Mining ArgoUML with Dynamic Analysis to Establish a Set of Key Classes for Program Comprehension

position paper

Andy Zaidman and Serge Demeyer

University of Antwerp
Department of Mathematics and Computer Science
Lab On Re-Engineering
Middelheimlaan 1, 2020 Antwerp, Belgium
{Andy.Zaidman, Serge.Demeyer}@ua.ac.be

Abstract

Initial program comprehension could benefit from knowing the most important classes in the system under study. Typically, in well-designed object-oriented programs, a few key classes work tightly together to provide the bulk of the functionality. Therefore, a viable strategy is to investigate highly coupled classes, as these are likely to be elements of the core design of a software system. Previous research has shown that using webmining principles in combination with dynamic coupling metrics can help in establishing a set of first-to-be-looked-at classes when familiarizing oneself with a software system. This paper describes the results of applying this technique on ArgoUML.

Keywords: *Dynamic analysis, program comprehension, coupling metrics, webmining, ArgoUML.*

1 Introduction

Reverse engineering is defined as the analysis of a system in order to identify its current components and their dependencies and to create abstractions of the system's design [2]. In practice, any reverse engineering operation almost always takes place in service of a specific purpose, such as re-engineering to add a specific feature, maintenance to improve the efficiency of a process, reuse of some of its modules in a new system, etc. [18, 20]. In order to perform any of these operations the software engineer must comprehend a given program sufficiently well to plan, design and implement modifications and/or additions.

As such, program comprehension can be defined as the process the software engineer goes through when studying the software artifacts, with as ultimate goal the sufficient understanding of the system to perform the required operations [16, 15]. Such software artifacts could include the source code, documentation and/or abstractions from the reverse engineering process.

Estimates go as far as stating that a programmer spends 40% of the time-budget of a maintenance operation on program comprehension [21, 17, 3]. Although gaining understanding is a daunting task and is often sped-up to deliver the project in the shortest possible timeframe [5] it is absolutely necessary to get an adequate understanding of a software system before making changes.

Dynamic analysis can be very helpful for understanding the behavior of a large system. Understanding an object-oriented system, for example, can be very hard if one relies solely on the source code and static analysis. Due to the abundant use of polymorphism in object oriented software, the relationships between the system artifacts tend to be obscure [11].

Dynamic data gathered from the execution of a scenario of the program is typically stored in so-called execution traces. These execution traces tend to explode in size, as the amount of dynamic data collected during the execution of a simple scenario is huge [11, 9].
Presenting this data to the user, without any form of processing is – due to its size – quite useless. What is needed is a technique that improves the intelligibility of the trace. In other words: the huge trace has to be chunked in readable and understandable parts. Preferably, these chunks presented to the user are essential in building up the knowledge of the software project under study.
Existing solutions rely on visualization schemes for presenting the trace to the user in a readable and under-

standable way [4, 12, 13, 19]. The amount of data however, remains the same. As such, it is up to the user to decide which information is appropriate and which information he/she actually needs. Another approach in this context is applying heuristics to the event trace in order to find either (1) what parts of the trace can be removed without losing information (e.g. duplicate parts) or (2) what parts of the trace are absolutely necessary in order to extract moderately sized high-level representations from it [11, 7, 6].
The research presented in this paper should be situated in the latter category.

Recently there have been a number of interesting advances within this branch of research [9, 11, 10, 22]. All techniques presented are targeted towards enabling program comprehension in a dynamic analysis context. However, in order to make an adequate comparison of these techniques, these techniques should all be applied on a common case study. The research presented in this paper is a first step in that direction.

This paper presents the results of applying the HITS webmining algorithm [14] on dynamic coupling measures [22]. As a common case for comparing different program comprehension tools, ArgoUML was chosen. ArgoUML is a CASE (Computer Aided Software Engineering) tool that allows for graphical software design. Its concept is centered around open source and open standards such as XMI, SVG and PGML.

The structure of this paper can be summarized as follows: in Section 2 we very briefly explain our technique. Section 3 shows the results of applying our technique on ArgoUML, while Section 4 concludes with future work and the conclusion.

2 Webmining

Using coupling measures to determine which classes ”distribute” functionality in an application seems to be a good starting point for determining which classes to examine first during early program comprehension. By using dynamic coupling measures, one eliminates the drawback that polymorphism isn’t taken into consideration. However, one of the drawbacks of using (dynamic coupling) measures is that each coupling measure is calculated between two classes. As such, a class *A*, which is weakly coupled to a class *B*, which itself is tightly coupled to several other classes, will remain a weakly coupled class. Nevertheless, *A* can have a great impact on *B* and all classes that *B* is coupled to. We want to address this situation, by looking at a measure of coupling that can best be described as being *transitive*.

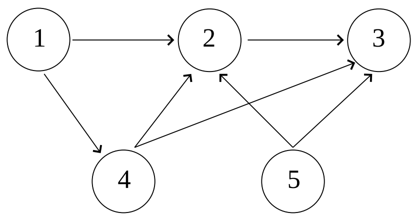

Figure 1. Example web-graph

This section will explain how webmining algorithms, that are iterative-recursive in nature, can be used for expressing this *transitiveness*.

2.1 Introduction to webmining

In datamining, many successful techniques have been developed to analyze the structure of the web [1, 8, 14]. Typically, these methods consider the Internet as a large graph, in which, based solely on the hyperlink structure, important web pages can be identified. In this section we show how to apply these successful web mining techniques to a compacted call graph of a program trace, in order to uncover important classes.

First we introduce the HITS webmining-algorithm [14] to identify so-called hubs and authorities on the web. Then, the HITS algorithm is combined with the compacted call graph. Through our case studies we will show that the classes that are associated with good “hubs” in the compacted call graph are good candidates for early program understanding.

2.2 Identifying hubs in large webgraphs

In [14], the notions of *hub* and *authority* were introduced. Intuitively, on the one hand, hubs are pages that rather refer to pages containing information then being informative themselves. Standard examples include web directories, lists of personal pages, ... On the other hand, a page is called an authority if it contains useful information. The HITS algorithm is based on this relation between hubs and authorities.

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

The HITS algorithm works as follows. Every page i gets assigned to it two numbers; a_i denotes the authority of the page, while h_i denotes the hubiness. An edge like $i \to j$ denotes that there is a hyperlink from page i to page j.
It is also possible to add weights to the edges in the graph. Adding weights to the graph can be interesting to capture the fact that some edges are more important than others. Let $w[i,j]$ be the weight of the edge from page i to page j. The recursive relation between authority and hubiness is captured by the following formulas.

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

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

The algorithm starts with initializing all h's and a's to 1, and repeatedly updates the values for all pages, using the formulas (1) and (2). If after each update the values are normalized, this algorithm is known to converge to stable sets of authority and hub weights. The convergence criterion, i.e. until the sum of absolute values of weight changes fall below a constant threshold, is shown to be reached around eleven iterations [14].

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

2.3 Applying webmining to execution traces

Within our problem domain, hubs can be considered *coordinating classes*, while authorities correspond to classes providing small functionalities that are used by many other classes. As such, hubs and authorities are conceptually similar to respectively import and export coupling. Again we expect hub classes to play a pivotal role in a system's architecture. Therefore, hubs are excellent candidates for beginning the program comprehension process or for gaining quick and initial program understanding [22].

In order to apply the HITS webmining algorithm to execution traces, an abstraction of the trace in the form of a graph has to be made. Because a trace can be seen as a large tree that contains the complete calling sequence of a program run, it is not so difficult to transform this large tree into a more compact call graph. Figure 2 shows an example of a so-called *compacted call graph*. The compacted call graph is derived from the dynamic call graph; it shows an edge between two classes A $\to$ B if an instance of class A sends a message to an instance of class B. As such, the compacted call graph doesn't has the complete calling-structure information of the program run, but it does have the data for each class-interaction in it.
The weights on the edges give an indication of the tightness of the collaboration as it is the number of distinct messages that are sent between instances of both classes. More formally:

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

where a_i and b_j are instances of respectively class A and class B and $M(a,b)$ is the set of messages sent from a to b. Also, to exclude cohesion: $A \neq B$. In this context a message is defined by its signature and thus by the type(s) of its formal parameter(s).

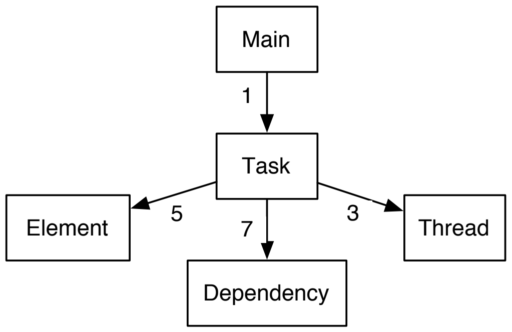

Figure 2. A compacted call graph

3 Applying the technique on ArgoUML

3.1 About ArgoUML

ArgoUML is an open source, Java-written CASE tool that embraces open standards such as XMI, SVG and PGML. It allows to design an application through a UML class diagram and has code-generation capabilities.

ArgoUML is centered around the concept of a *project*. A single project can be open at any time. One project corresponds to a *model* plus diagram information, i.e. everything you can edit within the ArgoUML window. This model can contain many *modelelements*, which form the complete UML description of the system you are describing.

ArgoUML requires a number of libraries in order to function, namely an XML parser (Xerces), a parser generator (Antlr), a logging framwork (log4j) and an internationalization library (i18n). For the purposes of our

program comprehension task, we excluded these libraries and/or frameworks from our analysis.
This leaves us with:

- 523 classes that form the core of ArgoUML
- 321 classes from external libraries (e.g. Sourceforge's Novosoft UML library and the OCL library from TU Dresden). We decided to include these external libraries because they form an integral part of the solution in the problem domain.

3.2 Execution scenario

Because our technique is based on dynamic analysis, a carefully chosen execution scenario is an necessity. The execution scenario we used for this small experiment was:

- Starting up ArgoUML
- Drawing a small class diagram which contains 6 classes
- Saving the project
- Quitting ArgoUML

3.3 Results

In our previous experiments with this technique, we have noticed that around 10% of the classes of a system should be considered as key classes for initial program understanding. For evaluation purposes, we considered the 15% highest ranked classes of our technique in those experiments, to give our heuristic a certain margin of error.
However, those software systems under study, where limited to between 100 and 150 classes, while in the case of ArgoUML, we are considering more than 500 classes. Because of the size of the project, we have decided to limit the results of our technique, to the 50 highest-ranked classes (or just below 10%). These results are shown in Table 1.

4 Conclusion and future work

The documentation of ArgoUML doesn't allow for an immediate benchmark as to how effective our technique is for detecting the key classes in ArgoUML for initial program comprehension. As such, no real conclusions can be drawn.
This paper must be seen as a first step in a larger study that compares recent research that aims at developing program comprehension support-tools based on dynamic analysis.

application.Main
ui.DetailsPane
ui.TabProps
diagram.ui.UMLDiagram
ui.TabStyle
ui.PropPanel
ui.ProjectBrowser
diagram.ui.FigNodeModelElement
diagram.static_structure.ui.UMLClassDiagram
kernel.Project
ui.ActionRemoveFromModel
diagram.use_case.ui.UMLUseCaseDiagram
diagram.static_structure.ui.FigClass
ui.targetmanager.TargetManager
diagram.ui.FigEdgeModelElement
ui.explorer.ExplorerTree
ui.TabTaggedValues
foundation.core.UMLModelElementNamespaceComboBoxModel
foundation.core.UMLModelElementStereotypeComboBoxModel
ui.StylePanel
ui.UMLComboBoxModel2
ui.UMLTreeCellRenderer
ui.UMLPlainTextDocument
diagram.ui.FigGeneralization
diagram.UMLMutableGraphSupport
diagram.static_structure.ui.ClassDiagramRenderer
persistence.AbstractFilePersister
ui.UMLCheckBox2
ui.explorer.rules.GoClassifierToSequenceDiagram
ui.explorer.rules.GoNamespaceToDiagram
ui.foundation.core.UMLGeneralizationPowertypeComboBoxModel
ui.foundation.core.ActionSetModelElementNamespace
ui.UMLModelElementListModel2
ui.TableModelTaggedValues
ui.foundation.core.UMLClassActiveCheckBox
ui.foundation.core.UMLGeneralizableElementAbstractCheckBox
ui.foundation.core.UMLGeneralizableElementLeafCheckBox
ui.foundation.core.UMLGeneralizableElementRootCheckBox
cognitive.ui.WizDescription
cognitive.ToDoList
foundation.core.UMLModelElementVisibilityRadioButtonPanel
ui.ActionCollaborationDiagram
ui.UMLRadioButtonPanel
explorer.rules.GoClassifierToBehavioralFeature
diagram.ui.ActionAddAttribute
diagram.ui.ActionAddOperation
ui.ActionGenerateOne
ui.ActionSequenceDiagram
ui.TabConstraints
explorer.rules.GoBehavioralFeatureToStateDiagram
explorer.rules.GoBehavioralFeatureToStateMachine

Table 1. 50 most important classes from ArgoUML

References

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

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

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

[4] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998.

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

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

[7] M. A. Foltz. Dr. jones: A software archaeologist's magic lens, 2002. http://citeseer.nj.nec.com/457040.html.

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

[9] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 314–323. IEEE Computer Society, 2005.

[10] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 112–121. IEEE Computer Society, 2005.

[11] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu. Challenges and requirements for an effective trace exploration tool. In *Proceedings of the 12th International Workshop on Program Comprehension (IWPC'04)*, pages 70–78. IEEE, 2004.

[12] D. Jerding and S. Rugaber. Using visualization for architectural localization and extraction. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, 1997.

[13] D. F. Jerding and J. T. Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):257–271, 1998.

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

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

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

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

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

[19] T. Systä. Understanding the behavior of java programs. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 214–223. IEEE, 2000.

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

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

[22] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 134–142. IEEE Computer Society, 2005.