CodeCrawler - A Lightweight Software Visualization Tool

Michele Lanza
lanza@iam.unibe.ch
Software Composition Group - University of Bern, Switzerland

Abstract

CodeCrawler is a language independent software visualization tool. It is mainly targeted at visualizing object-oriented software, and in its newest implementation it 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*, lightweight visualizations of software enriched with semantic information such as software metrics and various source code information.*

1 Introduction

CodeCrawler is a lightweight software visualization tool, whose first implementation dates back to 1998 and it has been implemented as part of Lanza's Ph.D. thesis [4]. In the meantime it has been evolved into an information visualization framework, and has been customized to work in contexts like website reengineering and concept analysis. It keeps however a strong focus on software visualization. CodeCrawler is a language independent software visualization tool, because it uses the Moose reengineering environment [2] which implements the FAMIX metamodel [1]. The FAMIX metamodel models object-oriented languages such as C++, Java, Smalltalk, and also procedural languages like COBOL.

In the remainder of this short paper we introduce the concept of the polymetric view and then give some examples of the visualizations that CodeCrawler enables the user to achieve.

2 The Principle of a Polymetric View

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 these node characteristics:

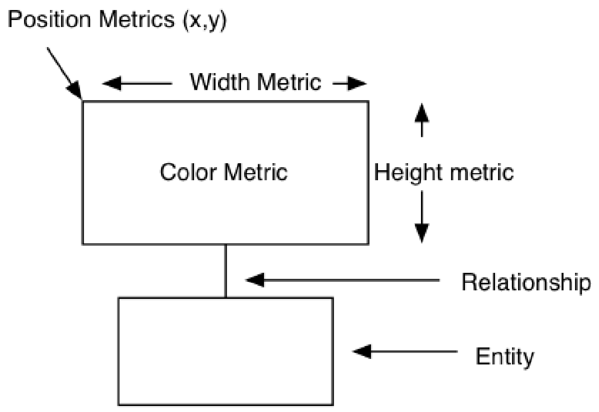
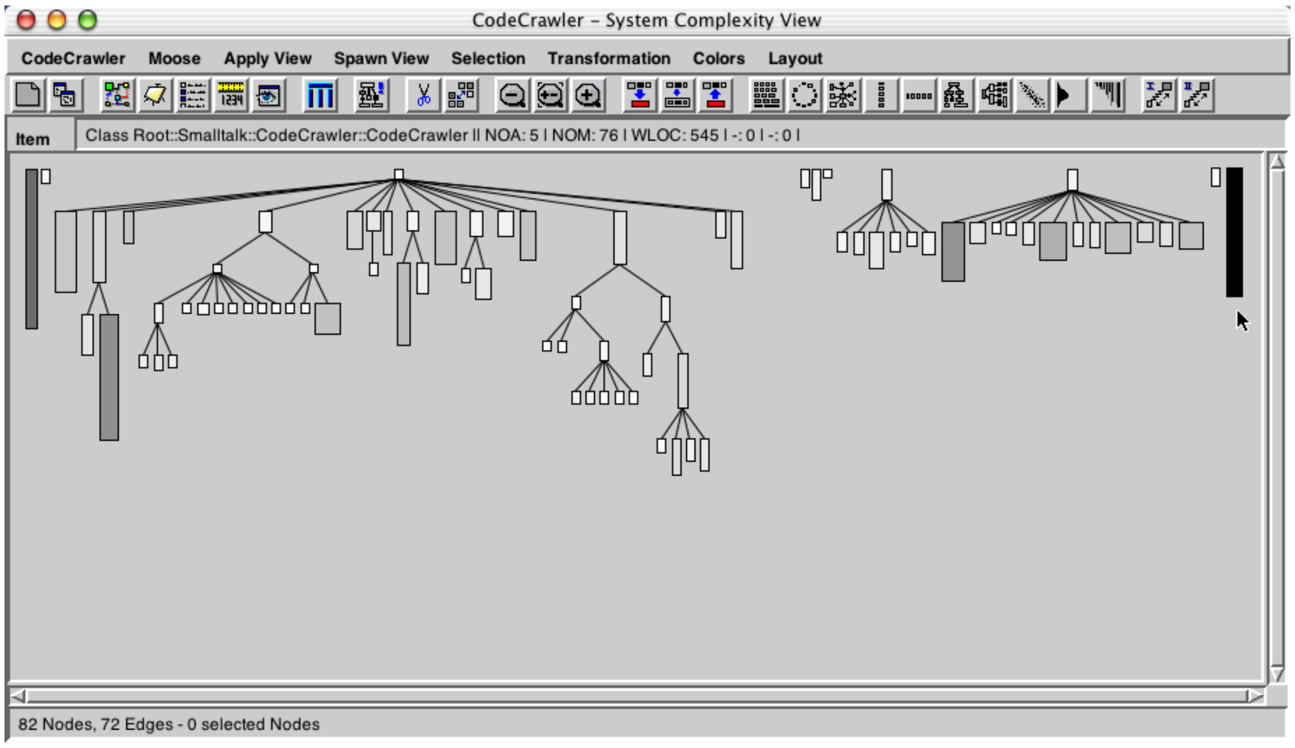


Figure 1. The principle of a polymetric view.

- *Node Size.* 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.
- *Node Color.* 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.
- *Node Position.* 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 intrinsically given by the tree layout and cannot be set by the user).

In Figure 2 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,



Class Root::Smalltalk::CodeCrawler::CodeCrawler || NOA: 5 | NOM: 76 | WLOC: 545 | -: 0 | -: 0 |

82 Nodes, 72 Edges - 0 selected Nodes

Figure 2. A screenshot of 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.**

the number of methods for the height, and the number of lines of code for the color of the displayed class nodes.

3 Example Polymetric Views

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

3.1 Coarse-grained views

Such views are targeted at visualizing very large systems (*e.g.*, over 100 kLOC to several MLOC). In Figure 3 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.

3.2 Fine-grained views

The most prominent view is the *Class Blueprint* view, a visualization of the internal structure of classes and class hierarchies [5].

In Figure 4 we see a class blueprint view 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:

- *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].

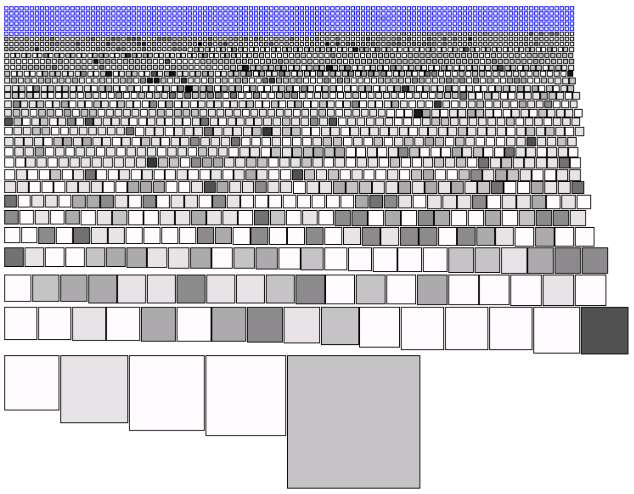


Figure 3. A *System Hotspots* view on 1.2 MLOC of C++ code. This view uses the following metrics: Width metric = height metric = number of methods, color metric = hierarchy nesting level (*i.e.,* how deep within a hiearchy a class resides).

- *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). Moreover, these two accessors do not have ingoing edges, which means that at least in the context of this small 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).

Please refer to [4] for a more in-depth discussion.

3.3 Evolutionary views

The most prominent view is the *evolution matrix* view, a visualization of the evolution of complete software systems [6].

In Figure 5 we see an example of such a visualization, which again allows us to develop a pattern language applicable in the context of software evolution. We can recognize the following patterns:

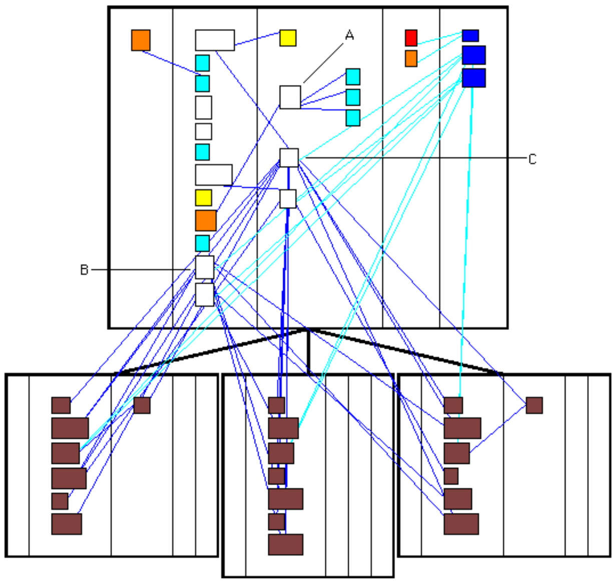


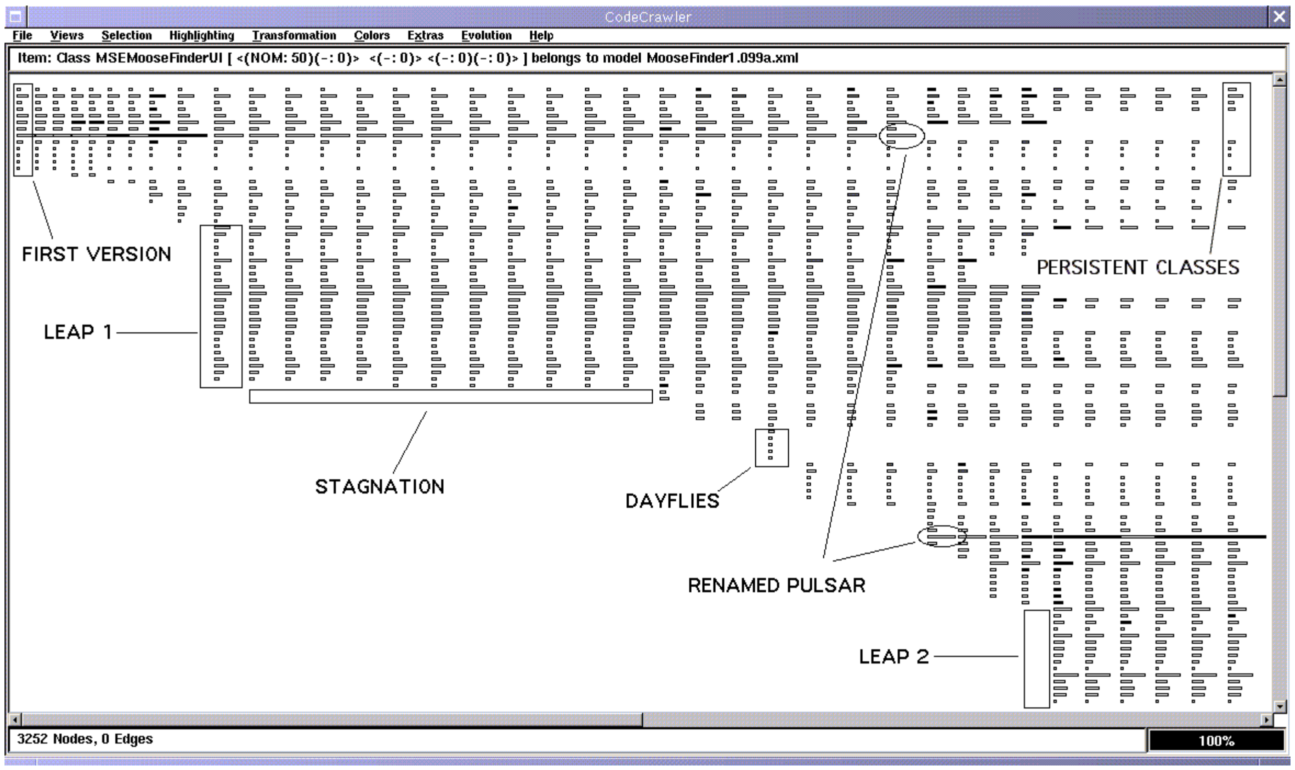
Figure 4. A *Class Blueprint* view on a small hierarchy of 4 classes written in Smalltalk.

- The number of classes which survived the complete evolution of the system since the beginning is annotated as *persistent classes*.
- The *dayfly classes* denote classes which have existed during one version of the system and have then be 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.

Please refer to [6] for a more in-depth discussion of the evolutionary views.

4 Features of CodeCrawler

Moreover, CodeCrawler features grouping support, customizable views, has been industrially validated, and is being used in software industry mainly by consultants. CodeCrawler is freeware and can be obtained at http://www.iam.unibe.ch/∼lanza/



Item: Class MSEMooseFinderUI [<(NOM: 50)(-: 0)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model MooseFinder1.099a.xml

3252 Nodes, 0 Edges

Figure 5. An *Evolution Matrix* **view on 38 versions of an application written in Smalltalk.**

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. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003. Recipient of the Denert-Stiftung Software Engineering Prize 2003.

[5] 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. 27 accepted papers on 145 (18%).

[6] 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.