# Softwarenaut: Cutting Edge Visualization

Mircea Lungu *and* Michele Lanza
*Faculty of Informatics*
*University of Lugano, Switzerland*

## 1 Introduction

Many reverse engineering and program understanding tools support the static analysis of large software systems by providing exploratory analysis and visualization services. Most of these tools use graph-based representations, in which entities are represented as nodes and dependencies between them as edges.

Most of the tools also provide ways to navigate from an overview of the system to details on entities of interest. The way in which they provide the detail is by using either focus and context techniques or a separate detail perspective. However, although all of them provide details and alternate views on the modules in the graph, to our knowledge, none present detailed views for the dependencies between modules. Softwarenaut, our research prototype [**?**], offers the user the possibility of displaying various types of detailed views for the dependencies between modules (Figure 1).

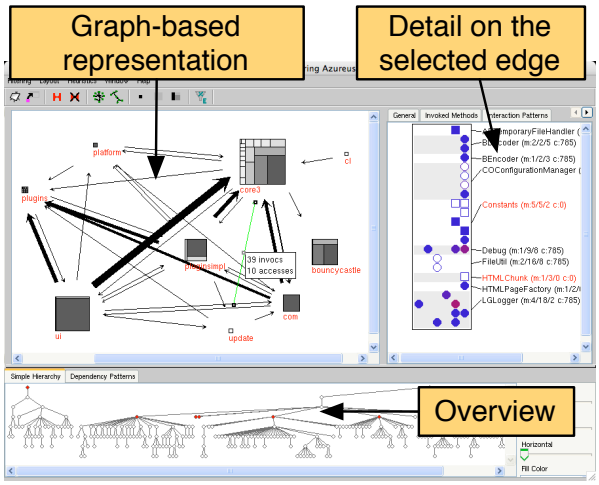


Figure 1: When a dependency edge is selected in the *graph-based representation*, the *detail panel* presents a suite of possible visualization perspectives on it

Providing detailed views on the dependencies is a first step towards understanding the structure and semantics of the dependencies between the modules and is useful during the maintenance phase. The poster presents several visual perspectives that Softwarenaut offers for the analysis of intermodule dependencies.

## 2 Perspectives on a Dependency Edge

We define a dependency between two modules as being an aggregation of the low-level dependencies between the basic elements contained in the modules such as method calls, variable access and inheritance relationships. Such a dependency can abstract a high number of invocations between two modules. For brevity, we call the module which provides functionality *provider module* and the module which uses this functionality *client module*.

Softwarenaut implements various detail perspectives on dependencies. For a first impression on the dependency statistics on the method calls can be displayed. Statistics include the percentage of accessors in the provider module's interface, various metrics computed on the involved methods from the client and provider or a histogram of the size of the methods.

**Cutting Edge Visualization.** One way in which structural information about the dependency between two modules can be displayed is using a dependency matrix[1]. Every row in the matrix corresponds to a method from the provider module which is invoked by any of the classes in the client module; every column corresponds to one such invoking class.

The granularity on the provider side is more fine grained (i.e. methods versus classes) because this helps uncover dependency patterns which would otherwise not be visible. The methods are grouped by classes to favor visual pattern detection and the classes have alternating gray and white backgrounds to emphasize the delimitations between them (See Fgure 2). The class names are visible on the right side of the matrix.

To put the classes with similar patterns of interaction closer on the x axis we use hierarchical clustering: we consider that every class has a signature which is a vector with a position corresponding to every provider method and on each such position a 1 denotes that there is a dependency between the class and method or 0 otherwise. The classes are clustered based on the distance between the vector space defined by the signatures and the order of the classes on the x axis is the result of traversing the dendrogram in preorder. This makes the classes which have similar access patterns to the provider methods to be contiguous on the x-axis. One such example is the classes which use the methods marked as (4) in Figure 2.

On top of the incidence matrix properties of the involved methods are superimposed using colors. Every intersection point in the matrix visually encodes properties about the method, such as:

- *The number of invocations* is presented using a heat map where blue means low invocation count and red means high invocation count.
- *The structural properties of a method* such as if it an accessor or not are encoded using the presence or absence of the fill property of the figures (accessors are empty).
- *The semantical properties of a method* are represented using shape. Utility methods (i.e. methods used by multiple classes) are represented as circles.

[1] We called this view a "cutting edge visualization" because it represents the relationships between two modules without using nodes and edges, so it effectively *cuts* the edges from the picture

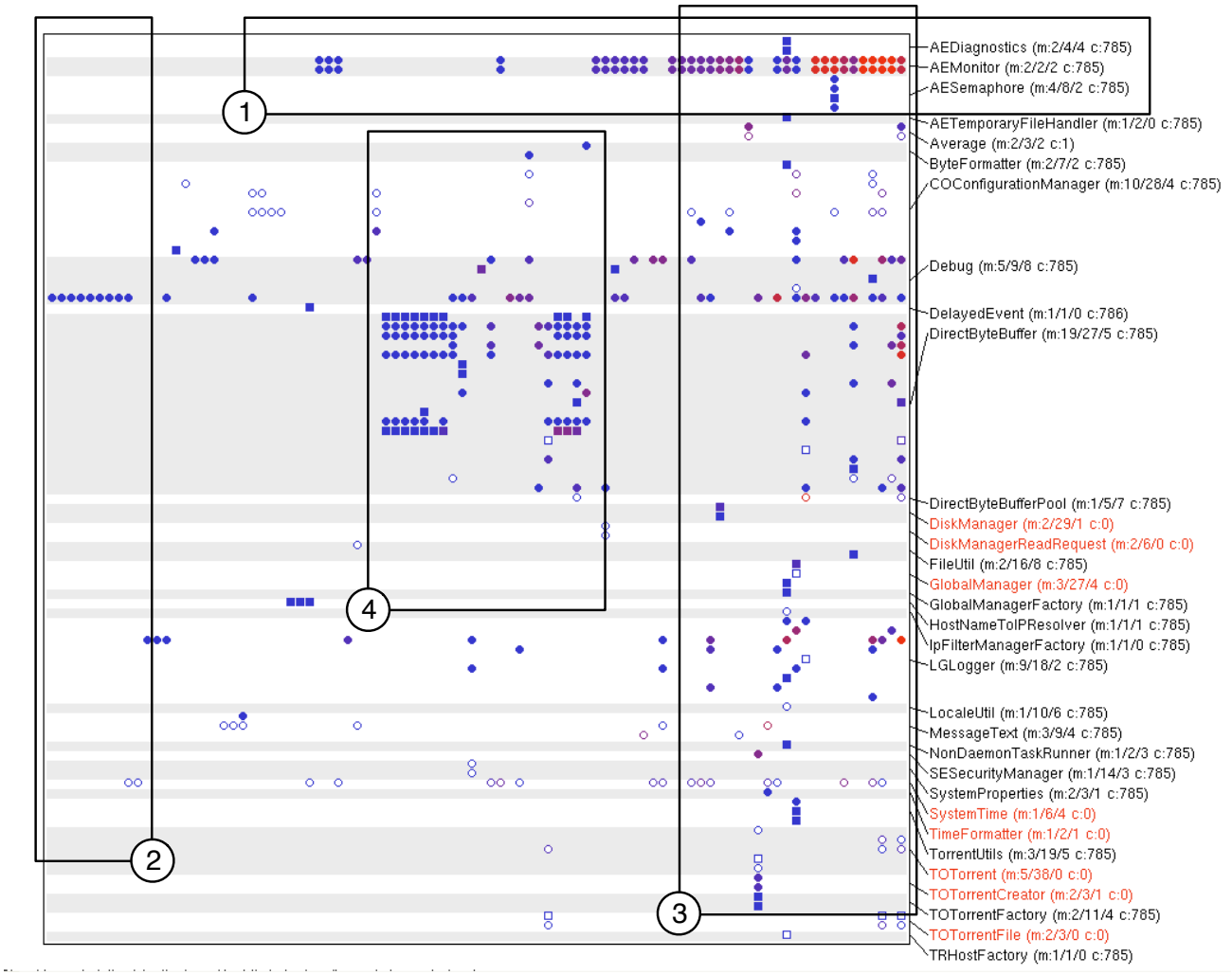


Figure 2: The dependency between the modules com and org.gudy.azureus.core3 in Azureus.

**Reading the Matrix.** Figure 2 presents details of the dependency between *com* and *org.gudy.azureus.core3* two of the modules in the Azureus system that we analyzed. The figure illustrates two of the properties that a dependency matrix constructed as shown in the previous section presents:

- The client classes which are strongly coupled with the provider package are positioned at the right side of the figure while the classes which are loosely coupled are positioned towards the left (marks (2) and (3) in Figure 2). Mark (2) from Figure 2 presents a group of classes which depend only on the *printStackTrace* method.
- Patterns of similar invocations appear as parallel point lines in the matrix. Figure 2 shows one such pattern annotated with (1): two methods are always called together by all the classes that use them. At a closer inspection we see that the two methods are *enter* and *exit* from the *AEMonitor* class which implements a synchroniz ation mechanism specific to the project.

**Interaction.** The visualization is closely integrated with the analysis environment and every visual element has as model an associated software entity. Contextual menus for the elements allow for navigation either to code level or to other predefined visualizations.

**Epilogue.** We are currently working on extending the visualization technique towards displaying information about the evolution of a dependency relation between two modules for the cases where multiple versions of the system are available for analysis. In a different direction, we are interested whether there exist dependency patterns between modules and wether we can define criteria for classification of these intermodule dependencies.