Understanding software evolution using a combination of software visualization and software metrics

Michele Lanza — Stéphane Ducasse

Software Composition Group
University of Berne, Switzerland
lanza@iam.unibe.ch, ducasse@iam.unibe.ch

ABSTRACT. *Coping with huge amounts of data is one of the major problems in the context of software evolution. Current approaches reduce this complexity by filtering out irrelevant information. In this paper we propose an approach based on a combination of software visualization and software metrics, as software visualization is apt for complexity reduction and metrics introduce the possibility to qualify evolution. 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. In addition we define a vocabulary that qualifies some specific situations that occurs when considering system evolution.*

RÉSUMÉ. *Analyser un très grand volume de données est un des problèmes majeurs lors de la compréhension de l'évolution de logiciels. Les approches existantes réduisent cette complexité en filtrant les informations non pertinentes. Dans cet article nous proposons une approche basée sur la combinaison de métriques et de visualisation, la visualisation permettant une réduction d'information et les métriques permettant une qualification de l'évolution. Ainsi nous présentons une* matrice d'évolution *: une visualisation simple et efficace qui aide à comprendre l'évolution des applications orientées objets. En plus, nous définissons un vocabulaire permettant de qualifier les situations caractéristiques rencontrées.*

KEYWORDS: *Evolution, Software Visualization, Software Metrics, Patterns, Reverse Engineering.*

MOTS-CLÉS : *évolution, visualisation de programmes, métriques du logiciel, patterns, rétroconception.*

1. Introduction

Coping with huge amounts of data is one of the major problems of software evolution research, as several versions of the same software must be analyzed in parallel. A technique which can be used to reduce this complexity is *software visualization*, as a good visual display allows the human brain to study multiple aspects of complex problems in parallel. Another useful approach when dealing with large amounts of data are *software metrics*. Metrics can help to assess the complexity of software and to discover software artifacts with unusual measurements. In this paper we present a visual technique called *evolution matrix* [LAN 01] that combines software visualization and software metrics. It allows for a quick understanding of the evolution of classes within software systems. Moreover the evolution matrix acts as a revealer of certain specific situations that occur during system evolution such as pulsating classes that grow and shrink during the lifetime of the system. We define a simple vocabulary to describe such specific behaviors. The intention is to build a vocabulary for software evolution. Note that even if the results we present are obtained on software systems written in Smalltalk, the approach presented here does not depend on a particular programming language, as our underlying metamodel is language-independent [DUC 00, DEM 01]. The paper is structured as follows: in the next section we present our visualization technique and, based on that, a categorization of classes. Afterwards we apply and discuss our approach on some case studies. We then discuss shortly CodeCrawler and Moose, the tools written by us. We conclude the paper by discussing the benefits and limits of our approach, as well as related work. Finally, we give an outlook on our future work in this area.

2. Combining metrics and software visualization

In this section we present our visualization technique called *evolution matrix* first presented in [LAN 01]. We want to stress that we put a special emphasis on lightweight techniques, in both fields of metrics and software visualization. We discuss the technique and then show an example matrix. At the end of this section we introduce a categorization of classes based on their visualization within the evolution matrix.

2.1. *A visualization technique for software evolution*

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. Two classes in two different versions are considered the same if they have the same name. Within the columns the classes are sorted alphabetically in case they appear for the first time in the system. Otherwise they are placed at the same vertical position as their predecessors. This order is important because it allows one to represent the continuous flow of development of existing classes and stresses the development of new ones. Figure 1 presents a schematic evolution matrix where the rows 5 and 6 represent new classes added in the system after the first release.

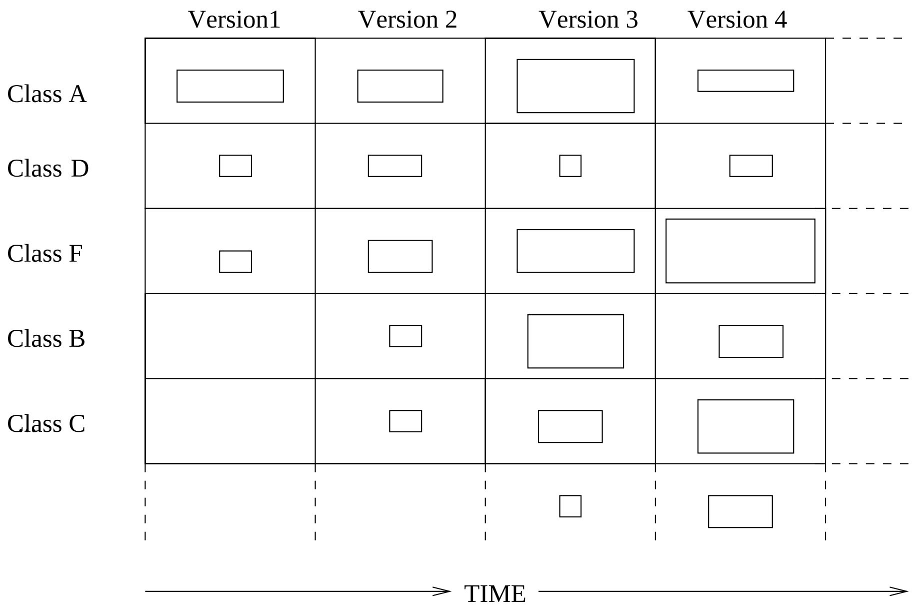


Figure 1. *A schematic display of the Evolution Matrix. Classes A,D and F are alphabetically ordered and stay since version 1. Classes B and C appeared after version 2*

The evolution matrix allows us to make statements on the evolution of an object oriented system at the system level. However, as the granularity at system level is too coarse, the evolution matrix is enhanced with additional information using metrics as shown in Section 3.

2.1.1. *Characteristics at system level*

As we see schematically in Figure 2 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 added at the bottom of the column of that version.

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

– **Growth and stabilization 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 stabilization phase (no classes are being added) the height of the matrix will stay the same. When a certain number of new classes are added they create a *leap phase.*

Besides characterizing the evolution at system level, the evolution matrix provides some information about the classes themselves. Two specific situations are worth being mentioned:

– **Dayfly classes**

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 classes**

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.

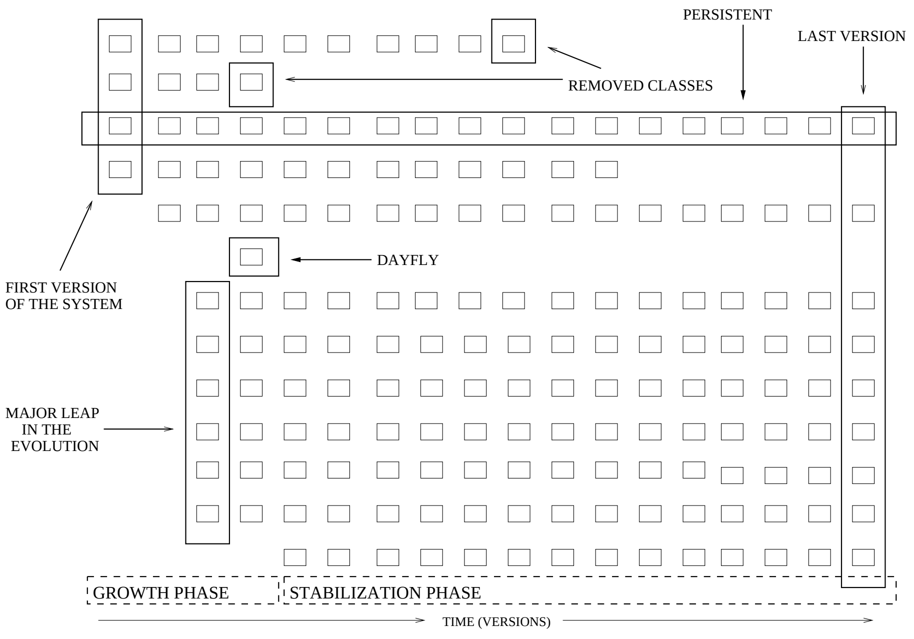


Figure 2. *System level evolution aspects using the Evolution Matrix*

Remarks. The system level view provided by the evolution matrix is not precise enough. Hence, a stabilization only describes the fact that classes stayed over multiple versions of the system. Nothing is said about the quality of the changes if any occurred. Such information is crucial for understanding a system, that's why the evolution matrix is enhanced using software metrics, as we see in Section 3.

3. Qualifying evolution: the metrics in play

While the evolution matrix shape provides some information regarding the system evolution, the granularity is too coarse. We now present how software metrics can improve the quality of the presented information. We distinguish two kinds of metrics: absolute ones, i.e., reflecting the values of the entity they measure and differential ones, i.e., measuring the difference of the values between two subsequent versions.

3.1. *Visualizing classes using metrics*

As we have previously seen we use two-dimensional boxes to represent classes. We can enrich this representation using metrics: the width and height of the boxes reflect metric measurements of the classes, as we see in Figure 3. This approach has been presented in [LAN 99] and [DEM 99a].

In the visualization presented in this paper we visualize classes and use the metrics *number of methods (NOM)* for the width and *number of attributes (NOA)* for the height, although in our tool we can choose other metrics. In the case of the differential metrics we just subtract the metric measurements of the predecessor of a class from its own measurements, i.e., if class A has 20 methods in the first version and 30 in the second, its differential metric value will be 30-20=10.

We chose deliberately to focus on metrics which can be easily extracted from the source code, as our intention is not to investigate new kinds of metrics.

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.

3.2. *A categorization of classes based on the Evolution Matrix*

We present here a categorization of classes 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

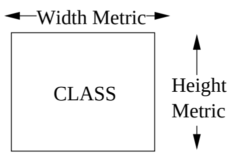


Figure 3. *A graphical representation of classes using metrics*

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 situation, like the evolution of software, can be communicated to another person in an efficient way. This idea comes from the domain of patterns [GAM 95]. 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 4. 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 hot places in the system: for every new version of the system changes on a pulsar class must be performed.

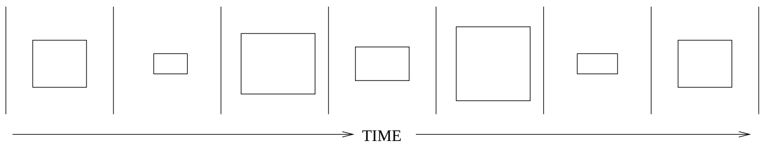


Figure 4. *The visualization of a Pulsar class. Note that the shape may change depending on the metrics associated with the representation*

– **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.

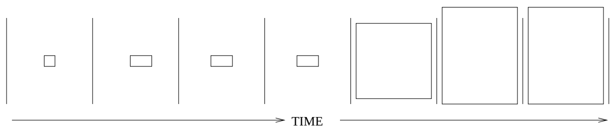


Figure 5. *The visualization of a Supernova class*

– 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. We can see a schematic display of a white dwarf class in Figure 6. White dwarf classes should be examined for signs of dead code, i.e., they may be obsolete and therefore be removed. Other possibilities could include a redistribution of responsibilities.

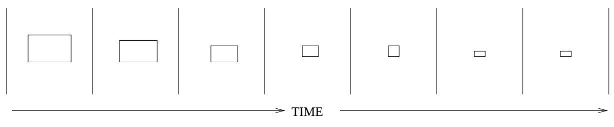


Figure 6. *The visualization of a White Dwarf class*

– Red Giant

A *red giant* class can be seen as a permanent god class [RIE 96], 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 [FOW 99].

– Idle

An *idle* 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 an idle class:

- Dead code. The class may have become obsolete at a certain point in time, but was not removed for varying reasons.

- Good design. Idle classes can have a good implementation or a simple structure which makes them resistant to changes affecting the system.

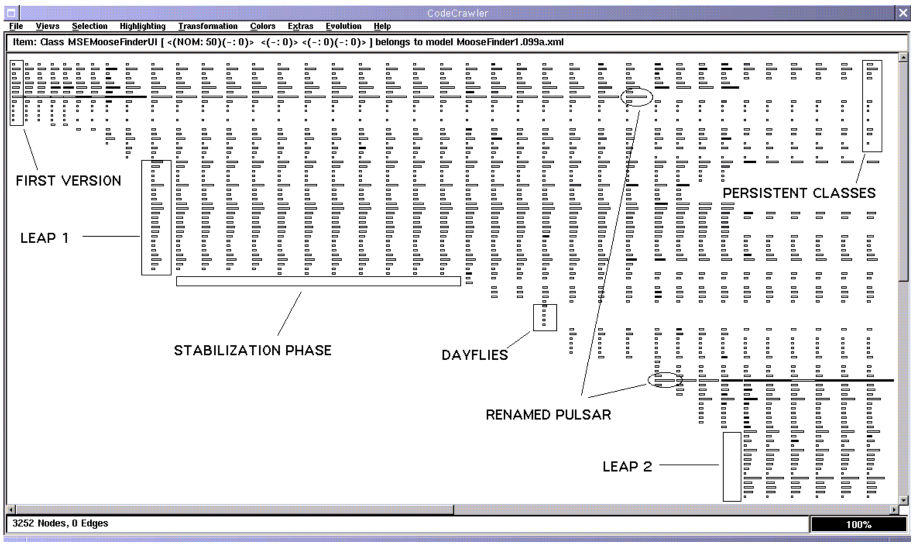
- The class belongs to a subsystem on which no work is being performed.

4. Illustration of the approach

In this section we present two case studies whose evolution we have visualized using the approach described above. We shortly introduce each case study, and then show and discuss them.

4.1. *MooseFinder*

MooseFinder [STE 01] is a small to medium sized application written in Visual-Works Smalltalk by one developer in little more than one year as part of a diploma thesis. We have taken 38 versions of the software as a case study.



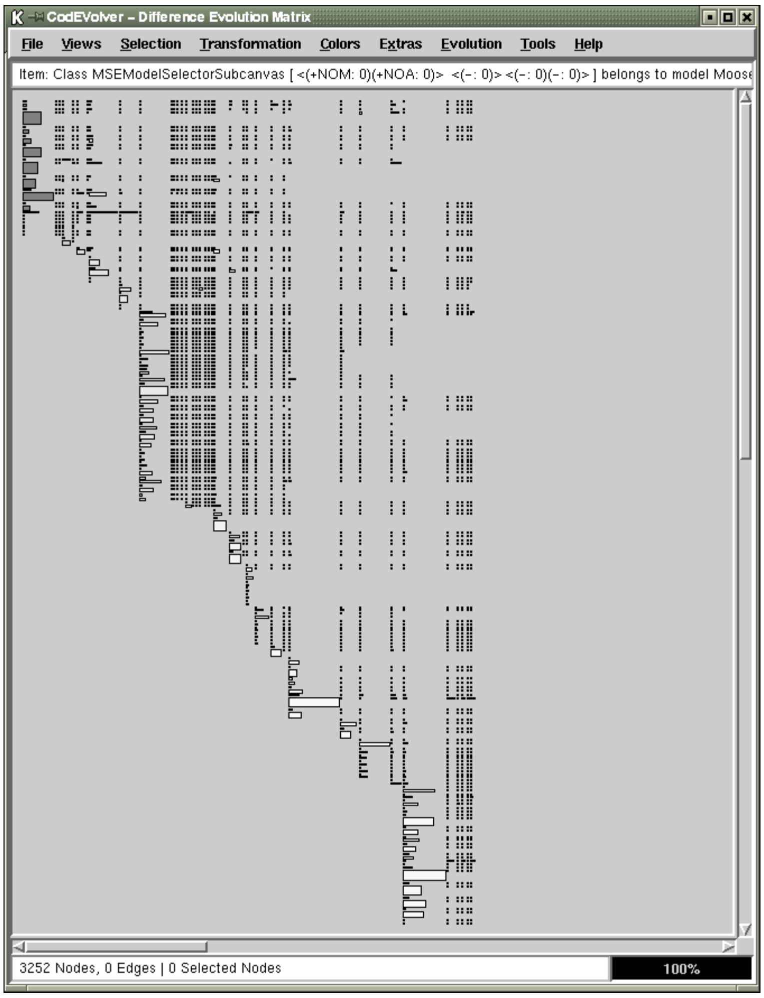
Item: Class MSEMooseFinderUI [<(NOM: 50)(-: 0)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model MooseFinder1.099a.xml

3252 Nodes, 0 Edges

Figure 7. *The Evolution Matrix of MooseFinder*

Discussion. In Figure 7 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 stabilization. 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.

Figure 8 presents the same system where the difference between metrics are represented. It reveals even more the sudden increases in size of certain classes. The



Item: Class MSEModelSelectorSubcanvas [<(+NOM: 0)(+NOA: 0)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model Moose

3252 Nodes, 0 Edges | 0 Selected Nodes

Figure 8. *The Difference Evolution Matrix of MooseFinder*

interesting property of this view is that emphasizes changes. For example, having two flat boxes following each others shows that the class grows over the two versions. Due to the graphical screen constraints, we nearly cannot see on the picture but this view allows also to see where attributes have been added.

For displaying boxes in the other view we have to have a default size for the nodes else as soon as one metrics values is zero we would end up to lose the other one. Such a choice has as consequence that metric with small values like the number of attributes added cannot easily be identifiable. The differential view solves this problem.

4.2. *Supremo*

Supremo [KON 01] is also written in VisualWorks Smalltalk. We have taken 21 versions of this application as a case study.

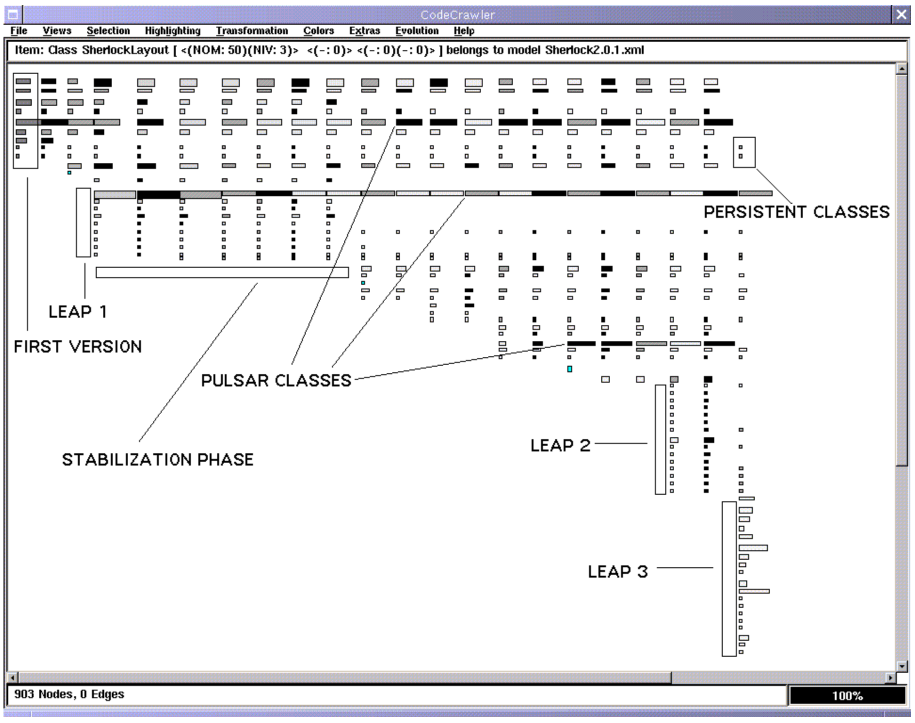
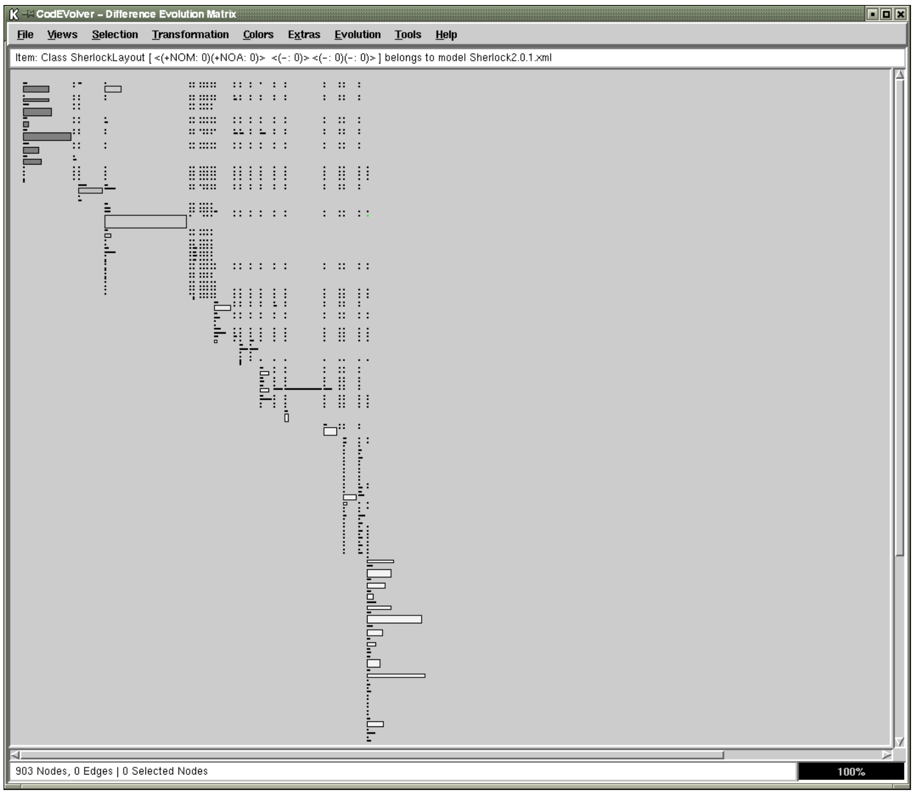

CodeCrawler
File Views Selection Highlighting Transformation Colors Extras Evolution Help
Item: Class SherlockLayout [<(NOM: 50)(NIV: 3)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model Sherlock2.0.1.xml
PERSISTENT CLASSES
LEAP 1
FIRST VERSION
PULSAR CLASSES
STABILIZATION PHASE
LEAP 2
LEAP 3
903 Nodes, 0 Edges
100%

Figure 9. *The Evolution Matrix of Supremo*

Discussion. In Figure 9 we see the evolution matrix of Supremo. We can see that there is apart from a stabilization 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. In Figure 10 presents the same system using the differential view which emphasizes the changes made.

5. Implementation: 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 soft-


CodEVolver – Difference Evolution Matrix
File Views Selection Transformation Colors Extras Evolution Tools Help
100%

Item: Class SherlockLayout [<(+NOM: 0)(+NOA: 0)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model Sherlock2.0.1.xml

903 Nodes, 0 Edges | 0 Selected Nodes

Figure 10. *The Difference Evolution Matrix of Supremo*

ware visualization [LAN 99, DEM 99a, DUC 01]. 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 [DEM 01], 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.

A simplified view of the FAMIX metamodel comprises the main object-oriented concepts - namely class, method, attribute and inheritance - plus the necessary associ-

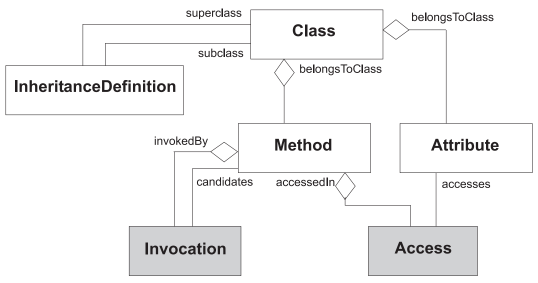


Figure 11. *A simplified view of the FAMIX metamodel*

ations between them - namely method invocation and attribute access (see Figure 11). We did not use the UML metamodel, as it is specifically targeted towards object-oriented analysis and design and not at representing source code as such. *In stricto sensu* we can say that UML is not sufficient for modelling source code for the purpose of reengineering [DEM 99b]. For an in-depth discussion of this question see [TIC 01]. Moose 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.

5.1. *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 [HOL 96] present a visualization tool called GASE to elucidate the architectural changes between different versions of a system.

Rayside et al. [RAY 98] 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. They provide evolution analysis at the level of Reuse Contracts [STE 96].

In [JAZ 99, RIV 98] Claudio Riva presents work which has similarities with ours, 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 [BUR 99]. 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 [BUR 99]. 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 [GAL 97]. 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 [GAL 98].

Sahraroui et al. [SAH 00, LOU 98] present another aspect of the research on software evolution which is the prediction of the evolution. Our current focus is to understand the evolution even if our long term goal is to gain a better prediction on which parts of the system will cause problems.

6. Conclusion and future work

We presented a lightweight approach for helping the understanding of system evolution which is based on the definition of a graphical matrix displaying classes that are enriched with metrics information. The approach has the following properties:

– it reduces complexity and provides system wide views that help to understand essential changes during the evolution of an application.

– it provides a finer understanding of the evolution of classes.

– it builds a vocabulary to describe system and class evolution.

– it scales well. However, we have some screen limitation problems with huge systems. Working at another level of abstraction will be required.

The presented approach has some limitations: it is fragile regarding the renaming of the classes. Right now we consider a class similar to the subsequent versions if it has the same name. This assumption is too limitating and we plan to remove it by applying some simple heuristics to identify renamed classes such as a percentage of common methods and attributes.

We would like to apply the evolution matrix at another levels of granularity. In particular, we want to be able to reason in terms of subsystems, packages or applications because these concepts represent conceptually linked classes in large applications. In such a context we would like to understand the evolution of subsystems, inside them and between them when for example a class has been moved from one subsytem to another.

Applying other metrics such the number of lines of code in combination with the number of methods or statements in the class should be investigated to see if we can qualify the actual changes, i.e., new methods can be added as the results of code refactoring while at the same time the number of lines can decrease. Other metrics, like the number of subclasses, the hierarchy nesting level of classes or the number of inherited methods, may help to detect hierarchy refactorings.

The choice of the case studies is also another factor that we would like to analyze. Indeed, the rates of change may be quite different with longer periods between releases. In our experiences we have access to all the versions made by the developers and could not really assess major versions. In the future we plan to apply the same approach to several versions of large systems like Squeak, Java Swing, VisualWorks Smalltalk and the Microsoft Foundation Classes (MFC) where the time spent between two versions can be months or years.

7. References

[BUR 99] BURD E., MUNRO" M., "An Initial Approach towards Measuring and Characterizing Software Evolution", *Proceedings of the Working Conference on Reverse Engineering, WCRE'99*, 1999, p. 168-174.

[DEM 99a] DEMEYER S., DUCASSE S., LANZA M., "A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization", BALMAS F., BLAHA M., RUGABER S., Eds., *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*, IEEE, Oct. 1999.

[DEM 99b] DEMEYER S., DUCASSE S., TICHELAAR S., "Why Unified is not Universal. UML Shortcomings for Coping with Round-trip Engineering", RUMPE B., Ed., *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, LNCS 1723, Kaiserslautern, Germany, Oct. 1999, Springer-Verlag.

[DEM 01] DEMEYER S., TICHELAAR S., DUCASSE S., "FAMIX 2.1 - The FAMOOS Information Exchange Model", report , 2001, University of Berne, to appear.

[DUC 00] DUCASSE S., LANZA M., TICHELAAR S., "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems", *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[DUC 01] DUCASSE S., LANZA M., "Towards a Methodology for the Understanding of Object-Oriented Systems", *Technique et science informatiques*, vol. 20, num. 4, 2001, p. 539-566.

[FOW 99] FOWLER M., BECK K., BRANT J., OPDYKE W., ROBERTS D., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[GAL 97] GALL H., JAZAYERI M., KLÖSCH R. R., TRAUSMUTH G., "Software Evolution Observations Based on Product Release History", *Proceedings of the International Conference on Software Maintenance 1997 (ICSM'97)*, 1997, p. 160-166.

[GAL 98] GALL H., HAJEK K., JAZAYERI M., “Detection of Logical Coupling Based on Product Release History”, *Proceedings of the International Conference on Software Maintenance 1998 (ICSM’98)*, 1998, p. 190-198.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns,* Addison Wesley, Reading, MA, 1995.

[HOL 96] HOLT R. C., PAK J., “GASE: Visualizing Software Evolution-in-the-Large”, *Proceedings of WCRE’96*, 1996, p. 163-167.

[JAZ 99] JAZAYERI M., GALL H., RIVA C., “Visualizing Software Release Histories: The Use of Color and Third Dimension”, *ICSM’99 Proceedings (International Conference on Software Maintenance)*, IEEE Computer Society, 1999.

[KON 01] KONI-N’SAPU G. G., “A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems”, Diploma thesis, University of Berne, June 2001.

[LAN 99] LANZA M., “Combining Metrics and Graphs for Object Oriented Reverse Engineering”, Diploma thesis, University of Bern, Oct. 1999.

[LAN 01] LANZA M., “The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques”, *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, 2001, Page to be published.

[LOU 98] LOUNIS H., SAHRAOUI H. A., MELO W. L., “Vers un modèle de prédiction de la qualité du logiciel pour les systèmes à objets”, *L’Objet, Numéro spécial Métrologie et Objets*, vol. 4, num. 4, 1998.

[RAY 98] RAYSIDE D., KERR S., KONTOGIANNIS K., “Change and Adaptive Maintenance Detection in Java Software Systems”, *Proceedings of WCRE’98*, IEEE Computer Society, 1998, p. 10–19, ISBN: 0-8186-89-67-6.

[RIE 96] RIEL A. J., *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

[RIV 98] RIVA C., “Visualizing Software Release Histories: The Use of Color and Third Dimension”, Master’s thesis, Politecnico di Milano, Milan, 1998.

[SAH 00] SAHRAOUI H. A., BOUKADOUM M., LOUNIS H., ETHÈVE F., “Predicting Class Libraries Interface Evolution: an investigation into machine learning approaches”, *Proceedings of 7th Asia-Pacific Software Engineering Conference*, 2000.

[STE 96] STEYAERT P., LUCAS C., MENS K., D’HONDT T., “Reuse Contracts: Managing the Evolution of Reusable Assets”, *Proceedings of OOPSLA ’96 Conference,* ACM Press, 1996, p. 268-285.

[STE 01] STEIGER L., “Recovering the Evolution of Object Oriented Software Systems Using a Flexible Query Engine”, Diploma thesis, University of Bern, June 2001.

[TIC 01] TICHELAAR S., “Modeling Object-Oriented Software for Reverse Engineering and Refactoring”, PhD thesis, University of Bern, 2001.