Visualizing and Characterizing the Evolution of Class Hierarchies

Tudor Gîrba and Michele Lanza
Software Composition Group
University of Berne
Switzerland
{girba, lanza}@iam.unibe.ch

Abstract

Analyzing historical information can show how a software system evolved into its current state, but it can also show which parts of the system are more evolution prone. Yet, historical analysis implies processing a vast amount of information which makes the interpretation difficult. To address this issue, we introduce the notion of history of source code artifacts as a first class entity and define measurements which summarize the evolution of such entities. We then use these measurements to define polymetric views for visualizing the effect of time on class hierarchies[1]. We show the application of our approach on one large open source case study and reveal how we classify the class hierarchies based on their history.

Keywords: reverse engineering, software evolution, historical measurements, software visualization, polymetric views

1 Introduction

Analyzing historical information is difficult due to the vast amount of information that needs to be processed, transformed, and understood. Suppose we had as a case study 40 versions of a software system's code, each version consisting on average of ca. 400 classes. We would have to analyze 40 times more data (ca. 16000 classes, *i.e.,* class versions) which would make the analysis much more difficult. Still, the code history of a system holds useful information that can be used to reverse engineer the most recent version of the system and to get an overall picture of its evolution. Nevertheless, it requires one to create higher level views of the data.

In this paper we concentrate on describing and understanding the evolution of class hierarchies. We introduce the notion of history as a first class entity and define measurements which summarize the evolution of an entity or a set of entities.

We use these measurements to define the *Class Hierarchy History Complexity View*, a polymetric view [8] [10] for visualizing the effect of time on class hierarchies. Based on the *Class Hierarchy History Complexity View* we define four characteristics of the evolution of class hierarchies: the age of the hierarchy, the inheritance relationship stability, the class size stability, and the development effort concentration balance. Based on these characteristics we define a vocabulary to describe patterns of evolution of class hierarchies.

We start by introducing the notion of a history and then define history measurements. In Section 3 we introduce the visualization based on the defined history measurements, and we introduce a vocabulary we use to describe patterns of class hierarchies evolution. In Section 5 we apply our approach on a case study called Jun, an open source software system, and discuss the results. Prior to concluding we present the related work.

2 History Measurements

To introduce the visualization, we first need to introduce our approach to measuring the evolution of entities. Thus, we define a *history* to be a sequence of versions of the

[1]The visualizations in this paper make use of colors, please obtain a color-printed or electronic version for better understanding.

same kind of entity (*e.g.,* class history, system history, etc.). By a version we understand a snapshot of an entity at a certain point in time (*e.g.,* class version, system version, etc.). Having history as a first class entity we define historical measurements which we later use to vizualize the evolution of class hierarchy.

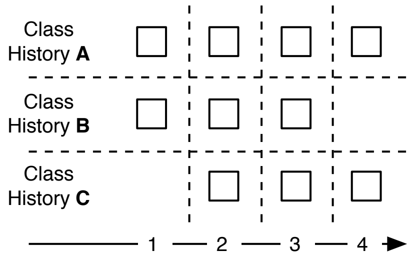


Figure 1: Example of a system history displayed with a simplified Evolution Matrix view.

Example. In Figure 1 we use a simplified example of the Evolution Matrix view [9] to display a system history with four versions. A cell in the matrix is marked by a square and represents a class version. A row in the matrix represents a class history and a column represents a system version. We see that class A was present in all four versions of the system, class B was removed in the last system version, while class C appeared in the system only after the first system version.

2.1 Evolution of a Version Property (*E*)

We define a generic measurement, called evolution of a version property P (*E(P, i)*), as being the absolute difference of that property between version $i-1$ and i:

$$(i > 1) \quad E_i(P,C) = |P_i(C) - P_{i-1}(C)| \tag{1}$$

E(P, 1..n) is the sum of the absolute difference of property P in subsequent versions from version 1 (*i.e.,* the first version) to version n (*i.e.,* the latest version) of a history H:

$$(n > 1) \quad E_{1..n}(P,C) = \sum_{i=2}^{n} E_i(P,C) \tag{2}$$

We instantiate this measurement by applying it on different version properties of classes like: *NOM* (the number of methods) or *NOS* (the number of statements). Thus we have two class history measurements: Evolution of Number of Methods (*ENOM*) and Evolution of Number of Statements (*ENOS*).

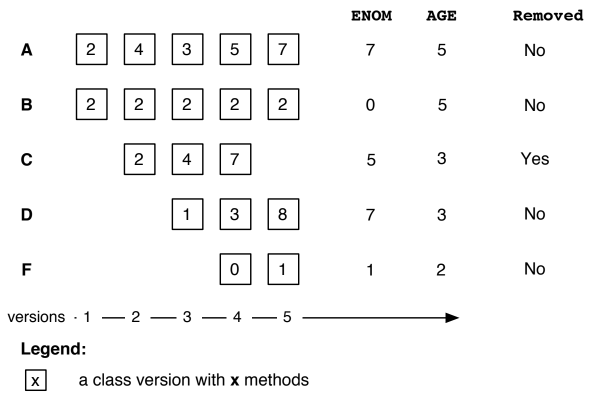
$$ENOM_{1..n}(C) = E_{1..n}(NOM, C) \tag{3}$$

$$ENOS_{1..n}(C) = E_{1..n}(NOS, C) \tag{4}$$

2.2 The age of a history and removed histories

Age of a history. We define the *Age* as being the number of versions of a history.

Removed histories. We say a history has been removed if its last version is not part of the last version of the system, *i.e.,* if it did not survive until the most recent version of the system.



	1	2	3	4	5	ENOM	AGE	Removed
A	2	4	3	5	7	7	5	No
B	2	2	2	2	2	0	5	No
C		2	4	7		5	3	Yes
D			1	3	8	7	3	No
F				0	1	1	2	No

versions · 1 — 2 — 3 — 4 — 5
Legend:
x a class version with **x** methods

Figure 2: An example of evolutionary measurements.

Example. In Figure 2 we display an Evolution Matrix of five system versions. Each cell in the matrix is a class version and the number inside the cell represents the number of methods in that particular version. We can see that:

- Class B was in the system from the very beginning to the very end, but no methods were detected as being added or removed during its history.
- Class A was also present in all the versions, but as opposite to class D, many more methods were added or removed during its history.
- Class A was in the system almost twice as many versions as class D, but in both class histories there were equal amounts of methods added or removed in subsequent versions.
- Class C has been removed from the system in its second last version.

3 Principles of a Polymetric View

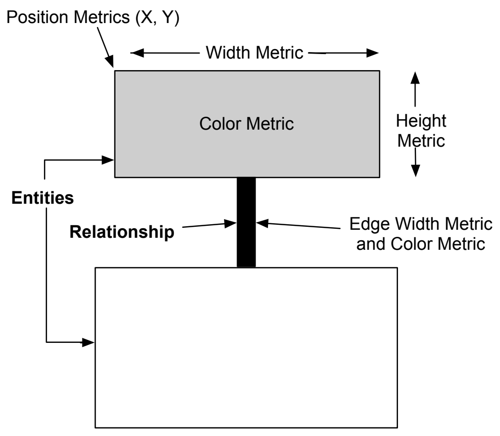


Figure 3: The principles of a polymetric view.

We use visualizations to understand the details of the evolution of class hierarchies. The visualization we propose is based on the polymetric views described by Lanza [8] [10]. In Figure 3 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:

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).

Edge width. The width of an edge can render a measurement: the wider the edge, the higher the measurement it is rendering.

Edge color. 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.

4 Class Hierarchy History Complexity View

In this paper we present a visualization called *Class Hierarchy History Complexity View*, whose specs are described in Table 1. It uses a simple tree layout to seemingly display classes and inheritance relationships. However, what it actually visualizes are the *histories* of classes and inheritance relationships.

Nodes and edges which have been removed while the system was evolving (*i.e.,* they are not present anymore) have a cyan color[2]. The color of the class history nodes and the width of the inheritance edges represents their age: the darker the nodes and the wider the edges, the more *grounded* in time they are, *i.e.,* the longer they have been present in the system. Thus, lightly colored nodes

[2]In a gray-scale print of the paper cyan will look like light gray.

***Class Hierarchy History Complexity View* Description**	
Layout	Tree
Nodes	Class histories
Edges	Inheritance histories
Scope	Full system history
Metric Scale	Linear
Node Width	*ENOM* of the class history
Node Height	*ENOS*/5 of the class history
Node Color	*Age*, Cyan = Removed
Edge Width	*Age* of the inheritance history
Edge Color	Cyan = Removed, Black = Present
Figure	Figure 4

Table 1: Specs of the *Class Hierarchy History Complexity View*.

and thin edges represent *younger* classes and inheritance relationships.

The width of the class history node is given by the *ENOM* while the height is given by the tenth part of the *ENOS* (*i.e., ENOS*/5). Thus, the wider a node is, the more methods were added or removed in subsequent versions in that class history; the greater the height of a node is, the more statements were added or removed in subsequent versions in that class history. We chose to use to divide *ENOS* by 5 because in the case study we analyzed, a method has on average around 5 statements. Therefore, a node would typically appear square in the view.

Based on the visualization we characterize the evolution of class hierarchies. We define a vocabulary based on four characteristics and different labels:

1. The age of the hierarchy:
    - *Newborn*. A newborn hierarchy is a freshly introduced hierarchy. The nodes in such a hierarchy will be colored in white.
    - *Young*. A young hierarchy is colored in light colors.
    - *Old*. As opposed to the young hierarchies, the old ones are colored in dark colors.
    - *Persistent*. We say a hierarchy is persistent if all the classes were present in all system versions. In a persistent hierarchy, the nodes will be black.
2. The inheritance relationship stability:
    - *Reliable*. We define a hierarchy as being reliable when the inheritance relationships between classes are stable and old. Thus, the edges of such a hierarchy will appear black and thick.
    - *Fragile*. A hierarchy is fragile when there are a lot of inheritance relationships which disappear. Such a hierarchy will appear as having a lot of cyan colored edges.
3. The size stability:
    - *Stable*. In a stable hierarchy the nodes are small.
    - *Unstable*. In an unstable hierarchy many methods are being added and removed during its evolution: the hierarchy contains large nodes.
4. The development effort concentration balance:
    - *Balanced*. In a balanced hierarchy, the effort is evenly spent among its classes. The hierarchy nodes will appear as being of about the same size.
    - *Unbalanced*. An unbalanced hierarchy is one in which the development effort is not equally distributed on the classes. In such a hierarchy, there will be some nodes which are bigger than the rest.

5 Classifying Class Hierarchy Histories of Jun

As case study we selected 40 versions of Jun[3]. Jun is a 3D-graphics framework written in Smalltalk. The project lasted for more than seven years and is still under development. As experimental data we took every 5th version starting from version 5 (the first public version) to version 200. The time distance between version 5 and version 200 is about two years, and the considered versions were released about 15-20 days apart. In terms of number of classes, in version 5 of Jun there are 170 classes while in version 200 there are than 740 classes.

[3]See http://www.srainc.com/Jun/ for more information.

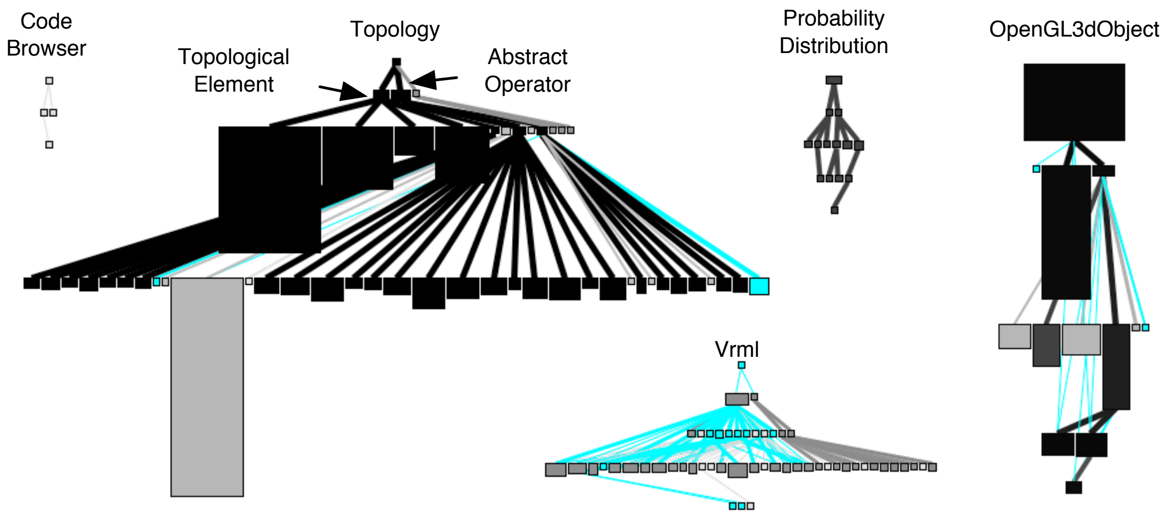


Figure 4: A *Class Hierarchy History Complexity View* of the evolution of five hierarchies from the Jun case study. The cyan nodes and edges denote removed classes and inheritance relationships.

Hierarchy	**Age**	**Inheritance Relationship**	**Stability**	**Effort Concentration**
Topology	Old	Reliable	Unstable	Unbalanced
CodeBrowser	Newborn	-	Stable	Balanced
OpenGL3dObject	Old	-	Unstable	Unstable Root, Unbalanced
Vrml	Persistent	Fragile	Stable	Balanced
ProbabilityDistribution	Old	Reliable	Stable	Balanced

Table 2: The characterization of five class hierarchies in Jun based on the four proposed characteristics.

Figure 4 shows five of the hierarchies we found when analyzing Jun: Topology, OpenGL3dObject, Vrml, ProbabilityDistribution, and CodeBrowser (we name the hierarchies according to the names of their root classes). In Table 2 we show the characterization of each hierarchy according to the proposed characteristics.

We selected those five because they represent five different types of hierarchies regarding their evolution over time:

The *Topology* hierarchy is the largest and oldest hierarchy in the system. In Figure 4 we marked the two sub hierarchies: *AbstractOperator* and *TopologicalElement*. The *TopologicalElement* subhierarchy is composed of classes which were changed a lot during their life time. Three of the leaf classes were detected as being GodClasses [14]. A large part of the *AbstractOperator* hierarchy has been in the system from the first version, but there is a young subhierarchy which looks different.

The *OpenGL3dObject* hierarchy experienced three times an insertion of a class in the middle of the hierarchy.

The *Vrml* hierarchy proved to have undergone heavy renaming refactorings. That is the reason why we see lots of removed nodes and removed inheritance relationships. Note also that the root class has been removed at a certain point in time, the original hierarchy has thus been split in two distinct hierarchies.

ProbabilityDistribution is an old hierarchy and very stable from the inheritance relationships point of view. Also, the classes in the hierarchy was changed very little during its history.

The *CodeBrowser* hierarchy is very thin and very light, meaning that it has been recently added to the system.

6 Related Work

Metrics and visualization are two traditional techniques used to deal with the problem of analyzing the history of software systems.

Lehmann used metrics starting from the 1970's to analyze the evolution of the IBM OS/360 system [11]. Lehmann, Perry and Ramil explored the implication of the evolution metrics on software maintenance [12] [13]. They used the number of modules to describe the size of a version and defined evolutionary measurements which take into account differences between consecutive versions.

Gall *et al.* [6] also employed the same kind of metrics while analyzing the continuous evolution of the software systems.

Burd and Munro analyzed the influence of changes on the maintainability of software systems. They define a set of measurements to quantify the dominance relations which are used to depict the complexity of the calls [2].

Lanza's Evolution Matrix [9] visualizes the system's history in a matrix in which each row is the history of a class (see a simplified version in Figure 1). A cell in the Evolution Matrix represents a class and the dimensions of the cell are given by evolutionary measurements computed on subsequent versions.

Jazayeri analyzed the stability of the architecture [7] by using colors to depict the changes.

Our approach differs from the above mentioned ones because we consider history to be a first class entity and define history measurements which are applied on the whole history of an entity and which summarize the evolution of that entity. The drawback of our approach consists in the inherent noise which resides in compressing large amounts of data into numbers.

Taylor and Munro [15] visualized CVS data with a technique called *revision towers*. Their approach resides at a different granularity level, *i.e.*, files, and thus does not display source code artifacts as in our approach.

Gall *et al.* [5] analyzed the history of changes in software systems to detect the hidden dependencies between modules. However, their analysis was at the file level, rather than dealing with the real code. In contrast, our analysis is placed at the class and inheritance level making the results finer grained. Demeyer *et al.* [4] propose practical assumptions to identify where to start a reverse engineering effort: working on the most buggy part first or focusing on clients most important requirements. These approaches, are based on information that is outside the code, while our analysis is based on code alone.

Another metrics-based approach to detect refactorings of classes was developed by Demeyer *et al.* [3]. While they focused on detecting refactorings, we focus on offering means to understand where and how the development effort was spent in a hierarchy.

7 Conclusions and Future Work

History holds useful information, but the analysis is difficult due to the large amount of data. We approached this problem by defining the history as a first class entity and then defined history measurements which summarize the evolution of an entity. We used the measurements to display a polymetric view of the evolution of class hierarchies. We applied our approach on a large open source project and showed how we could describe the evolution of class hierarchies.

It is difficult to validate and prove the value of our approach. We are convinced that the information that one can extract from an evolutionary polymetric view such as the *Class Hierarchy History Complexity View* is useful in different contexts. On the one hand, it reveals information about the system which would be otherwise difficult to extract (*e.g.,* knowing that a hierarchy is stable/unstable in time is valuable for deciding maintenance effort and doing quality assessment). On the other hand, we have to stress that polymetric views as we implement them are intrinsically interactive and that just looking at the visualization is only of limited value. Indeed, the viewer must interact with the visualization to extract finer-grained and more useful information. For example in Figure 4 one would like to know what class has been removed from the Topology hierarchy and also why, since this quite large hierarchy has been very stable in terms of inheritance relationships. The viewer can do so by pointing and inspecting the cyan class history node. In the case of the Vrml hierarchy one would like to find out why it is so unstable in terms of inheritance relationships, since one could expect that such a hierarchy merely implements a standardized framework like OpenGL3dObject.

In the future, we want to investigate possibilities of adding more semantic information to the view we propose. For example, we want to add information like refactorings that have been performed.

References

[1] E. Burd and M. Munro. An initial approach towards measuring and characterizing software evolution. In *Proceedings of the Working Conference on Reverse Engineering, WCRE '99*, pages 168–174, 1999.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA '2000 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 166–178, 2000.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[4] 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.

[5] H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance 1997 (ICSM '97)*, pages 160–166, 1997.

[6] M. Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.

[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. 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.

[9] 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.

[10] M. M. Lehman and L. Belady. *Program Evolution — Processes of Software Change*. London Academic Press, 1985.

[11] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolution metrics on software maintenance. In *ICSM*, pages 208–, 1998.

[12] L. MM, R. J. Perry DE, T. WM, and W. PD. Metrics and laws of software evolution - the nineties view. In *Metrics '97, IEEE*, pages 20 – 32, 1997.

[13] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu. Using history information to improve design flaws detection. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 233–232, 2004.

[14] C. M. B. Taylor and M. Munro. Revision towers. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50. IEEE Computer Society, 2002.