The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques

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

ABSTRACT

One of the major problems in software evolution is coping with the complexity which stems from the huge amount of data that must be considered. The current approaches to deal with that problem all aim at a reduction of complexity and a filtering of the relevant information. In this paper we propose an approach based on a combination of software visualization and software metrics which we have already successfully applied in the field of software reverse engineering. Using this approach we discuss a simple and effective way to visualize the evolution of software systems which helps to recover the evolution of object oriented software systems.

Keywords

Software Visualization, Software Metrics, Reverse Engineering

1. INTRODUCTION

One of the major problems in software evolution is coping with the complexity which stems from the huge amount of data that must be considered.

A technique which can be used to reduce complexity is *software visualization*, as a good visual display allows the human brain to study multiple aspects of complex problems in parallel (This is often phrased as "One picture conveys a thousand words").

Another useful approach when dealing with large amounts of complex data are *software metrics*. Metrics can help to assess the complexity of software and to discover artifacts with unusual measurement values (i.e., in this context very large classes or subsystems, etc.).

In this paper we present a *combination* of these two approaches, with which we obtain the *evolution matrix*. It allows for a quick understanding of the evolution of an object-oriented system at system and class level.

We would like to stress that the approach presented here does not depend on a particular language, as our underlying metamodel is language-independent [5, 3]. However we present results obtained on Smalltalk case studies.

This paper is structured as follows: in the next section we present the evolution matrix and a categorization of classes based on their visualization within the evolution matrix. Afterwards we apply and discuss this approach in the section on the case studies. We then discuss shortly CodeCrawler and Moose, the tools used to generate the evolution matrix. We conclude the paper by discussing the benefits and limits of our approach, as well as related work, and by giving an outlook on our future work.

2. THE EVOLUTION MATRIX VIEW

In this section we present the *evolution matrix*. We first discuss the visualization technique we use and then show an example matrix. We then examine the characteristics of the evolution matrix. At the end of this section we introduce a categorization of classes based on their visualization within the evolution matrix.

2.1 Visualizing Classes using Metrics

We use two-dimensional boxes to represent classes and use the width and height of the boxes to reflect the metric measurements of the classes, as we see in Figure 1. This approach has been presented in [13] and [2]. In the evolution matrix discussed in this paper we visualize classes and therefore use the metrics *number of methods (NOM)* for the width and *number of instance variables (NIV)* for the height, although in our tool we can choose other metrics.

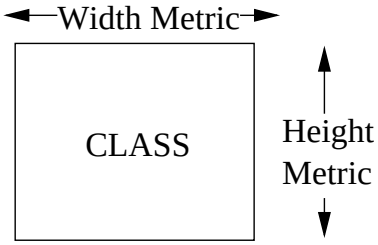

Figure 1: A graphical representation of classes using metrics.

2.2 Characteristics of the Evolution Matrix

The evolution matrix displays the evolution of the classes of a software system. Each column of the matrix represents a version of the software, while each row represents the different versions of the same class. The columns are sorted alphabetically. We see a schematic evolution matrix Figure 2.

The evolution matrix allows us to make statements on the evolution of an object oriented system at two granularity levels, which we discuss below: System Level and Class Level.

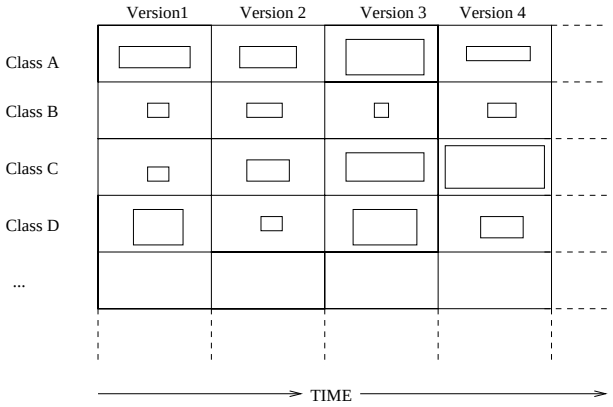

Figure 2: A schematic display of the Evolution Matrix.

2.2.1 Characteristics at System Level

As we see schematically in Figure 3 at system level we are able to recover the following information regarding the evolution of a system:

- **Size of the system.** The number of present classes within one column is the number of classes of that particular version of the software. Thus the height of the column is an indicator of the system's size in terms of classes.

- **Addition and removal of classes.** The classes which have been added to the system at a certain point in time can easily be detected, as they are they are added at the bottom of the column of that version.

  Removed classes can easily be detected as well, as their absence will leave empty space on the matrix from that version on.

- **Growth and stagnation phases in the evolution.** The overall shape of the evolution matrix is an indicator for the evolution of the whole system. A growth phase is indicated by an increase in the height of the matrix, while during a stagnation phase (no classes are being added) the height of the matrix will stay the same.

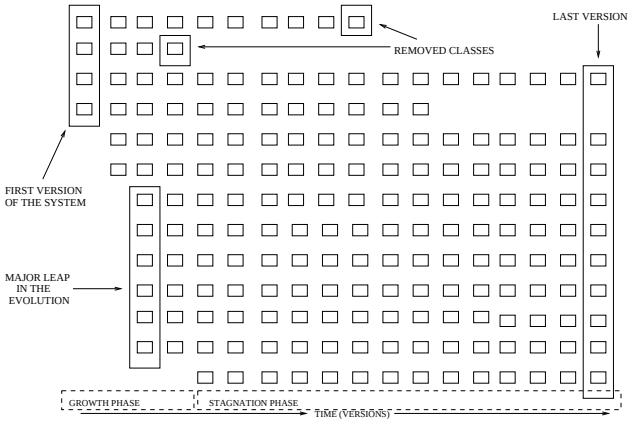

Figure 3: Some characteristics of the Evolution Matrix.

2.2.2 Characteristics at Class Level

We visualize each class using two different metrics. We have decided upon the number of methods and the number of variables. Since we visualize different versions of the same class, we can effectively see if the class grows, shrinks or stays the same from one version to another. In the figures in the paper we use colors to denote the changes from version to version: We use black for growing classes, light gray for shrinking classes and white for classes which stay the same.

2.3 A Categorization of Classes based on the Evolution Matrix

We present here a categorization of classes based on the evolution matrix, i.e., based on the visualization of different versions of a class. The categorization stems from the experiences we obtained while applying our approach on several case studies. A large part, but not all, of the vocabulary used here is taken out of the domain of astronomy. We do so because we have found that some of the names from this domain convey extremely well the described types of evolution. This vocabulary is of utmost importance because a complex context and situations, like the evolution of a class, can be communicated to another person in an efficient way. This idea comes from the domain of patterns [7].

During our case studies we have encountered several ways in which a class can evolve over its lifetime. We list here the most prominent types. Note that the categories introduced here are not mutually exclusive, i.e. a class can behave like a pulsar for a certain part of its life and then become a white dwarf for the rest of its life.

- **Pulsar.** A *pulsar* class grows and shrinks repeatedly during its lifetime, as we see in Figure 5. The growth phases are due to additions of functionality, while the shrinking phases are most probably due to refactorings and restructurings of the class. Note that a refactoring may also make a class grow, for example when a long method is broken down into many shorter methods. Pulsar classes can be seen as hotspots in the system: for every new version of the system changes on a pulsar class must be performed.

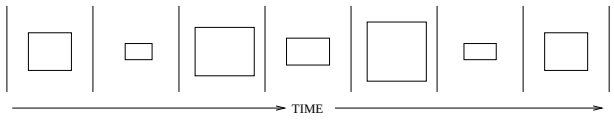

Figure 5: The Visualization of a Pulsar class.

- **Supernova.** A *supernova* is a class which suddenly explodes in size. The reasons for such an explosive growth may vary, although we have already made out some common cases:

  – Major refactorings of the system which have caused a massive shift of functionality towards a class.
  – Data holder classes which mainly define attributes whose values can be accessed. Due to the simple structure of such classes it is easy to make such a class grow rapidly.
  – So-called *sleeper* classes. A class which has been defined a long time ago but is waiting to be filled with functionality. Once the moment comes the developers may already be certain about the functionality to be introduced and do so in a short time.

  Supernova classes should be examined closer as their accelerated growth rate may be a sign of unclean design or introduce new bugs into the system.

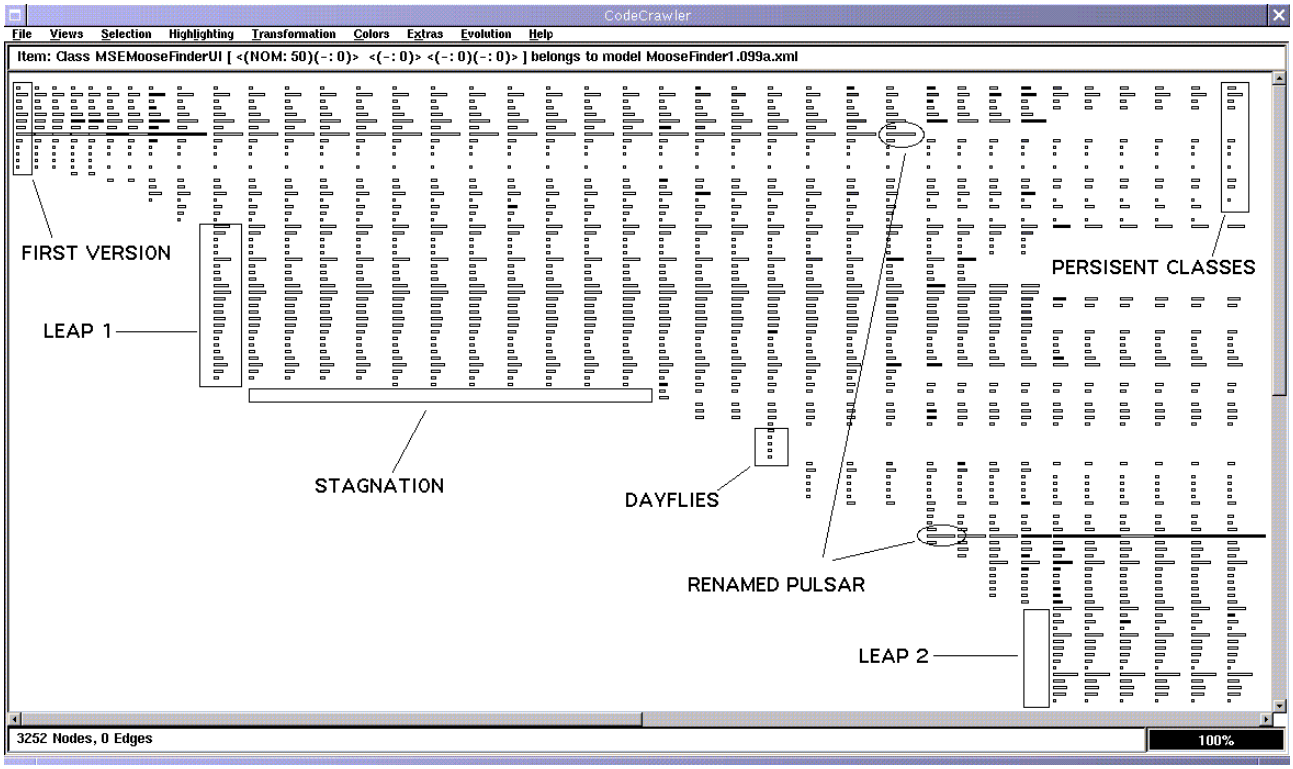

Figure 4: The Evolution Matrix of MooseFinder.

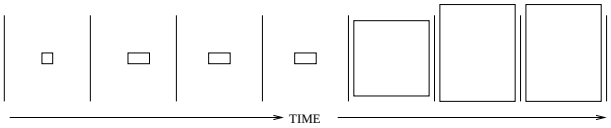

Figure 6: The Visualization of a Supernova class.

- **White Dwarf.** A *white dwarf* is a class who used to be of a certain size, but due to varying reasons lost the functionality it defined to other classes and now trundles along in the system without a real meaning. We can see a schematic display of a white dwarf class in Figure 7. White dwarf classes should be examined for signs of dead code, i.e. they may be obsolete and therefore be removed.

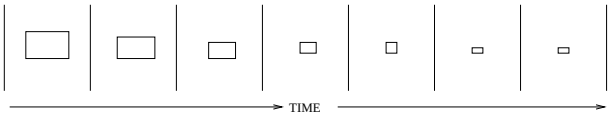

Figure 7: The Visualization of a White Dwarf class.

- **Red Giant.** A *red giant* class can be seen as a permanent god class [15], which over several versions keeps on being very large. God classes tend to implement too much functionality and are quite difficult to refactor, for example using a split class refactoring [6].
- **Stagnant.** A *stagnant* class is one which does not change over several versions of the software system it belongs to. We list here a few reasons which may lead to a stagnant class:
  - Dead code. The class may have become obsolete at a certain point in time, but was not removed for varying reasons.
  - Good design. Stagnant classes can have a good implementation or a simple structure which makes them resistant to changes affecting the system.
  - The class belongs to a subsystem on which no work is being performed.
- **Dayfly.** A *dayfly* class has a very short lifetime, i.e., it often exists only during one version of the system. Such classes may have been created to try out an idea which was then dropped.
- **Persistent.** A *persistent* class has the same lifespan as the whole system. It has been there from the beginning and is therefore part of the original design. Persistent classes should be examined, as they may represent cases of dead code that no developer dares to remove as there is no one being able to explain the purpose of that class.

3. CASE STUDIES

In this section we present some case studies whose evolution we have visualized using the evolution matrix view. We shortly introduce each case study, and then show and discuss their evolution matrix.

3.1 MooseFinder

MooseFinder [17] is an average sized application written in VisualWorks Smalltalk by one developer in little more than one year as part of a diploma. We have taken 38 versions of the software as

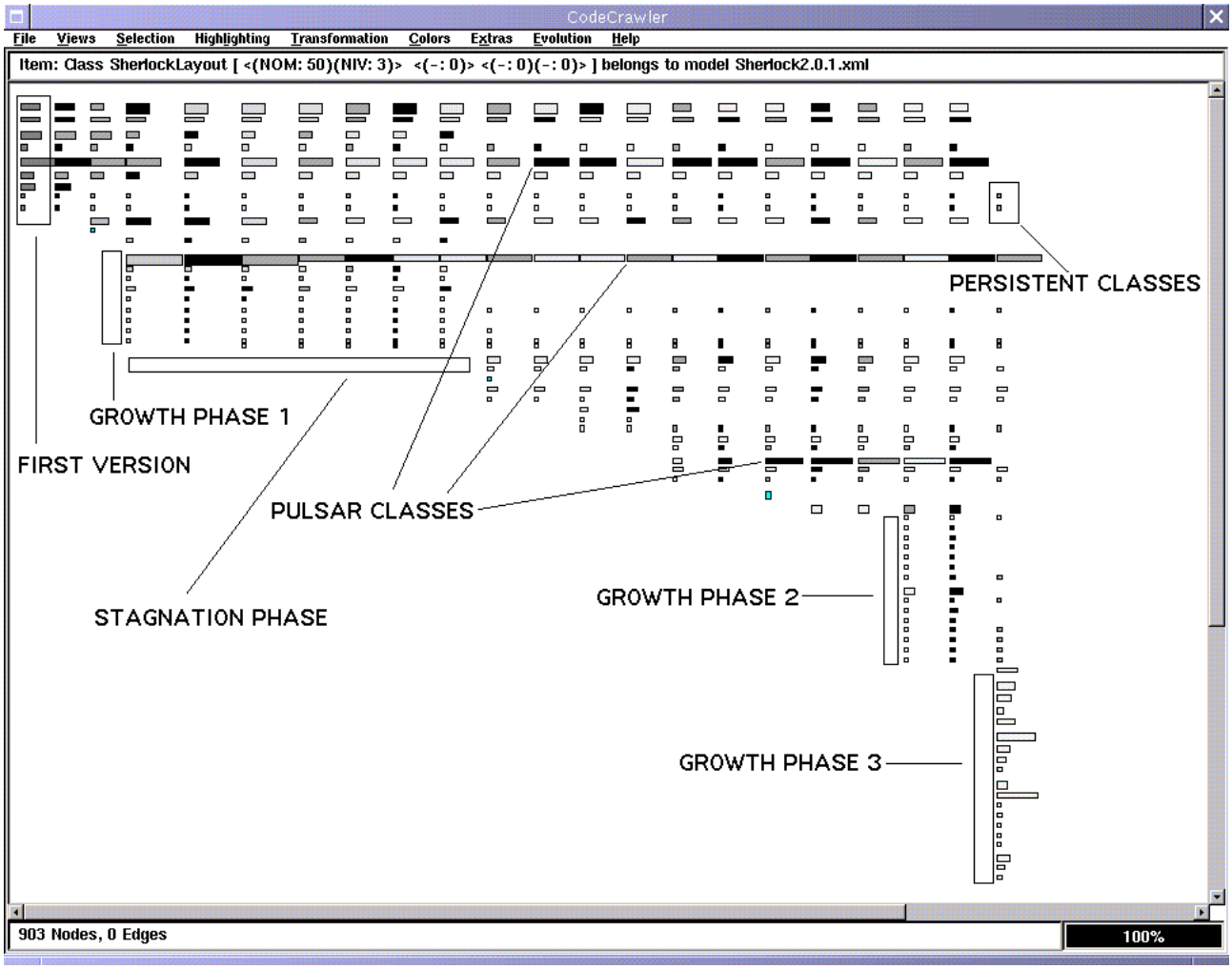

Figure 8: The Evolution Matrix of Sherlock.

a case study.

Discussion. In Figure 4 we can see the evolution matrix of MooseFinder. We see that the first version on the left has a small number of classes and that of those only few survived until the last version, i.e., are persistent classes. We can also see there have been two major leaps and one long phase of stagnation. Note that the second leap is in fact a case of massive class renaming: many classes have been removed in the previous version and appear as added classes in the next version. There is also a version with a few dayfly classes. The classes themselves rarely change in size except the class annotated as a renamed pulsar class, which at first sight seems to be one of the central classes in the system.

3.2 Supremo

Supremo [12] is also written in VisualWorks Smalltalk. We have taken 21 versions of this application as a case study.

Discussion. In Figure 8 we see the evolution matrix of Supremo. We can see that there is apart from a stagnation phase a constant growth of the system with three major growth phases. Note that the last growth phase is due to a massive renaming of classes. There are several pulsar classes which strike the eye, some of which have considerable size. We can also see that from the original classes only two are persistent, i.e. the whole system renewed itself nearly completely.

4. CODECRAWLER AND MOOSE

CodeCrawler is the tool used to generate the views presented in this paper. CodeCrawler supports reverse engineering through the combination of metrics and software visualization [13, 2, 4]. Its power and flexibility, based on simplicity and scalability, has been repeatedly proven in several large scale industrial case studies.

CodeCrawler is implemented on top of Moose. Moose is a language independent reengineering environment written in Smalltalk. It is based on the FAMIX metamodel [3], which provides for a language independent representation of object-oriented sources and contains the required information for the reengineering tasks performed by our tools. It is *language independent*, because we need to work with legacy systems written in different implementation languages. It is *extensible*, since we cannot know in advance all information that is needed in future tools, and since for some reengineering problems tools might need to work with language-specific information, we allow for language plug-ins that extend the model with language-specific features. Next to that, we allow tool plug-ins to extend as well the model with tool-specific information.

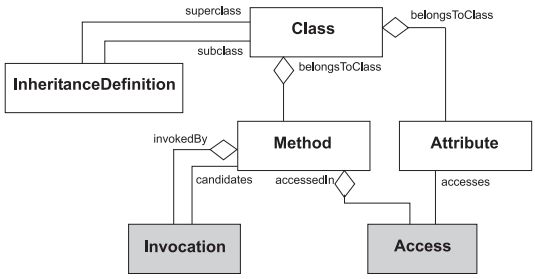

Figure 9: A simplified view of the FAMIX metamodel.

A simplified view of the FAMIX metamodel comprises the main object-oriented concepts - namely Class, Method, Attribute and Inheritance - plus the necessary associations between them - namely Invocation and Access (see Figure 9).

Moose can be used in the context of evolution as it allows several models to be loaded at the same time. If we load models of different versions of the same software we get a sequence of snapshots of the evolution of the software. In this paper we use this technique as a base for the evolution matrix visualization.

5. CONCLUSION

In this paper we have presented the evolution matrix, a novel way to visualize the evolution of classes in object oriented software systems. The evolution matrix can greatly reduce the amount of data one has to deal with when analyzing the evolution of software using a simple visualization approach. Based on the visualizations obtained we have introduced a categorization of classes based on their personal evolution. We have applied the evolution matrix on some case studies to verify the usefulness of this approach.

5.1 Limits of the Approach

The approach presented here is limited in the following ways:

- The effectiveness depends on the number of available versions of a software (the more the better), as well as the amount of changes between one version and the next: as our approach aims mostly at the differences between two versions, in the case of two versions which are too distant from each other in terms of changes many details get lost.

- A major aspect of classes in object-oriented programming is their capability to inherit from each other: a class seldom exists on its own, but is often embedded in the context of its inheritance hierarchy. This aspect goes lost with the current evolution matrix visualization.

- The approach is not immune to name changes. If a class has been renamed at a certain point in time, it will be treated as a class which has been removed and a new class which has been added.

- Software visualization techniques must deal with the issue of scalability. In the case of the evolution matrix the approach has worked for systems of nearly 100 classes. For larger systems we end up with a very large matrix where mainly general statements about the system's evolution can be made, rather than precise statements about particular classes.

5.2 Related Work

Among the various approaches to understand software evolution that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids.

Holt and Pak [10] present a visualization tool called GASE to elucidate the architectural changes between different versions of a system.

Rayside et al. [14] have built a tool called JPort for exploring evolution between successive versions of the JDK. Their intent was to provide a tool for detecting possible problem areas when developers wish to port their Java tools across versions of the JDK.

In [11, 16] Claudio Riva presents work which has similarities, i.e. he also visualizes several versions of software (at subsystem level) using colors. Through the obtained colored displays they can make conclusions about the evolution of a system. Their approach differs as they do not have actual software artifacts but only information about software releases. This implies that they cannot verify the correctness of their informations. Our approach allows us to enrich the display using metrics information as well as being able to access every version of the software artifacts.

Burd and Munro have been analyzing the calling structure of source code [1]. They transformed calling structures into a graph using dominance relations to indicate call dependencies between functions. Dominance trees were derived from call-directed-acyclic-graphs [1]. The dominance trees show the complexity of the relationships between functions and potential ripple effects through change propagation.

Gall and Jazayeri examined the structure of a large telecommunication switching system with a size of about 10 MLOC over several releases [9]. The analysis was based on information stored in a database of product releases, the underlying code was neither available nor considered. They investigated first in measuring the size of components, their growth and change rates. The aim was to find conspicuous changes in the gathered size metrics and to identify candidate subsystems for restructuring and reengineering. A second effort on the same system focused on identifying logical coupling among subsystems in a way that potential structural shortcomings could be identified and examined [8].

Most publications and tools that tackle the problem of software evolution using software visualization work at higher abstraction levels, i.e. systems, subsystems, etc. We provide a visualization of classes as well as a categorization of classes based on that.

5.3 Future Work

In the future we plan to apply the evolution matrix approach on large industrial case studies to evaluate its usefulness and scalability. One of the major problems which we foresee is the availability of several versions of an industrial system.

We also plan to extend and enrich the evolution matrix to increase its usability. At this time the classes are treated as standalone objects. We think the introduction of relationships between classes, especially inheritance, will increase its usefulness.

The use of other metrics remains also to be explored. Preliminary uses of difference metrics have yielded interesting results, and we plan to further explore this direction.

To tackle the problem of scalability we will introduce grouping techniques to reduce the number of displayed entities. We also plan to examine very large systems by visualizing higher level constructs like subsystems and applications instead of classes.

Acknowledgments. We would like to thank Stéphane Ducasse for comments on drafts of this paper.

6. REFERENCES

[1] E. Burd and M. Munro. An initial approach towards measuring and characterizing software evolution. In *Proceedings of WCRE'99*, pages 168–174. IEEE Computer Society, 1999.

[2] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

[3] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne, Aug. 1999.

[4] S. Ducasse and M. Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et Science Informatique*, 2001. To appear in Techniques et Sciences Informatiques, Edition Speciale Reutilisation.

[5] 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.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

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

[9] R. K. H. Gall, M. Jazayeri and G. Yrausmuth. Software evolution observations based on product release history. In *ICSM'97 Proceedings (International Conference on Software Maintenance)*, pages 160–166. IEEE Computer Society, 1997.

[10] R. Holt. Gase: visualizing software evolution-in-the-large. In *Proceedings of WCRE'96*, pages 163–167. IEEE Computer Society, 1996.

[11] M. Jazayeri, H. Gall, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM'99 Proceedings (International Conference on Software Maintenance)*. IEEE Computer Society, 1999.

[12] G. G. Koni-N'Sapu. Supremo - a scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Bern, June 2001.

[13] M. Lanza. Combining metrics and graphs for object oriented reverse engineering. Diploma thesis, University of Bern, Oct. 1999.

[14] D. Rayside, S. Kerr, and K. Kontogiannis. Change and adaptive maintenance detection in java software systems. In *Proceedings of WCRE'98*, pages 10–19. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.

[15] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[16] C. Riva. Visualizing software release histories: The use of color and third dimension. Master's thesis, Politecnico di Milano, Milan, 1998.

[17] L. Steiger. Recovering the evolution of object oriented software systems using a flexible query engine. Diploma thesis, University of Bern, June 2001.