CodeCrawler - Polymetric Views in Action

Michele Lanza
Software Engineering Group, University of Zurich, Switzerland
lanza@ifi.unizh.ch

Abstract

CodeCrawler is a language independent software visualization tool. It is mainly targeted at visualizing object-oriented software, and in its newest implementation has become a general information visualization tool. It has been validated in several industrial case studies over the past few years. CodeCrawler strongly 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.*

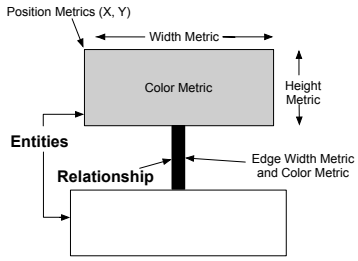

Figure 1. The principles of a polymetric view.

Introduction CodeCrawler implements is a software visualization tool which implements *polymetric views*, lightweight 2D- and 3D- visualizations enriched with semantic information such as metrics. It relies on the FAMIX Metamodel [1], which models object-oriented languages such as C++, Java, Smalltalk, and also procedural languages like COBOL. FAMIX has been been implemented in the *Moose* reengineering environment [2] which offers a wide range of functionalities like metrics, query engines, navigation, etc.

We shortly introduce the concept of the polymetric view and then give some examples of the visualizations that CodeCrawler enables the user to achieve.

The Principle of a Polymetric View The visualizations implemented in CodeCrawler are based on the polymetric views described by Lanza[7]:

In Figure 1 we see that, given two-dimensional nodes representing entities and edges representing relationships, we enrich these simple visualizations with up to 5 metrics on the node characteristics and 2 metrics on the edge characteristics: (1+2) The width and height of a node can render two measurements. We follow the convention that the wider and the higher the node, the bigger the measurements its size is reflecting. (3) The color interval between white and black can display a measurement. Here the convention is that the higher the measurement the darker the node is. Thus light gray represents a smaller metric measurement than dark gray. (4+5) The X and Y coordinates of the position of a node can reflect two other measurements. 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 given by the tree layout and cannot be set by the user). (A) The width of an edge can render a measurement: the wider the edge, the higher the measurement. (B) The color interval between white and black can display a measurement. Here the convention is that the higher the measurement the darker the edge is.

In Figure 4 we see CodeCrawler visualizing itself with a polymetric view called *System Complexity*. The metrics used in this view are the number of attributes for the width, the number of methods for the height, and the number of lines of code for the color of the displayed class nodes.

Example Polymetric Views CodeCrawler visualizes three different types of polymetric views: coarse-grained, fine-grained, and evolutionary views.

Coarse-grained views. Such views are targeted at visualizing very large systems (*e.g.*, over 100 kLOC to several MLOC) [7].

In Figure 2 we see a *System Hotspots* view of 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.

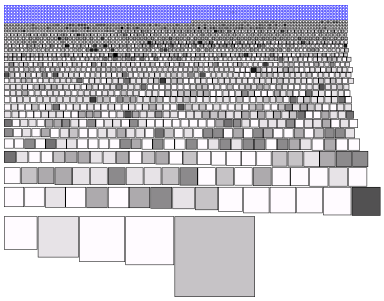

Figure 2. A *System Hotspots* view on 1.2 MLOC of C++ code. This view uses the following metrics: Width = height = number of methods, color = hierarchy nesting level.

Fine-grained views. The most prominent view is the *Class Blueprint* view, a visualization of the internal structure of classes and class hierarchies [6].

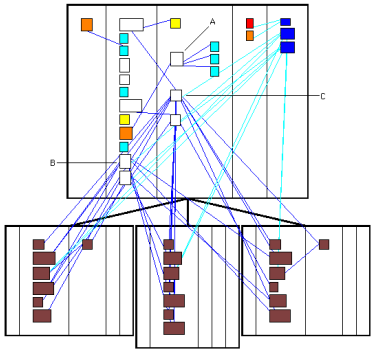

Figure 3. A *Class Blueprint* view on a small hierarchy of 4 classes written in Smalltalk.

In Figure 3 we see a class blueprint view of a small hierarchy of 4 classes. The class blueprint view helped to develop a pattern language [5]. In the present example we see the following patterns: *Pure overrider*: The three subclasses implement only overriding methods (denoted by the brown color). *Siamese twin*: The two subclasses on the left and the right are structurally identical, not only do they implement exactly the same methods (the methods differ within their body, of course), 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 (denoted by their cyan color). This is known as the *template method* design pattern [3]. *Inconsistent accessor use*: The superclass defines only two accessors (positioned in the second layer from the right), while it defines three attributes (last layer to the right). These two accessors do not have ingoing edges:at least in the context of this hierarchy they are not used at all. *Direct attribute access*: We see that 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. The most prominent view is the *evolution matrix* view, a visualization of the evolution of complete software systems [4].

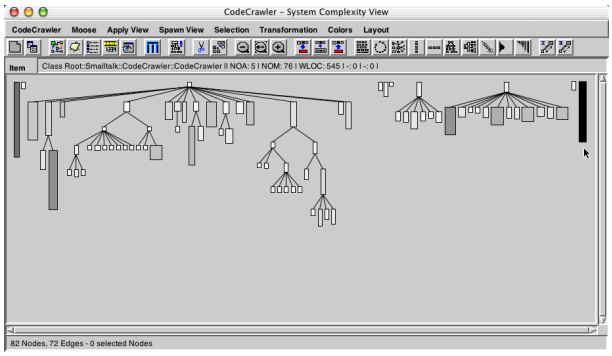

Figure 4. CodeCrawler visualizing itself with a *System Complexity* view. This view uses the following metrics: Width metric = number of attributes, height metric = number of methods, color metric = number of lines of code.

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, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[4] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, pages 37–42, 2001.

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

[6] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 300–311, 2001.

[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.