CodeCrawler - An Information Visualization Tool for Program Comprehesion

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

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

Harald Gall, Martin Pinzger
gall,pinzger@ifi.unizh.ch
Department of Informatics
University of Zurich, Switzerland

ABSTRACT

CODECRAWLER is a language independent, interactive, 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 successfully 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. CODECRAWLER is built on top of Moose, an extensible language independent reengineering environment that implements the FAMIX metamodel. In its last implementation, CODECRAWLER has become a general-purpose information visualization tool.

1. INTRODUCTION

CODECRAWLER is a software and information visualization tool [11, 12] which implements polymetric views, lightweight 2D- and 3D- visualizations enriched with semantic information such as metrics or information extracted from various code analyzers [10].

It relies on the FAMIX metamodel [2] 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. [3].

We shortly introduce the principles of polymetric views and then give some examples of the visualizations that CODECRAWLER enables the user to achieve. The proposed visualizations support both program comprehension and problem detection, and target three different aspects of software systems, namely coarse-grained, fine-grained, and evolutionary aspects. We apply CODECRAWLER on itself and hightlight some of the implementation characteristics.

2. THE PRINCIPLES OF A POLYMETRIC VIEW

The visualizations implemented in CODECRAWLER are based on the polymetric views described by Lanza [7, 10]. 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.

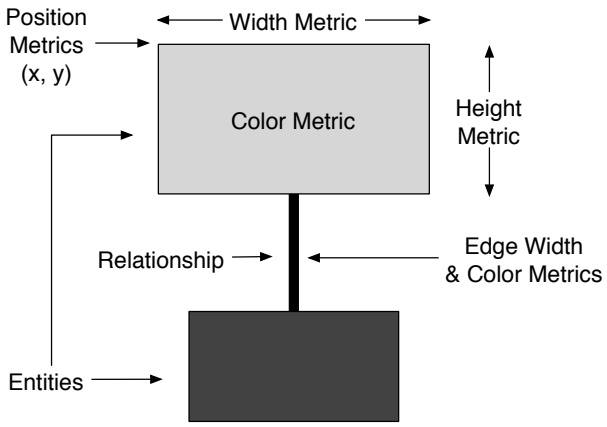

Figure 1: The principles of a polymetric view.

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:

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

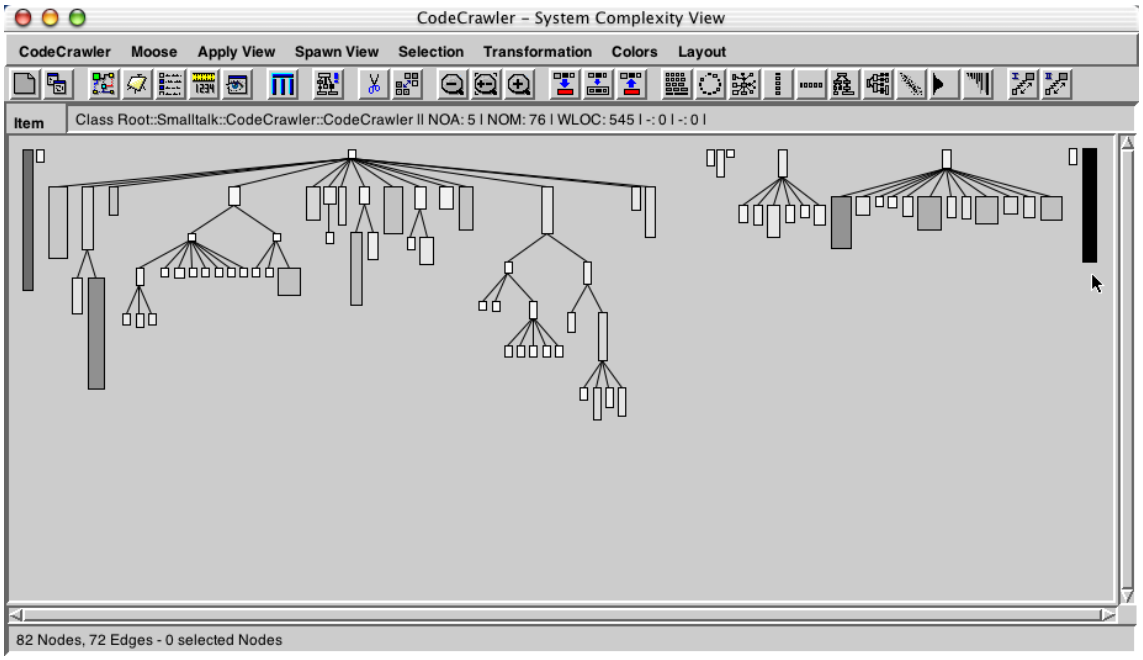

Figure 2: A screenshot of CODECRAWLER visualizing itself with a *System Complexity* view. This view uses the following metrics: Width metric = number of attributes, height metric = number of methods, color metric = number of lines of code.

The polymetric views in CODECRAWLER can be created either programmatically in Smalltalk by constructing the view objects, or over an easy-to-use View Editor, where each view can be composed using drag and drop. In Figure 3 we see CODECRAWLER's View Editor with the specification of the System Complexity view: the user can freely compose and specify the types of items that will be displayed in a view and also define the way the visualization will be performed: for every node and edge the user can choose from a selection of metrics.

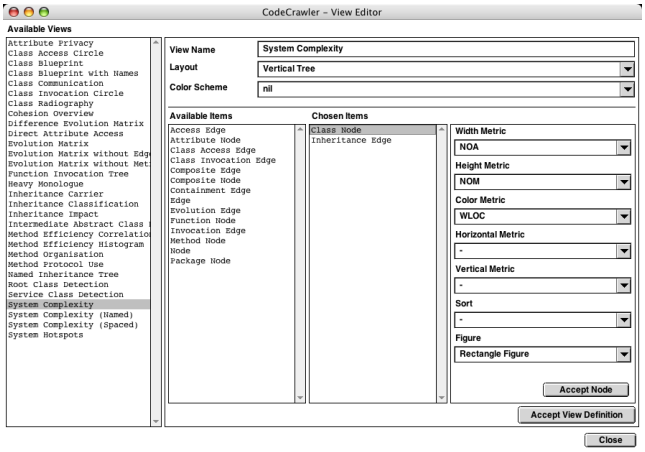

Figure 3: CODECRAWLER's View Editor.

3. EXAMPLE POLYMETRIC VIEWS

CODECRAWLER visualizes polymetric views that address different aspects of the implementation of software systems: 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 4 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.

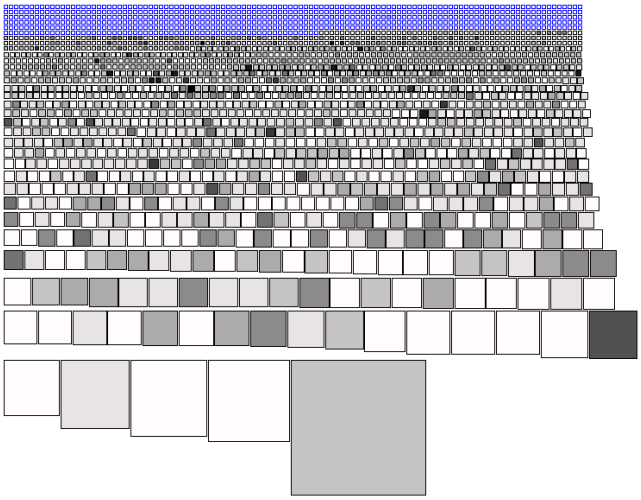

Figure 4: A *System Hotspots* view on 1.2 MLOC of C++ code. This view uses the following metrics: Width metric = height metric = number of methods, color metric = hierarchy nesting level (*i.e.*, how deep within a hiearchy a class resides).

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 [8].

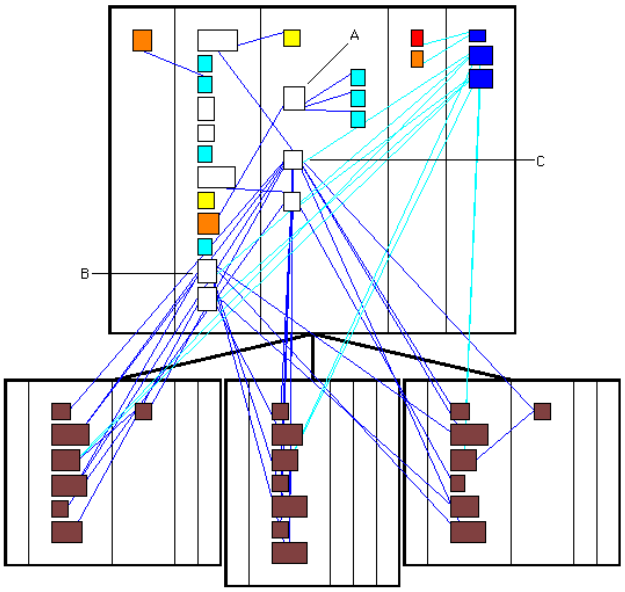

Figure 5: A *Class Blueprint* view on a small hierarchy of 4 classes written in Smalltalk.

In Figure 5 we see a class blueprint view of a small hierarchy of 4 classes. The class blueprint view helped to develop a pattern language [7]. 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 [6].
- *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 [7] 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 [9].

In Figure 6 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:

- 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 [9] for a more in-depth discussion of the evolutionary views.

3.4 Coupling Views

Recent work on CODECRAWLER was concerned with extending it to visualize abstract polymetrics 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 file includes, class inheritance, or method calls, and further from pairwise changes, so called logical couplings. Latter coupling is obtained from release history data as described in [5],[4].

Lower-level information of source code entities and their coupling dependencies is condensed to different metrics that are mapped to graphical attributes in the graph. Figure 7 depicts an example of an abstracted view of 7 Mozilla modules of two releases (0.92 on the left, 1.7 on the right) of Mozilla.

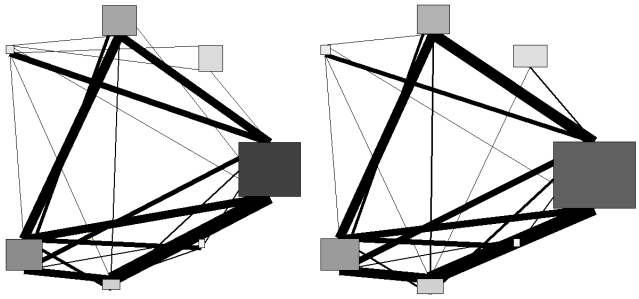

Figure 7: A comparison of 7 Mozilla modules between release 0.92 (on the left) and release 1.7 (on the right).

The nodes represent modules with the number of classes for the width, number of files for the height, and number of directories for the 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). Views clearly highlight large modules and strong coupling dependencies between modules.

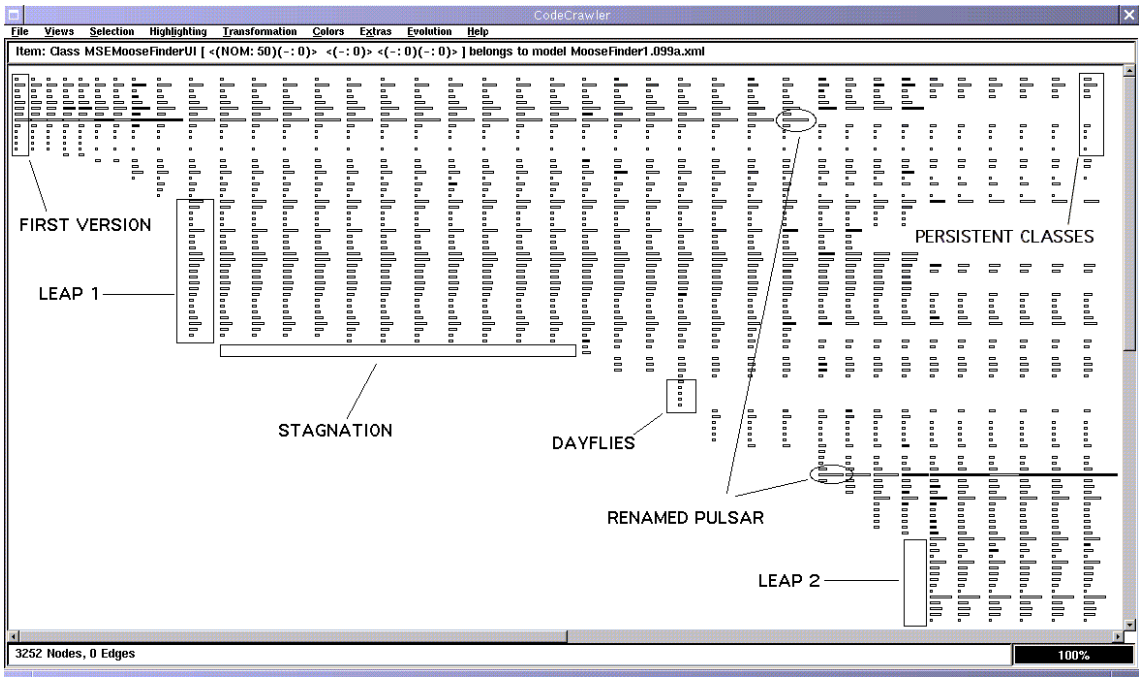

Figure 6: An *Evolution Matrix* view on 38 versions of an application written in Smalltalk.

However, differences between abstracted views are not straight forward to grasp with these graphs. Basically, because subsequent releases often lead to similar graphs. For example, try to compare the two graphs of Figure 7. CODECRAWLER handles this problem by computing the difference between graphs of two selected releases on the basis of metric values. Basically, the differences between metric values of each node and edge attribute are computed. Figure 8 depicts the diff-graph computed for the two Mozilla release graphs presented before.

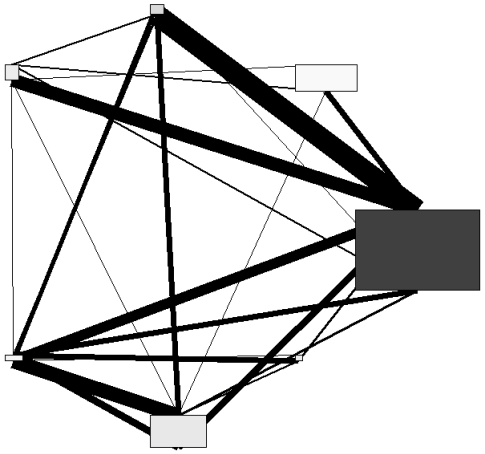

Figure 8: Diff-graph of 7 Mozilla modules between releases 0.92 and 1.7.

This graph clearly highlights the major changes made to selected modules. For instance, the DOM module on the the right side increased by 150 classes and 95 source files. In contrast, 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] G. Arévalo. X-Ray views on a class using concept analysis. In *Proceedings of WOOR 2003 (4th International Workshop on Object-Oriented Reengineering)*, pages 76–80. University of Antwerp, July 2003.

[2] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[3] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Reengineering Environments*. tba, 2004. to appear.

[4] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, Sept. 2003.

[5] 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.

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

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

[8] 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.

[9] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software

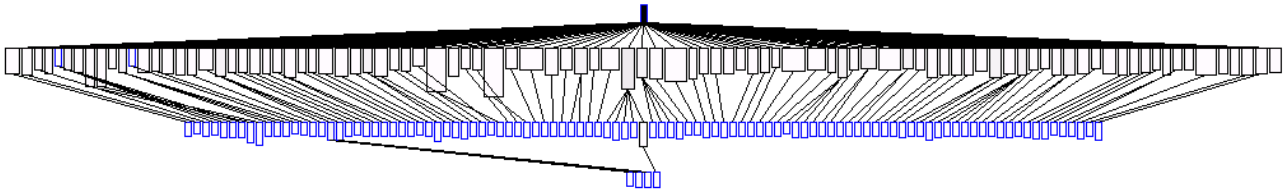

Figure 9: A Polymetric View of a 200-classes hiearchy from an industrial C++ system.

metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[10] 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.

[11] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[12] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.

[13] C. Wysseier. CCJun – polymetric views in three-dimensional space. Informatikprojekt, University of Berne, June 2004.

APPENDIX

A. PRESENTATION DESCRIPTION

We are going to use CODECRAWLER to reverse engineer a software system on-the-fly. The system in question will be a large Java or Smalltalk system ($>$ 100 kLOC). We are convinced this is the best way to show the capabilities of the tool and could give us important feedback on aspects that the audience likes or dislikes about the tool.

We start by using coarse-grained polymetric views (such as the one displayed in Figure 9) to get a first general impression about the system and use a combination of coloring and selection to tear the system apart in front of the audience, before diving into specifics with the more fine-grained views (such as the one displayed in Figure 5).

Thereafter we focus on evolution and show evoluationary views of the system pointing candidates of the different evolution patterns.

Finally, we present recent results we obtained from the case study we did with the Mozilla open source software project. We use CODECRAWLER to compute and present abstracted views on selected Mozilla modules and their coupling dependencies. Further, we demonstrate the diff-graphs to highlight the changes between different Mozilla releases.

Disclaimer. We are aware that this description is very short. However, our experience has shown that usually the audience catches on the process and gives its own suggestions about which parts of the system to explore, leading to an interesting and improvised demo session.

B. TOOL INFORMATION

CODECRAWLER's implementation started in 1998 as part of the Master and Ph.D. work of Michele Lanza, in the context of the European FAMOOS ESPRIT Project. It has been used for various industrial consultancy projects since its first implementation and has been re-implemented 4 times since then. In its newest implementation it has become a general information visualization tool (*e.g.*, visualization of concept lattices [1] and websites) and also supports 3D-Visualizations [13]. CODECRAWLER uses the HotDraw framework for the 2D visual output and the Jun framework for the 3D visual output. It uses the Moose reengineering environment for the data input. In Figure 10 we see a schematic description of CODECRAWLER s general architecture.

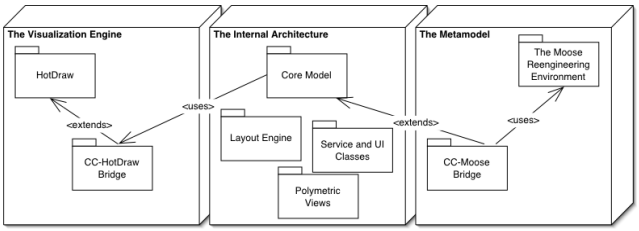

Figure 10: The general architecture of CODECRAWLER, composed of 3 main subsystems: the core, the metamodel, and the visualization engine.

B.1 Tool Availability

CODECRAWLER is implemented in Smalltalk under the BSD license: it is free and open source software. It runs on every major platform (Windows, Mac OS, Linux, Unix) and is freely available for download. Currently the webpage is located at:
`http://www.iam.unibe.ch/~scg/Research/CodeCrawler/`

Moreover, CODECRAWLER is also available as free goodie on the Visual-Works Smalltalk CD, a professional, commercial development environment developed and sold by the company Cincom which however also exists in a non-commercial version freely available for download at:
`http://www.cincomsmalltalk.com/`