Interactive Exploration of Semantic Clusters

In proceedings of the International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)

Mircea Lungu[1], Adrian Kuhn[2], Tudor Gîrba[3] and Michele Lanza[4]

[1,4] Faculty of Informatics
University of Lugano, Switzerland

[2,3] Software Composition Group
University of Berne, Switzerland

Abstract

Using visualization and exploration tools can be of great use for the understanding of a software system when only its source code is available. However, understanding a large software system by visualizing only its lower level artifacts (e.g., classes, methods) and the relations between them does not scale for industrial-size systems. To address the scalability issue, higher level hierarchical abstractions (e.g., package structure, clustered decompositions of the system) should be used together with relations between them that are usually aggregated from the lower level relations. In this paper, we present the concepts behind Softwarenaut, a tool aimed at exploring any kind of hierarchical decompositions of a system, and then we look at a specific exploration of a system. In the experiment, the hierarchical decomposition of the system is the result of applying a semantical clustering to group classes that use similar terms.

Keywords: software exploration, visualization, clustering, LSI

1 Introduction

When only the source code is available, recovering the architecture of a large software system is a difficult task because of its sheer size and complexity. Considering that even in the presence of a well-thought initial design the evolutionary processes, such as bug fixing and feature additions, lead to a decay of both the architecture and the source code itself [3], the difficulty of understanding a legacy system only by analyzing the code becomes obvious.

One approach to software reverse engineering is the use of visualization techniques to represent the software entities and their relationships [14, 13, 7]. However, understanding a large software system by visualizing only its lower level artefacts (*e.g.,* classes, files) and the relations between them does not scale for industrial sized systems. To address the scalability issue we use higher level hierarchical abstractions (*e.g.,* package structure) and relations between them. When there are no explicit relationships between abstractions we aggregate them from the lower level relations. Furthermore, we present the reverse engineer with several integrated complementary views of the system: a view of the current high-level focus, a map for showing the location of the current focus and a detailed view of a selection.

Another widely used approach in reverse engineering is clustering [5, 10]. The clustering techniques provide an automatic way of separating a complex system in simpler components. For example, in the case of hierarchical clustering the system is decomposed into hierarchical decompositions that have to be manually inspected for the right abstraction level to be detected.

In this article we present an approach to interactively explore the hierarchical clusters given by the classes that use similar terms. The approach is based on Softwarenaut, an environment for the interactive, visual exploration of any hierarchical decomposition of a software system. In the particular case of the article, the hierarchical decomposition that we will use will be provided by Hapax, our semantic analysis framework. For the semantic clustering Hapax uses an information retrieval technique called Latent Semantic Indexing (LSI) [6]. The user can interact with, and navigate the visualizations of the semantical clusters, aided by complementary lower level information about the properties and interconnections between the components of the clusters.

Structure of the paper. We start by presenting the model for the hierarchical structures that can be explored with the techniques presented in this article. In Section 3 we describe the visualization and exploration techniques that

are employed in Softwarenaut. We continue in Section 4 on how we did extract the semantic information using Hapax. Section 5 shows how we applied the approach on a case study while Section 7 concludes and presents future work.

2 The Underlying Model

Softwarenaut is meant to be used for the exploration of any hierarchical decomposition of a system that conforms to the model presented in the diagram from Figure 1.

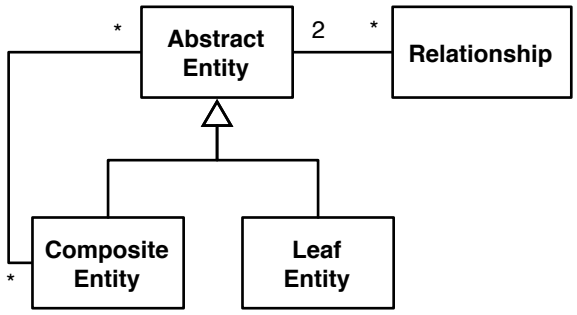

Figure 1. A decomposition of a system that conforms to the model, can be explored using Softwarenaut.

The three types of entities in the diagram are:

- *Leaf Entities.* The leaf entities are the basic programming language blocks used for the structuring of the software (*e.g.,* functions, classes or files).
- *Relationships.* These are the relations between two entities. They can be relationships from the source code like include relations for files, inheritance relations for classes and function calls for methods and functions.
- *Composite Entities.* The composite entities are containers for other abstract entities that are not necessarily supported by the programming language (*e.g.,* packages, namespaces, clusters).

The model admits relationships between any abstract entities. However, in software systems explicit relationships usually exist only between the leaf entities. Therefore, the relations between the composite entities are inferred bottom-up from the relations existing between the leafs.

Although the techniques presented in this article can work with any model of a system that conforms to the previously presented schema, for the remaining parts of the article, we will limit the discussion to the specific case of a clustered decomposition based on semantic analysis. In this case, the leafs are classes in the system, the relations are dependency relations between classes refined from the method calls and the composites are the result of clustering the classes by using hierarchical clustering based on LSI.

3 The Three Perspectives

Our exploration environment approach is based on offering the reverse engineer several interactive complementary visualization perspectives of the module structure of a system, integrated in one window. The views corresponding to the different perspectives are coupled, meaning that when the user operates a modification in one view, the complementary views are also updated. Moreover, the visualizations are supported by complementary information in the form of software metrics and other semantic information about the currently inspected part of the system.

Figure 2 shows the Softwarenaut tool visualizing the semantic clusters of an example software system. The window presents three views: Exploration View, Map View and Detail View.

3.1 The Exploration View

The Exploration View presents a graph that can be interactively navigated by expanding, collapsing and filtering operations. The view uses nodes to represent the entities and edges to represent the dependency relationships between them. At any given moment, only a subset of the composites in the system and the interdependencies between them are visible, subset that we will call *the working set*. The elements of the Exploration View are:

The *nodes* are represented by square figures which have a given metric mapped on their side and another on their shade of gray. In Figure 2 the size of the square is proportional to the number of lines of code defined in the cluster while the shade of gray is an expression of the semantic cohesiveness of the contained classes: the darker the cluster, the more cohesive the contents.

The *edges* are aggregations of the invocations between the contents of the corresponding composites. They have metrics mapped on their width and shade. In Figure 2 the width is proportional to the number of methods abstracted in the edge, while the shade of grey is proportional to the percentage of accessor methods contained in edge.

Interacting with the Exploration View. The Exploration View provides operations for the filtering, expanding and collapsing of the nodes in the working set. The expanding operation replaces a composite with its direct descendants while the collapse operation replaces several compos-

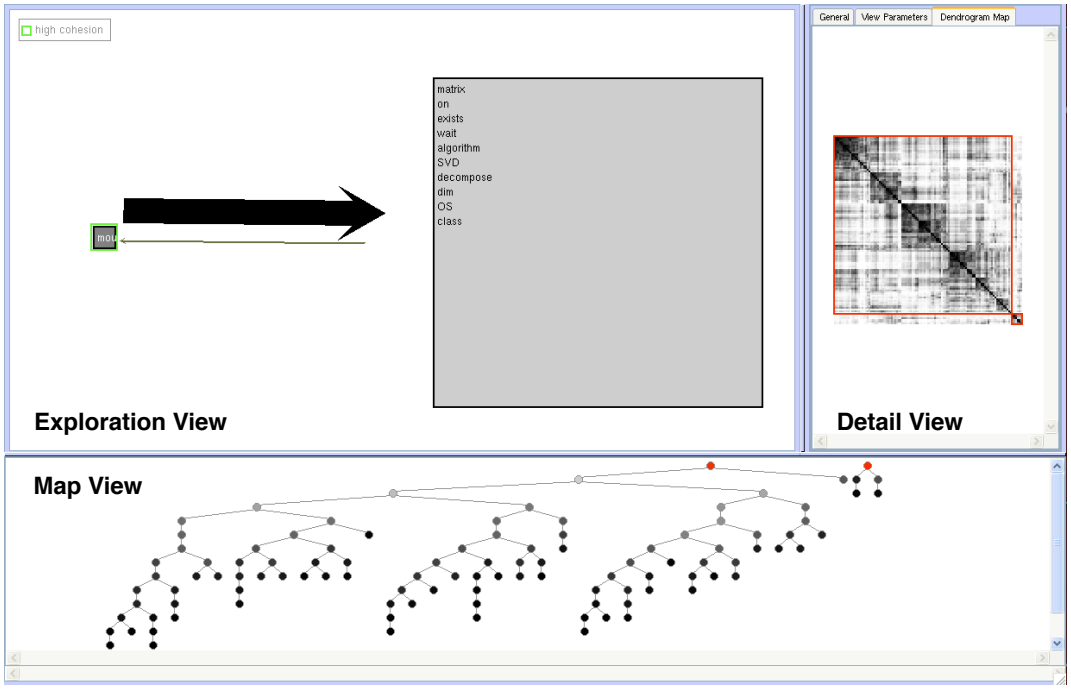

Figure 2. The Softwarenaut window to explore the semantic clusters. The top-left part of the window is the Exploration View, the top-right is the Detail View and the bottom is the Map View.

ites with their container. Filtering operations based on metrics and structural information are also available but they do not represent the focus of this article.

3.2 The Map View

The Map View displays a map of the cluster hierarchy and keeps the user oriented during the exploration by marking the visible nodes on the map. In the figures, the visible nodes are marked in red on the map. Moreover, when a node is selected in the main view, its descendants are colored with shades of gray corresponding to the semantic cohesion of the contents.

There are several reasons why the information presented in the exploration view needs to be seen in the context of the entire system's cluster containment tree (hereafter referred to as *system's tree map*).

The different nesting levels of the displayed clusters. At a given moment, as a result of the exploration, the clusters in the exploration view will be at various depth levels in the system's tree map. This is not visible in the exploration view, therefore, one responsibility of the map view is to represent the system's tree map and to emphasize on it the composites that are currently displayed in the exploration view. In Figure 2, on the map view, the currently displayed composites are colored in red.

The high abstraction levels of the edges. Each composite in the exploration view represents the whole containment hierarchy under it, and each edge represents an abstraction of the invocations between the containment hierarchies of the two adjacent composites. While it is valuable to know which specific descendants in the hierarchies adjacent to an edge are involved in the invocations abstracted by the edge, it is too expensive to expand the corresponding nodes to determine this fact: A better solution is to emphasize the descendants that are implied by the edge on the system's map when an edge is selected.

The abstraction level of the composites. In the exploration view, a composite encapsulates a whole hierarchy, hiding therefore valuable information about its structure. This information can be discovered by expanding the composite, but when only an overview is needed, expanding a whole subtree is an operation which costs too much. Moreover, in the expanding process the overview aspect is lost. A solution that takes advantage of the fact that the map view already contains a repre-

sentation of the system's tree map, is to highlight on it the descendant tree corresponding to the composite currently selected. Besides highlighting, more properties of the descendants can be displayed. In our case for each descendant, we map its semantic cohesion on its color in the map view.

3.3 The Detail View

The Detail View presents details for the entity that is selected in the Exploration View or, alternatively, a map of the whole system when no entity is selected. Because there are many types of details that can be displayed for an entity, they are implemented as plugins. The system searches for the available list of plugins for the type of the selected entity and displays all the available plugins in separate tab panels. Figure 3 presents the contents of such a tab panel that shows the semantic terms in a cluster ordered by their relevance.

4 Semantic Clustering in a Nutshell

To provide a hierarchical decomposition of a system for the exploration system, we have implemented the following steps in Hapax:

1. First, we preprocess the source code to obtain a term-document matrix. The input data is the source code, broken into pieces at an arbitrary level of granularity (*e.g.,* modules, classes, methods *etc.*) to define the documents used by LSI. The terms are all words found in the source, except keywords of the programming language.

2. LSI is then applied to the term-document matrix to build an index with semantic relationships. From this index we can compute the semantic similarity between both software artifacts and terms. More in-depth information on using LSI is given in [2].

3. To better understand this semantic correlations, we group the software artifacts with a hierarchical clustering algorithm. This algorithm creates a dendrogram: A hierarchical tree with clusters as its nodes and the documents as its leaves [4]. Thus we get a hierachy of nested semantic concepts, ranging - from root to leaves - from general concepts to more specific concepts. The deeper a cluster is nested in the tree, the more semantical similar are the software artifacts it contains.

4. Finally we use the LSI index to provide automaticaly retrieved labels, describing the clusters and concepts. More on using LSI to perform search queries, see [1].

5 Experiment

To show how we use the approach, we present here a first experiment we have carried out to study the Hapax tool. Hapax is the semantic analysis tool we have built. It is written in Smalltalk and has 100 classes and 1300 methods. The experiment was carried out by one of the authors that did not know the code and then the lessons learnt were cross-checked with the author of the code. In the followings we report on the steps of the analysis.

Step 1: Discovering the UI. Figure 2 presents the first view we get on the system: a small cluster using in a very intensive way a much bigger one. Looking at the detail panel which presents the significant terms in the small cluster we see "mouse click sensor active open . . . view . . . ". We infer that we have detected a UI cluster which uses intensively the model classes in the big cluster. By looking at the classes in the cluster the supposition is validated.

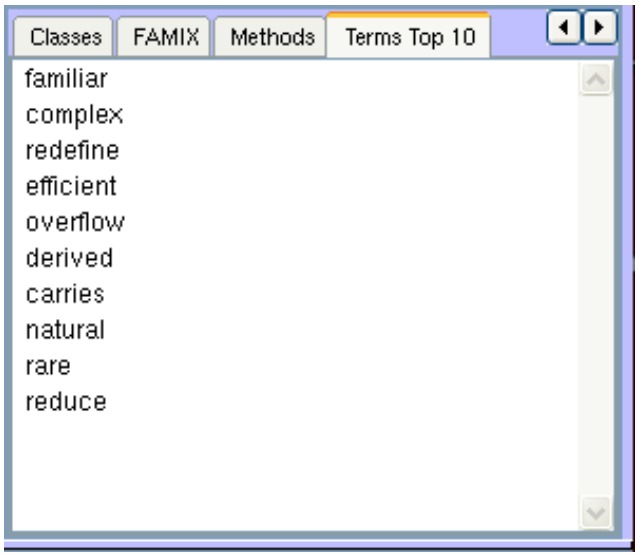

Figure 3. The terms in the detail panel show the fact that the selected cluster is related to mathematical notions

Step 2: Discovering the Mathematical Extensions. We expand the big cluster and observe that it contains another two clusters out of which again a small and a big one. We inspect the terms in the small cluster and discover mathematical-like terms (see Figure 3): "complex, overflow, natural, carries". From the detail panel containing the classes in the cluster we go to the source code of the classes and see that they are just extensions to some

Smalltalk classes[1]. We delete the cluster as non-relevant for the overall architecture.

Step 3: Intermediary View. Because the big cluster in the view has a low semantic cohesion (*i.e.,* the node appears light grey) we expand it again (see the result in Figure 4). The next clusters are comparable in size. We look at the first and see the terms "matrix decompose . . . algorithm " that look related but are interwoven with some other like: "SVD, svlib" which do not fit, so we will expand this node to obtain more cohesive clusters.

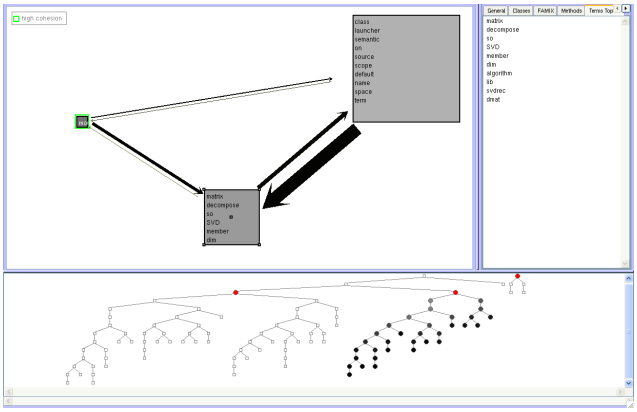

Figure 4. An intermediary view.

Step 4: Discovering the SVD Computation. Expanding gives two other clusters, out of which one is very cohesive around the terms "SVD exe lib win libSvd". Because the occurrence of OS-related terms is unexpected we inspect the classes that are contained in the cluster and find out that the algorithms that compute the Singular Value Decomposition (SVD) are in an external program.

The other cluster in the view is not very cohesive, as can be seen from its light color, but the terms are strongly related: "matrix row . . . column triangle . . . ". Inspecting the contained classes in the detail panel, we see that most of the classes are part of the same module. Considering that the semantic cohesion (seen in the color of the node) is also high, we conclude that the cluster is self-contained and should not be explored further, and direct our attention towards the big remaining cluster.

Step 5: Finalizing the Analysis. One of the two resulting clusters is very cohesive around the terms: "term semantic latent space . . . ". We infer the cluster captures the entities that model the LSI concepts. Looking in the map view, the

[1]In Smalltalk, extensions to the existing classes can be defined, which, while loaded in the system, add functionality to the original classes

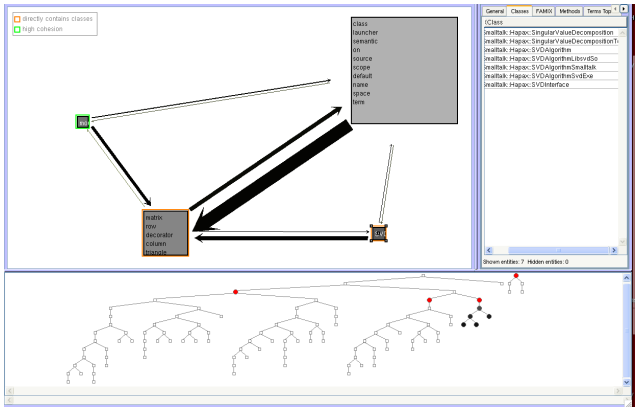

Figure 5. The Detail Panel shows the classes contained in the cluster which computes the Single Value Decomposition

colors of the descendants show that there is not any gain in cohesiveness if we explore this cluster any further; so we focus our attention on the other cluster. After expanding it we observe two very cohesive clusters.

The first is grouped around the terms "java token parser stem". Inspecting the edges that come and go from the cluster (see Figure 6) we see that it uses a lot of string processing which when correlated with the term "stem" makes us believe that the classes in the cluster are responsible with preprocessing the terms for suffix removal.

After inspecting the second cluster we observe that it contains a collection of utilities and some extensions.

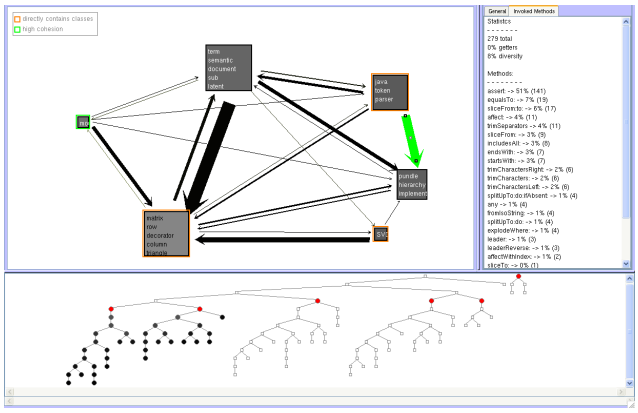

Figure 6. The main components of the Hapax system are visible in the Main View

The relations in the view need to be inspected and some might need to be filtered as they might not be representative

for the interactions between the detected clueters. However, this is beyond the scope of the article.

6 Related Work

Our work expands over two active fields: Semantic analysis and interactive visualization. A first attempt to combine the two fields was done by Wong who took into account the prefixes of the files when clustering them into subsystems that were visualized by Rigi [16]. Maletic and Marcus were the first to apply LSI for software reverse engineering [8]. They used LSI to analyze the semantic clusters of the files of Mosaic. Even if they only considered files, they showed the usefulness of information retrieval techniques in reverse engineering. The relation between the structure of the system and the semantical information was explored in a follow-up work by the same authors when they analyzed the same case study, only at the level of procedures [9]. Tools that use structural exploration are Rigi [14] and SHriMP [13]. Rigi differs from our approach by using a bottom-up approach to the architecture recovery and multiple windows for presenting different perspectives on the system. SHriMP supports a top-down approach to software exploration while employing a nested-graph visualization technique. The difference between SHriMP and our tool is that we use polymetric visualizations and use the Detail View for presenting the contents of the composites instead of representing the contents inside them.

7 Conclusions

When only the source code is available, reverse engineering a large software system is a difficult task because of its sheer size and complexity. Two of the methods used are interactive software exploration and clustering.

In this work we propose the use of an interactive navigation through the a hierarchical clustering of a software system based on semantical information. At any point in time, the reverse engineer is presented with a clustering structure that she can expand, delete or inspect. We also show the structural relationships (*i.e.,* invocations) between the clusters, thus providing the reverse engineer with more information to support the decision of navigation and inspection.

As a validation, we presented an initial result of applying the approach on a case study. We focused on the interactive nature of the process and showed for each step how the information presented guide the reverse engineering decisions.

Acknowledgments. Gîrba gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

References

[1] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. Technical Report UT-CS-94-270, 1994.

[2] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[3] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[4] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, Sept. 1999.

[5] R. Koschke. An incremental semi-automatic method for component recovery. In *Working Conference on Reverse Engineering*, pages 256–, 1999.

[6] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of Working Conference On Reverse Engineering (WCRE 2005)*, pages ??–??, Nov. 2005. to appear.

[7] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[8] J. I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000)*, pages 46–53. IEEE Computer Society, Nov. 2000.

[9] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the International Conference on Software Engineering (ICSE 2001)*, pages 103–112, 2001.

[10] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[11] M. Marchesi and G. Succi, editors. *Extreme Programming and Agile Processes in Software Engineering*. Springer, 2003.

[12] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, 2004.

[13] J. Michaud, M.-A. Storey, and H. Muller. Integrating information sources for visualizing Java programs. In *Proceedings of IEEE International Conference on Software Maintenance (ICSM'01)*, pages 250–259. IEEE, Nov. 2001.

[14] H. A. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86. IEEE Computer Society Press, 1988.

[15] M. M. Shinji Kawaguchi, Pankaj K. Garg and K. Inoue. Mudablue: An automatic categorization system for open source repositories. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC.04)*, 2004.

[16] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, Jan. 1995.