CodeCrawler - An Information Visualization Tool for Program Comprehension

Michele Lanza
Faculty of Informatics, Univ. of Lugano
michele.lanza@unisi.ch

Stéphane Ducasse
Software Composition Group, Univ. of Bern
ducasse@iam.unibe.ch

Harald Gall
Insitute of Informatics, Univ. of Zurich
gall@ifi.unizh.ch

Martin Pinzger
Insitute of Informatics, Univ. of Zurich
pinzger@ifi.unizh.ch

ABSTRACT

CODECRAWLER (in the remainder of the text CC) is a language independent, interactive, information visualization tool. It is mainly targeted at visualizing object-oriented software, and has been successfully validated in several industrial case studies over the past few years. CC adheres to lightweight principles: it implements and visualizes *polymetric views*, visualizations of software enriched with information such as software metrics and other source code semantics. CC is built on top of Moose, an extensible language independent reengineering environment that implements the FAMIX metamodel. In its last implementation, CC has become a general-purpose information visualization tool.

Categories and Subject Descriptors: D.2.7 Distribution, Maintenance, and Enhancement: Restructring, reverse engineering, and reengineering

General Terms: Measurement.

Keywords: Information Visualization.

1. INTRODUCTION

CC is a software and information visualization tool which implements polymetric views, lightweight 2D- and 3D- visualizations enriched with semantic information such as metrics or information extracted from various code analyzers [6].

It relies on the FAMIX metamodel [1] which models object-oriented languages such as C++, Java, Smalltalk, but also procedural languages like COBOL. FAMIX has been implemented in the Moose reengineering environment that offers a wide range of functionalities like metrics, query engines, navigation, etc. [2].

We shortly introduce the principles of polymetric views and then give some examples of the visualizations that CC enables the user to achieve. The proposed visualizations support both program comprehension and problem detection.

2. POLYMETRIC VIEW PRINCIPLES

The visualizations implemented in CC are based on the polymetric views described by Lanza [4, 6]. The principle is to represent source code entities as nodes and their relationships as edges between the nodes, but to use figure shapes to convey semantics about the source code entities they represent.

In Figure 1 we see that, given two-dimensional nodes representing entities and edges representing relationships, we enrich these visualizations with up to 5 metrics: (1+2) *Node Size.* The width and height of a node: The wider and the higher the node, the bigger the measurements its size is reflecting. (3) *Node Color.* The color interval between white and black. Here the convention is that the higher the measurement the darker the node is. (4+5) *Node Position.* The X and Y coordinates of the position of a node. This requires the presence of an absolute origin within a fixed coordinate system, therefore not all views can exploit such metrics (for example in the case of a tree view, the position is intrinsically given by the tree layout and cannot be set by the user).

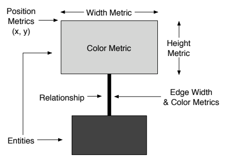

Figure 1: The principles of a polymetric view.

The polymetric views in CC can be created either programmatically in Smalltalk by constructing the view objects, or over an easy-to-use View Editor.

3. EXAMPLE POLYMETRIC VIEWS

Coarse-grained views. Such views are targeted at visualizing very large systems (*e.g.,* over 100 kLOC to several MLOC). In Figure 2 we see a *System Complexity* view of a single hierarchy which makes 10% of a 1.2 million lines of C++ code. The view uses the number of methods for the width and height of the class nodes. We gather for example from this view that there are classes with several hundreds of methods (at the bottom), while at the top we see a large number of structs, identifiable by the fact that most of them do not implement any methods.

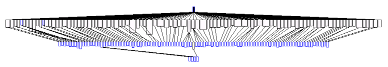

Figure 2: A System Complexity View of a 200-classes hierarchy from an industrial C++ system. It uses as width metric the number of attributes and as height metric the number of methods.

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.
ACM 1-58113-963-2/05/0005.

Fine-grained views. In Figure 3 we see a visualization of the internals of a small hierarchy of 4 classes. The class blueprint view helped to develop a pattern language [4]. In the present example we see the following patterns:

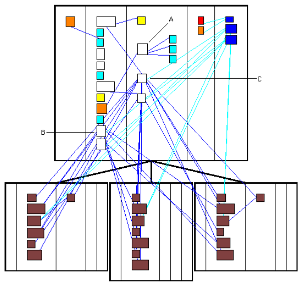

Figure 3: A *Class Blueprint* view on a small hierarchy of 4 classes written in Smalltalk.

Pure overrider: The three subclasses implement only overriding methods (brown color). *Siamese twin*: The two subclasses on the left and the right are structurally identical, not only do they implement the same methods (the methods differ within their bodies), their static invocation structure is also the same. *Template method*: The method node in the superclass annotated as *A* is a concrete method which only invokes abstract methods (cyan color). This is known as the *template method* design pattern. *Inconsistent accessor use*: The superclass defines only two accessors (second layer from the right), while it defines three attributes (rightmost layer). These two accessors do not have ingoing edges: In the context of this small hierarchy they are unused. *Direct attribute access*: The attribute nodes of the superclass are directly accessed by several methods. The methods annotated as *B* and *C* seem to play an important role in these classes: They are invoked by many methods (several ingoing edges) and they invoke several methods (numerous outgoing edges).

Evolutionary views. In Figure 4 we see an example [5] of such a visualization, which again allows us to develop a pattern language applicable in the context of software evolution:

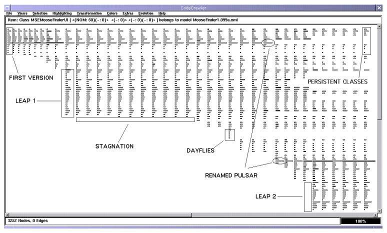

Figure 4: An *Evolution Matrix* view on 38 versions of an application written in Smalltalk.

The number of classes which survived the complete evolution of the system since the beginning is annotated as *persistent classes*. The *dayfly classes* existed during one version of the system and were then removed. Probably the developer tried out something implementation-wise and removed this 'experiment' right away. The *pulsar class* denotes a class whose size in terms of number of methods and attributes varies, making it thus an expensive class of this system. A long stagnation phase where the system did not grow in terms of number of classes, and two major leaps where the system rapidly grew between two versions.

Coupling Views. Recent work on CC was concerned with extending it to visualize polymetric views of several releases of a software system. The objective of these views is to highlight the coupling dependencies between modules of a software system. Couplings arise from structural dependencies between source code entities, such as includes, inheritance, invocations, and also from pairwise changes, logical couplings obtained from release history data as described in [3].

Lower-level information of source code entities and their coupling dependencies is condensed to different metrics that are mapped to graphical attributes and then visualized (left of Figure 5).

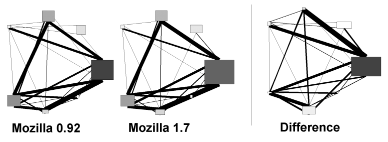

Figure 5: Left: A comparison of 7 Mozilla modules between release 0.92 (on the left) and release 1.7 (on the right). Right: Diff-graph between the releases.

The nodes represent modules with the number of classes (width), number of files (height), and number of directories (color). The edges represent abstracted invocation relationships between the modules (the width of the edges represents the weight, *i.e.,* the number of grouped function calls). This view highlights large modules and strong coupling dependencies between modules.

Differences between abstracted views are not straight forward to grasp with these graphs, because subsequent releases often lead to similar graphs. CC handles this problem by computing the difference between graphs of two selected releases on the basis of metric values. The differences between metric values of each node and edge attribute are computed. The right of Figure 5 depicts the diff-graph computed for the two Mozilla release graphs.

This graph highlights changes made to selected modules: The DOM module on the the right side increased by 150 classes and 95 source files. The coupling dependencies (*i.e.,* number of method calls) from the XML module in the upper left corner to the DOM module decreased by 142.

4. REFERENCES

[1] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[2] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a Collaborative and Extensible Reengineering Environment. In *Tools for Software Maintenance and Reengineering*. Franco Angeli, 2004.

[3] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[4] M. Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[5] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[6] 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.