√					√
→ **recall** (%)	93	43	43	7	7	-
→ **precision** (%)	46	8	8	1.4	1.4	-
Trace collection	45 min					
Metric calculation	30 min		1h30			
HITS algorithm	30 sec		1 min			
→ **total time**	1h15:30		1h31			

having access to the source code is an important prerequisite for any static analysis based approach. On the other hand, having access to the source is generally much easier than having access to a good execution scenario. As such, for this criterion, static analysis is to be favored.

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 a clear advantage here.
3. *Scalability issues.*
   In terms of scalability the dynamic process is plagued by the possibly huge size of the trace file, which result in long analysis times. However, when comparing these times with the static approach, we observe that our prototype metrics engine also takes a long time to compute the metrics. While the analysis times do not differ that much from

Table IX. Summary of the case studies.

		Recall		Precision		Time-effort	
		Ant	JMeter	Ant	JMeter	Ant	JMeter
Dynamic	IC_CM	40%	14%	21%	7%	1h45	1h15
	IC_CC	70%	21%	37%	11%	1h45	1h15
	IC_CC′	70%	71%	37%	36%	1h45	1h15
	IC_CM + webmining	60%	36%	32%	18%	1h45:30	1h15:30
	IC_CC + webmining	80%	50%	42%	25%	1h45:30	1h15:30
	IC_CC′ + webmining	90%	93%	47%	46%	1h45:30	1h15:30
Static	SM_PO + webmining	50%	43%	8%	8%	1h01	1h31
	SM_PW + webmining	50%	43%	8%	8%	1h01	1h31
	SM_SO + webmining	30%	7%	5%	1.4%	1h01	1h31
	SM_SW + webmining	30%	7%	5%	1.4%	1h01	1h31

the dynamic process, the dynamic process is still burdened by the time-intensive tracing step, which makes that the total time for the dynamic process is significantly larger.

6. Discussion

Table IX extends Table VI by also taking into account the results of the static variant of our approach. Table IX shows that the best-performing dynamic analysis based variant of our approach, namely the IC_CC′ metric combined with the webmining solution provides a level of recall of at least 90%, while safeguarding a level of precision of slightly under 50%. When we look at the results of the static coupling metrics that we introduced in this chapter, we see that we are able to reach a maximum level of recall of 50%, while the level of precision drops to 8% or less. This observation makes it quite obvious that the dynamic approach is the solution of choice when only considering the recall and precision results.

When considering the time-effort for these analyses, we see that the static approach (the SM_* metrics) performs better than the dynamic variants, with the important remark that recall and precision clearly fall behind the best-performing dynamic variant of our approach.

As such, we conclude that for the purpose of detecting the key classes that can be helpful for early program comprehension, the dynamic variant of our approach is the best choice, even though the time effort needed for the detection process should be considered as a serious drawback.

6.1. Threats to validity

Over the course of our case studies we noted a number of factors that could influence the validity of our conclusions. We will now discuss these threats to validity.

1. The design documents that we use as the basis for the program comprehension baseline are likely to be subjective, as each developer probably has a preference for the parts of the application that he has written himself. This problem is inherent to the intrinsic

evaluation that we perform and would likely also occur when consulting the developers or maintainers of a project directly (instead of working with documentation).
2. Although the results of our approach are very positive, we must also not forget that the intrinsic evaluation as we have performed it in this paper might not be representative for how developers get acquainted with a software system. Therefore, we foresee a controlled experiment in the future which will probably get a more realistic picture of the actual usefulness of our technique in practice.
3. For the evaluation of our detection technique, we relied on the concepts of precision and recall. However, when using a fixed retrieval rate (e.g., 15%) precision is directly tied to recall and would thus appear to be redundant. Nevertheless, when using the technique for understanding a software system, it is still beneficial to know how many possible *false positives* are returned in the result set. As such, we continue to work with both recall and precision.

7. Related work

Within the research community, three distinct approaches exist with regard to reverse engineering software systems, namely (1) static analysis, (2) dynamic analysis and (3) a hybrid approach combining the previous two. The second category, dynamic analysis, is characterized by the need to process huge amounts of data, and thus, dynamic analysis solutions are often tailored around the problem of scalability. Nevertheless, many researchers emphasize the importance of dynamic analysis in the reverse engineering process; this is especially true in the context of object-oriented systems [27]. The need for a hybrid approach where static analysis is reinforced by dynamic information, or vice versa, has also been advocated in the research community (e.g., [28]).

To overcome scalability issues when using dynamic analysis, two distinct approaches are currently used, namely (1) the compression and abstraction of dynamic information and (2) the visualization of dynamic information through condensed views. We provide a brief overview of both categories in Sections 7.2 and 7.3 respectively. We start with Section 7.1 where we discuss a static analysis based technique that closely matches our own technique.

7.1. Static analysis based

Robillard presents a technique whereby given a set of classes under investigation, a number of (related) classes that should be investigated next are provided [29]. The technique described is based on the (static) topological structure of the dependencies in a software system. Given an input set, the technique produces a fuzzy set describing other elements of potential interest. As such, the main difference between our solution and Robillard's solution is the fact that his solution needs a pre-established set of points of interest, whereas our solution provides these automatically based on the execution scenario. Further study of possible interactions between both solutions seems warranted.

7.2. Abstraction and compression

Hamou-Lhadj et al. have been working on a number of trace abstraction techniques [30]. The one that is most relevant in the context of our own technique is the technique that removes classes from a trace that are solely responsible for low-level functionality [31]. Just as in our technique, determining coupling lies at the basis of this removal technique. The major difference then being that Hamou-Lhadj et al.'s technique works from the bottom up, while our technique is more top-down oriented. Another technique of interest that was developed by Hamou-Lhadj and Lethbridge is *trace summarization*, where interesting sections of a trace are identified, which are then presented to the user. This technique can also be cataloged as being top-down [32].

Another common approach for abstracting traces is *feature analysis*, where – parts of – traces are correlated to the exact feature that the trace is performing. Greevy et al. [33] and Eisenbarth et al. [34] have been actively working in this field. Other interesting work is performed by Reis and Renieris [35], who encode program executions, and Richner and Ducasse [36], who reason over execution traces with logic queries. None of these techniques however try to provide key classes to the software engineer for further investigation.

7.3. Visualization

Over the years many visualization techniques have been used to visualize the interaction of runtime objects [37, 38], with the main aim of abstracting the large amounts of dynamic information. We list a number of recently developed techniques in this context.

Some of the proposed visualizations build upon the idea of UML sequence diagrams, such as the work of De Pauw et al. in Jinsight [39, 40]. Jerding et al. also use sequence diagrams in ISVis, but in order to overcome scalability, they also offer a mural view of that sequence diagram to allow for easy navigation [38, 41].

Other work introduces more novel visualization ideas. These visualizations are often built around a metaphor, such as the work of Kuhn and Greevy, who interpret traces as signals in time [42], or Cornelissen et al., who project a software system's execution on the edge of a circle [43]. Greevy et al. use *growing towers* to visualize the number of instances of a certain class that are active [44].

Finally, some techniques have previously been used for displaying static information, but have been adapted to work with dynamic information. In this context Ducasse et al. use polymetric views [45]. These polymetric views were borrowed from Lanza's CodeCrawler tool [46]. The AVID visualization tool from Walker et al. [47, 48], has close ties with Murphy et al.'s Reflexion [49].

8. Conclusion

New programmers are often stuck sorting through hundreds of classes in order to find the few that offer significant insight into the interworkings of the project. Our *Key Class Identification* technique approach can reduce their difficulties, because it allows for the

automatic identification of those classes in a software system that are prime candidates for early program comprehension. We demonstrated our approach using two open source software systems, namely Apache Ant and Jakarta JMeter, which were specifically chosen because of their rich documentation set. With this documentation we were able to establish a 'program comprehension baseline'.

Fundamental to the approach we propose is the concept of coupling and the HITS webmining technique. Using these two basic principles, we applied our approach both on dynamically and statically obtained information from the target systems. Our case studies have shown that when using dynamically gathered information and taking into account indirect coupling, we are able to recall 90% of the key classes present in a system according to the documentation. Furthermore, we are able to provide this level of recall while remaining at a level of precision of slightly under 50%.

As such, we feel that we have presented a valuable technique for a software developer or maintainer who is aiming to familiarize himself with a previously unknown software system. By starting from the key classes, the user has a limited number of starting points to further his quest for gaining a thorough level of knowledge of the system.

8.1. Future work

For future work, we have identified a number of paths that allow to refine the validation of the key class identification technique. Firstly, we aim to perform a controlled experiment that lets us assess the usefulness of having key classes to start understanding a complex software system when the user has no knowledge of the system. Secondly, we want to apply the technique on a wide variety of applications from diverse problem domains. We have started working in this direction by applying the technique on industrial software and performing a validation with the actual developers. An initial report on this experiment can be found in [50].

ACKNOWLEDGEMENTS

This research has been carried out within the *ARRIBA* research project, sponsored by the *Institute for the Promotion of Innovation by Science and Technology* in Flanders, Belgium. Other sponsoring came from the *NWO Jacquard RECONSTRUCTOR* project and the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy (project *MoVES*).

REFERENCES

1. Lehman M, Belady L. *Program evolution: processes of software change*. Academic Press Professional, Inc.: San Diego, CA, USA, 1985.
2. Demeyer S, Ducasse S, Nierstrasz O. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.
3. Ko AJ, Myers BA, Coblenz MJ, Aung HH. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering* 2006; **32**(12):971–987.
4. Corbi TA. Program understanding: Challenge for the 90s. *IBM Systems Journal* 1990; **28**(2):294–306.
5. Spinellis D. *Code Reading: The Open Source Perspective*. Addison-Wesley: Boston, MA, USA, 2003.

6. Wilde N. Faster reuse and maintenance using software reconnaissance. *Technical Report*, Technical Report SERC-TR-75F, Software Engineering Research Center, CSE-301, University of Florida, CIS Department, Gainesville, FL 1994. URL `citeseer.nj.nec.com/wilde94faster.html`.
7. Biggerstaff TJ, Mitbander BG, Webster D. The concept assignment problem in program understanding. *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society: Los Alamitos, CA, USA, 1993; 482–498.
8. Lakhotia A. Understanding someone else's code: Analysis of experiences. *Journal of Systems and Software* Dec 1993; **23**(3):269–275.
9. von Mayrhauser A, Vans AM. Program comprehension during software maintenance and evolution. *IEEE Computer* Aug 1995; **28**(8):44–55.
10. Robillard MP, Coelho W, Murphy GC. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering* 2004; **30**(12):889–903.
11. Tahvildari L, Kontogiannis K. Improving design quality using meta-pattern transformations: A metric-based approach. *Journal of Software Maintenance and Evolution: Research and Practice* 2004; **16**(4–5):331–361.
12. Wand Y, Weber R. An ontological model of an information system. *IEEE Transactions on Software Engineering* November 1990; **16**(11):1282–1292.
13. Selby RW, Basili VR. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering* 2 1991; **17**(2):141–152.
14. Lethbridge TC, Anquetil N. Experiments with coupling and cohesion metrics in a large system 1998. Working paper, School of Information Technology and Engineering, also see http://www.site.uottawa.ca/ tcl/papers/ metrics/ExpWithCouplingCohesion.html.
15. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 1994; **20**(6):476–493.
16. Arisholm E, Briand L, Foyen A. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering* 8 2004; **30**(8):491–506.
17. Zaidman A, Calders T, Demeyer S, Paredaens J. Applying webmining techniques to execution traces to support the program comprehension process. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Los Alamitos, CA, USA, 2005; 134–142.
18. Yang HY, Tempero E, Berrigan R. Detecting indirect coupling. *Proceedings of the Australian Software Engineering Conference (ASWEC)*, IEEE Computer Society: Los Alamitos, CA, USA, 2005; 212–221.
19. Briand LC, Daly JW, Wüst JK. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering* 1999; **25**(1):91–121.
20. Brin S, Page L. The anatomy of a large-scale hypertextual web search engine. *Computer Networks* 1998; **30**(1-7):107–117.
21. Gibson D, Kleinberg JM, Raghavan P. Inferring web communities from link topology. ACM: New York, NY, USA, 1998; 225–234.
22. Kleinberg JM. Authoritative sources in a hyperlinked environment. *Journal of the ACM* 1999; **46**(5):604–632. URL `citeseer.ist.psu.edu/article/kleinberg98authoritative.html`.
23. Zaidman A. Scalability solutions for program comprehension through dynamic analysis. PhD Thesis, University of Antwerp 2006.
24. Zaidman A, Demeyer S. Managing trace data volume through a heuristical clustering process based on event execution frequency. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Los Alamitos, CA, USA, 2004; 329–338.
25. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.
26. Zaidman A, Du Bois B, Demeyer S. How webmining and coupling metrics can improve early program compehension. *Proceedings of the International Conference on Program Comprehension (ICPC)*, IEEE Computer Society: Los Alamitos, CA, USA, 2006; 74–78.
27. Stroulia E, Systä T. Dynamic analysis for reverse engineering and program understanding. *ACM SIGAPP Applied Computing Review* 2002; **10**(1):8–17.
28. Systä T. On the relationships between static and dynamic models in reverse engineering java software. *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, IEEE Computer Society: Los Alamitos, CA, USA, 1999; 304–313.
29. Robillard MP. Automatic generation of suggestions for program investigation. *SIGSOFT Software Engineering Notes* 2005; **30**(5):11–20.

30. Hamou-Lhadj A. Techniques to simplify the analysis of execution traces for program comprehension. PhD Thesis, University of Ottawa, Canada 2005.
31. Hamou-Lhadj A, Braun E, Amyot D, Lethbridge T. Recovering behavioral design models from execution traces. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Los Alamitos, CA, USA, 2005; 112–121.
32. Hamou-Lhadj A, Lethbridge T. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. *Proceedings of the International Conference on Program Comprehension (ICPC)*, IEEE Computer Society: Los Alamitos, CA, USA, 2006; 181–190.
33. Greevy O, Ducasse S. Correlating features and code using a compact two-sided trace analysis approach. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Los Alamitos, CA, USA, 2005; 314–323.
34. Eisenbarth T, Koschke R, Simon D. Locating features in source code. *IEEE Transactions on Software Engineering* 2003; **29**(3):210–224.
35. Reiss SP, Renieris M. Encoding program executions. *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM: New York, NY, USA, 2001; 221–230.
36. Richner T, Ducasse S. Using dynamic information for the iterative recovery of collaborations and roles. *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society: Los Alamitos, CA, USA, 2002; 34–43.
37. Pauw WD, Jensen E, Mitchell N, Sevitsky G, Vlissides JM, Yang J. Visualizing the execution of java programs. *Software Visualization, International Seminar Dagstuhl Castle (2001)*, *Lecture Notes in Computer Science*, vol. 2269, Diehl S (ed.), Springer: Berlin/Heidelberg, Germany, 2002; 151–162.
38. Jerding DF, Stasko JT, Ball T. Visualizing interactions in program executions. *Proceedings of the International conference on Software Engineering (ICSE)*, ACM: New York, NY, USA, 1997; 360–370.
39. De Pauw W, Helm R, Kimelman D, Vlissides JM. Visualizing the behavior of object-oriented systems. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM: New York, NY, USA, 1993; 326–337.
40. De Pauw W, Lorenz D, Vlissides J, Wegman M. Execution patterns in object-oriented visualization. *Proceedings of the Conference on Object-Oriented Technologies and Systems (COOTS)*, USENIX: Berkeley, CA, USA, 1998; 219–234.
41. Jerding DF, Stasko JT. The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics* 1998; **4**(3):257–271.
42. Kuhn A, Greevy O. Exploiting the analogy between traces and signal processing. *Proceedings of the International Conference on Software Maintenenace (ICSM)*, IEEE Computer Society: Los Alamitos, CA, USA, 2006; 320–329.
43. Cornelissen B, Holten D, Zaidman A, Moonen L, van Wijk JJ, van Deursen A. Understanding execution traces using massive sequence and circular bundle views. *Proceedings of the International Conference on Program Comprehension (ICPC)*, IEEE Computer Society: Los Alamitos, CA, USA, 2007; 49–58.
44. Greevy O, Lanza M, Wysseier C. Visualizing live software systems in 3D. *Proceedings of the Symposium on Software visualization (SoftVis)*, ACM: New York, NY, USA, 2006; 47–56.
45. Ducasse S, Lanza M, Bertuli R. High-level polymetric views of condensed run-time information. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Los Alamitos, CA, USA, 2004; 309–318.
46. Lanza M. Object-oriented reverse engineering — coarse-grained, fine-grained, and evolutionary software visualization. PhD Thesis, University of Berne 2003.
47. Walker RJ, Murphy GC, Freeman-Benson B, Wright D, Swanson D, Isaak J. Visualizing dynamic software system information through high-level models. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, *ACM SIGPLAN Notices*, vol. 33, ACM: New York, NY, USA, 1998; 271–238.
48. Chan A, Holmes R, Murphy GC, Ying AT. Scaling an object-oriented system execution visualizer through sampling. *Proceedings of the International Workshop on Program Comprehension (IWPC)*, IEEE Computer Society: Los Alamitos, CA, USA, 2003; 237–244.
49. Murphy GC, Notkin D, Sullivan K. Software reflexion models: bridging the gap between source and high-level models. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, ACM: New York, NY, USA, 1995; 18–28.
50. Zaidman A, Adams B, De Schutter K, Demeyer S, Hoffman G, De Ruyck B. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, IEEE Computer Society: Los Alamitos, CA, USA, 2006; 89–98.