Characterizing the Evolution of Class Hierarchies

Tudor Gîrba
Software Composition Group
University of Bern
Switzerland
girba@iam.unibe.ch

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

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

Abstract

Analyzing historical information can show how a software system evolved into its current state, which parts of the system are stable and which have changed more. However, historical analysis implies processing a vast amount of information making the interpretation of the results difficult. To address this issue, we introduce the notion of the history *of source code artifacts as a first class entity and define measurements which summarize the* evolution *of such entities. We use these measurements to define rules by which to detect different characteristics of the evolution of class hierarchies. Furthermore, we discuss the results we obtained by visualizing them using a polymetric view*[1]*. We apply our approach on two large open source case studies and classify their class hierarchies based on their history.*

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

1 Introduction

History holds useful information that can be used when reverse engineering a system. However, analyzing historical information is difficult due to the vast amount of information that needs to be processed, transformed, and understood. Therefore, we need higher level views of the data which allow the reverse engineer to ignore the irrelevant details.

We concentrate on describing and characterizing the evolution of class hierarchies. Class hierarchies provide a grouping of classes based on similar semantics thus, characterizing a hierarchy as a whole reduces the complexity of understanding big systems. In particular, we seek answers to the following questions:

1. *How old are the classes of a hierarchy?* On one hand, the old classes may be part of the original design and thus hold useful information about the system's design. On the other hand, a freshly introduced hierarchy might indicate places where future work will be required.

2. *Were there changes in the inheritance relationships?* Changes in the inheritance relationships might indicate class renaming refactorings, introduction of abstract classes, or removal of classes.

3. *Are classes from one hierarchy modified more than those from another one?* Understanding which parts of the system changed more is important because they might be also changed in the future [9].

4. *Are the changes evenly distributed among the classes of a hierarchy?* If we find that the root of the hierarchy is often changed, it might indicate that effort was spent to factor out functionality from the subclasses and push it to the superclass, but it might also be a sign of violations of the open-closed principle [18].

All information needed to answer the above questions is already in the history of a system, but in raw form and not easily accessible. We want to stress that processing such a huge amount of data and understanding the results are difficult, therefore there is a need for an adequate approach to analyze and understand the amount of information. Furthermore, the above questions need a model which represents semantical meaningful changes (*e.g.,* did the superclass of a particular class change from one version to another) and not just text-based change logs.

Rather than placing our analysis at the level of change logs, we define a meta-model in which history is a first class entity and which allows us to define history measurements which summarize the evolution of an entity or a set of entities. For example, we quantify how much a class was changed in terms of number of methods.

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

We are set to detect four characteristics of class hierarchy evolution: (1) the age of the hierarchy, (2) the inheritance relationship stability, (3) the class size stability, and (4) the change balance. We quantify these characteristics in a set of measurements-based rules, and define a language for describing different evolution patterns of class hierarchies.

To analyze the obtained results we developed a visualization called *Hierarchy Evolution Complexity View*, an evolutionary polymetric view [12]. We use *software visualization* because visual displays allow the human brain to study multiple aspects of complex problems – like reverse engineering – in parallel [20].

We validated our approach on two open source case studies written in Java and Smalltalk.

The contributions of the paper are (1) the characterization of class hierarchy evolution based on explicit rules which detect different change characteristics, and (2) the definition of a new polymetric view which takes into account the history of entities.

Structure of the paper. In Section 2 we introduce the notion of a history and then define history measurements for characterizing the evolution of classes and class hierarchies. In Section 3 we define rules to detect different evolution patterns of the evolution of class hierachies and in Section 4 we introduce the visualization based on the defined measurements. In Section 5 we apply our approach on two large case studies. Prior to concluding we discuss variation points of our approach (Section 6), present its implementation (Section 7) and discuss related work (Section 8).

2 Modeling History

We define a *history* to be a sequence of versions of the 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.). Centered around the notion of history we build a history meta-model called Hismo [5].

We then define historical measurements which characterize the evolution of classes and inheritance relationships and then use them to characterize the evolution of class hierarchies.

2.1 Measuring Class Histories

History Age. We define the *Age* as being the number of versions of a history. In this paper we talk about the *Age* of a class history ($ClassAge$), of an inheritance relationship history ($InhAge$), or of a system history ($SystemAge$).

Removed history. A history has been removed if its last version is not part of the last version of the system. Depending on the kind of history we analyze, we can refer to a removed class history, a removed inheritance relationship history, etc.

Evolution of a Version Property (*E*). We define $E_{1..n}(P,H)$ as the sum of the absolute difference of property P in subsequent versions from version 1 (the first version) to version n (the latest version) of a history H [9]:

$$(n>1) \quad E_{1..n}(P,H) = \textstyle\sum_{i=2}^{n} |P_i(H) - P_{i-1}(H)| \qquad (1)$$

We instantiate this measurement by applying it on different version properties of classes such as: *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*).

$$ENOM_{1..n}(C) = E_{1..n}(NOM, C) \qquad (2)$$

$$ENOS_{1..n}(C) = E_{1..n}(NOS, C) \qquad (3)$$

As a short notation, we use $E(P,H) = E_{1..n}(P,H)$.

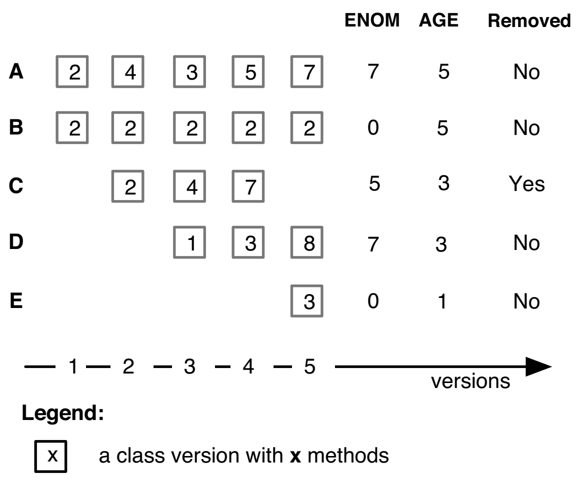


Figure 1. An example of history measurements.

Example. In Figure 1 we display an Evolution Matrix [13] 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 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 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. Class E has been added to the system in the last version.

2.2 Measuring Class Hierarchy Histories

We consider a class hierarchy as being a group of classes and a group of inheritance relationships. To measure class hierarchy histories, we apply group operators like average (*Avg*), maximum (*Max*) and total (*Tot*). Thus we have the average age of the class histories in a class hierarchy Ch as given by: $Avg(ClassAge, Ch)$.

3 Characterizing Class Hierarchy Histories

We formulate a vocabulary based on four characteristics: (1) hierarchy age, (2) inheritance relationship stability, (3) class size stability, and (4) change balance.

These four characteristics are orthogonal with each other (*e.g.,* the same hierarchy can be old but at the same time can have unstable classes).

In the followings, we use the history measurements to define rules to qualify a class hierarchy based on the four characteristics:

1. How old is a hierarchy? We distinguish the following types of hierarchy histories based on the average age of their class histories:
   - *Newborn*. A newborn hierarchy is a freshly introduced hierarchy (*i.e.,* on the average the age of the class histories is no more than a tenth of the age of the system).

     $$Avg(ClassAge, Ch) < 0.1 * SystemAge$$

   - *Young*. A young hierarchy is older than a newborn hierarchy, but its age is less than half of the system age.

     $$Avg(ClassAge, Ch) \geqslant 0.1 * SystemAge \land \\ Avg(ClassAge, Ch) < 0.5 * SystemAge$$

   - *Old*. Old hierarchies have been in the system for a long time, but not for the entire life of the system.

     $$Avg(ClassAge, Ch) \geqslant 0.5 * SystemAge \land \\ Avg(ClassAge, Ch) < 0.9 * SystemAge$$

   - *Persistent*. We say a hierarchy is persistent if the classes were present in almost all versions of the system (*i.e.,* in more than 90% of the system versions).

     $$Avg(ClassAge, Ch) \geqslant 0.9 * SystemAge$$

2. Were there changes in the inheritance relationship? We divide the hierarchies into two categories:
   - *Solid*. We define a hierarchy as being solid when the inheritance relationships between classes are stable and old.

     $$Tot(RemovedInh, Ch) < 0.3 * NOInh(Ch)$$

   - *Fragile*. A hierarchy is fragile when there are many inheritance relationships which disappear.

     $$Tot(RemovedInh, Ch) \geqslant 0.3 * NOInh(Ch)$$

3. Are classes from one hierarchy modified more than classes from another hierarchy? From the stability of size point of view we detect two kind of hierarchies:
   - *Stable*. In a stable hierarchy the classes have few methods and statements added or removed compared with the rest of the system.

     $$Avg(ENOM, Ch) < Avg(ENOM, S) \land \\ Avg(ENOS, Ch) < Avg(ENOS, S)$$

   - *Unstable*. In an unstable hierarchy many methods are being added and removed during its evolution.

     $$Avg(ENOM, Ch) \geqslant Avg(ENOM, S) \lor \\ Avg(ENOS, Ch) \geqslant Avg(ENOS, S)$$

4. Are the changes evenly distributed among the classes of a hierarchy? We distinguish two labels from the change balance point of view:
   - *Balanced*. In a balanced hierarchy, the changes are evenly performed among its classes.

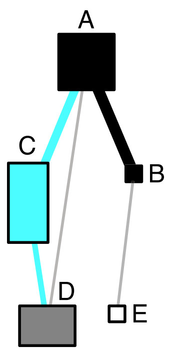
     $$Avg(ENOM, Ch) \geqslant 0.8 * Max(ENOM, Ch)$$

   - *Unbalanced*. An unbalanced hierarchy is one in which the changes are not equally distributed in the classes.

     $$Avg(ENOM, Ch) < 0.8 * Max(ENOM, Ch)$$

4 Class Hierarchy History Complexity View

To analyze the results we obtain when applying the rules defined in the previous section, we use visualization as it allows one to identify cases where different characteristics apply at the same time for a given hierarchy. The visualization we propose is called *Hierarchy Evolution Complexity View* and is a polymetric view[12]. *Hierarchy Evolution Complexity View* uses a simple tree layout to seemingly display classes and inheritance relationships. What is new is that it actually visualizes the *histories* of classes and of inheritance relationships.



Legend:

Nodes:	Class Histories
Node Width:	ENOM of the class history
Node Height:	ENOS of the class history
Node Color:	AGE of the class history; Cyan for removed history
Edges:	Inheritance Histories
Edge Width:	AGE of the inheritance history
Edge Color:	AGE of the inheritance history; Cyan for removed history

Figure 2. An example of the *Hierarchy Evolution Complexity View*.

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

The width of the class history node is given by the Evolution of Number of Methods (*i.e., ENOM*) while the height is given by the fifth part of the Evolution of Number of Statements (*i.e., ENOS*/5) [3]. 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 *ENOM* and *ENOS* because we wanted to see the correlation between changes in the behavior of a class and its internal implementation changes.

Example. In Figure 2 we show an example of such a view in which we display an imaginary hierarchy history of the class histories presented in Figure 1. From the figure we infer the following information:

- Classes A and B are in the system from the very beginning, and they appear colored in black. The inheritance relationship between these classes is black and thick marking that the relationship is old. Class E is small and white, because it was recently introduced in the system.

- Class C was removed from the system and is colored in cyan. Class D has been introduced after several versions as a subclass of C, but in the last version it has become a subclass of A.

- Class B is small because there were no changes detected in it. Classes A and D have the same width, but class D appears to have less statements added or removed because the node is more wide than tall. Class C is much taller compared to its width, denoting a greater implementation effort.

Based on the visualization we can detect two more characteristics:

- *Heterogeneous*. We characterize a class hierarchy history as being heterogeneous if the class histories have a wide range of ages. Such a hierarchy will appear colored with a wide range of grays.

- *Unstable Root.* In a hierarchy with an unstable root, the root node is large compared with the rest of the nodes.

5 Characterizing the Class Hierarchy Histories of JBoss and Jun

For our experiments we chose two open source systems: JBoss[4] and Jun[5]. Table 1 gives an overview of the case studies.

System	Language	Versions	First Version (Size)	Last Version (Size)
JBoss	Java	14	35 kLOC 628 classes	302 kLOC 4975 classes
Jun	Smalltalk	40	12 kLOC 170 classes	132 kLOC 740 classes

Table 1. Characteristics of the JBoss and Jun case studies.

JBoss. For the first case study, we chose 14 versions of JBoss. JBoss is an open source J2EE application server written in Java. The versions we selected for the experiments were at two months distance from one another starting from the beginning of 2001 until 2003. The first version has 628 classes, the last one has 4975 classes.

Jun. As a second case study we selected 40 versions of Jun. Jun is a 3D-graphics framework written in Smalltalk currently consisting of more than 700 classes. 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

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

[3]We chose to divide *ENOS* by 5 because in the case studies we analyzed there was an average of 5 statements per method.

[4]See http://www.jboss.org for more information.

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

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 740 classes.

Table 2 and Table 3 show some of the results we obtained on six large hierarchies of both case studies. These results are now explained using the visualization.

5.1 Case Study: JBoss

Figure 3 shows the largest hierarchy in JBoss (ServiceMBeanSupport), and Figure 4 shows five other hierarchies from JBoss: JBossTestCase, J2EEManagedObject, Stats, MetaData and SimpleNode (we name the hierarchies according to the names of their root classes). For space reasons Figure 3 is scaled with a 0.5 ratio (*i.e.,* zoomed out) as compared with the Figure 4. In Table 2 we show the characterization of each hierarchy according to the proposed characteristics.

ServiceMBeanSupport is a large hierarchy with nodes and edges of different colors and shapes: As a whole, we classify it as heterogeneous from the age point of view, fragile from the inheritance relationship point of view, and unstable and unbalanced from the changes point of view.

The J2EEManagedObject is a sub-hierarchy of ServiceMBeanSupport and is heterogeneous and unbalanced from the point of view of performed changes and is fragile in terms of inheritance. In the visualization the hierarchy is displayed with nodes colored in various grays and with many cyan edges and nodes.

JBossTestCase is composed of classes of different age. On average, the nodes are rather small, meaning that the classes were stable. The hierarchy is heterogeneous from the class age point of view which shows that the tests were continuously developed along the project. Also, because most of the nodes are small it means that most of the classes are stable both from the methods and from the implementation point of view. In the unstable classes there were more implementation changes than methods added or removed. Also, once the test classes were created not many test methods were added or removed afterwards.

The Stats hierarchy has been recently introduced and did not yet experience major changes.

MetaData is represented with nodes of different sizes colored either in dark colors or in cyan: It is an old hierarchy which is unstable and unbalanced from the performed changes point of view. The edges are either thick and dark or cyan, and as the number of removed edges are not high, we characterize the inheritance relationships as being solid.

The SimpleNode hierarchy is fairly old and experienced very few changes during its lifetime, making it thus a very stable hierarchy.

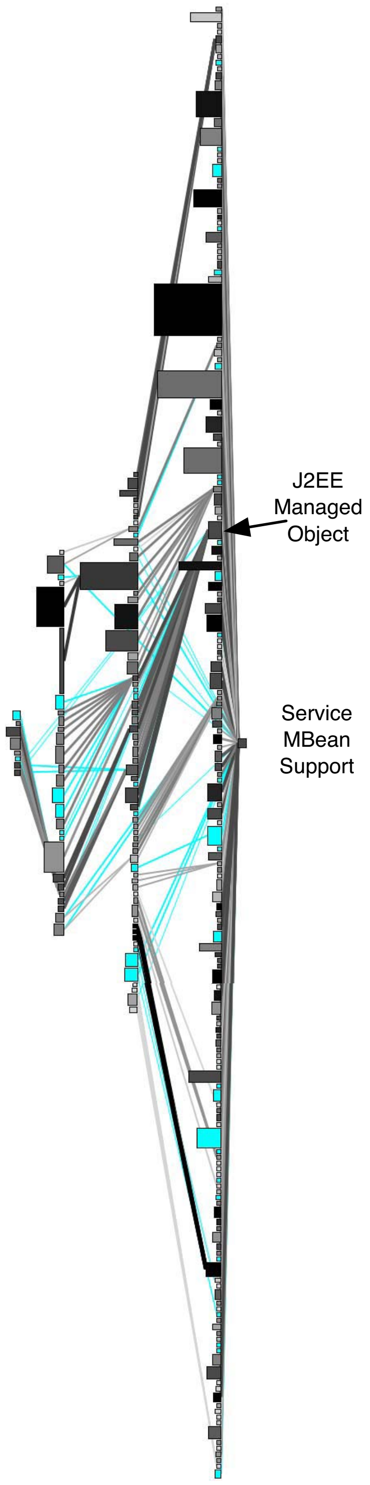


Figure 3. A *Hierarchy Evolution Complexity View* **of the evolution of the largest hierarchy from 14 versions of JBoss.**

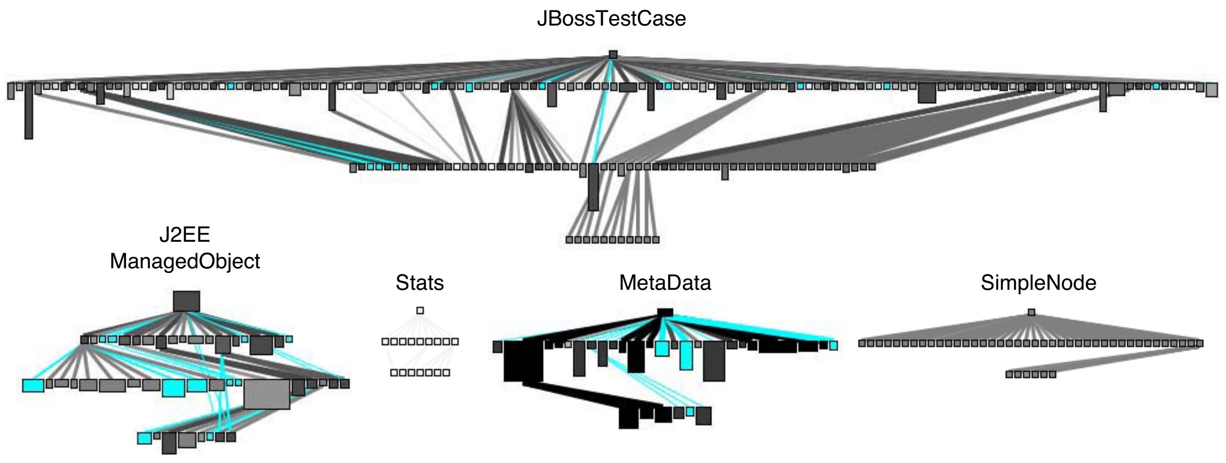


Figure 4. A *Hierarchy Evolution Complexity View* **of the evolution of five hierarchies from 14 versions of JBoss.**

Hierarchy	Age	Inheritance Relationships	Stability	Change Balance
ServiceMBeanSupport	Old, Heterogeneous	Fragile	Unstable	Unbalanced
JBossTestCase	Old, Heterogeneous	Solid	Stable	Balanced
J2EEManagedObject	Old, Heterogeneous	Fragile	Unstable	Unbalanced
Stats	Newborn	Solid	Stable	Balanced
MetaData	Old	Solid	Unstable	Unbalanced
Simple Node	Old	Solid	Stable	Balanced

Table 2. The characterization of five class hierarchies from JBoss.

5.2 Case Study: Jun

Figure 5 shows six of the hierarchies of Jun: Topology, CodeBrowser, OpenGL3dObject, Vrml, OpenGLDisplayModel, and ProbabilityDistribution. In Table 3 we show the characterization of each hierarchy according to the proposed characteristics.

The Topology hierarchy is the largest and oldest hierarchy in the system. In Figure 5 we marked the two sub-hierarchies: AbstractOperator and TopologicalElement. The TopologicalElement sub-hierarchy is composed of classes which were changed a lot during their life time. Three of the leaf classes were detected as being God Classes [19]. A large part of the AbstractOperator hierarchy has been in the system from the first version, but there is a young sub-hierarchy which looks different.

The CodeBrowser hierarchy is thin and lightly colored, meaning that it has been recently added to the system.

The OpenGL3dObject hierarchy experienced three times an insertion of a class in the middle of the hierarchy: There are removed inheritance relationships colored in cyan.

The Vrml hierarchy proved to have undergone extensive renaming refactorings: We see many removed nodes and removed inheritance relationships. Even the root class has been removed at a certain point in time: The original hierarchy has thus been split in two distinct hierarchies.

The OpenGLDisplayModel hierarchy has an old and unstable root. This is denoted by a large rectangle colored in dark gray. The Chart sub-hierarchy is thin and lightly colored denoting it is newborn.

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

6 Discussion and Variation Points

It is difficult to empirically validate and prove the value of our approach. However, our experience with two large and significant industrial applications shows that the approach provides valuable information in a short time.

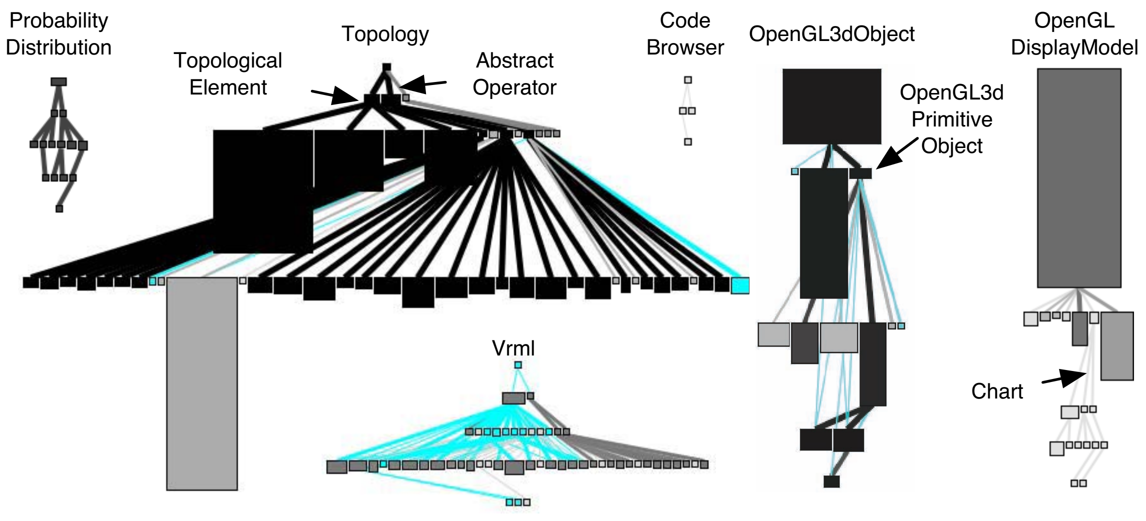


Figure 5. A *Hierarchy Evolution Complexity View* **of the evolution of six hierarchies from the 40 versions of Jun.**

Hierarchy	Age	Inheritance Relationships	Stability	Change Balance
Topology	Old	Solid	Unstable	Unbalanced
CodeBrowser	Newborn	Solid	Stable	Balanced
OpenGL3dObject	Old	Fragile	Unstable	Unbalanced, Unstable Root
Vrml	Persistent	Fragile	Stable	Balanced
OpenGLDisplayModel	Old	Solid	Stable	Balanced, Unstable Root
ProbabilityDistribution	Old	Solid	Stable	Balanced

Table 3. The characterization of six class hierarchies from Jun.

6.1 Measurements Variation Points

The presented measurements have already been used by the authors to describe changes in a system and on that occasion some variation points have been discussed [9]:

- A variation point when computing E measurement is the release period. If, for example, we consider the release period of one week, we focus the analysis to immediate changes. If we consider the release period of half a year, we emphasize the size of the changes that accumulate in the class histories.
- The number of versions is another variation point when computing the measurements. By increasing the number of analyzed versions we obtain a long-term indicator of effort, while by decreasing the number of versions we concentrate on the short-term indicator of effort.
- Changing the threshold used for characterizing the evolution is also a variation point. For example, instead of using 10% of the total number of system versions for qualifying a class hierarchy as young, we can use 3 versions as the threshold (that is the classes should be introduced no later than 2 versions ago).

6.2 Visualization Variation Points

In our visualization we sought answers to four questions regarding the age of the hierarchy, the inheritance relationship stability, the class size stability and the change balance. The purpose of the visualization is to provide an overview of the evolution of hierarchies, but is of limited use when a deep understanding is required.

However, we are convinced that the information that one can extract from the analysis we propose and the use of an evolutionary polymetric view such as the *Hierarchy Evolution Complexity View* is useful: 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). In addition, 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 visualiza-

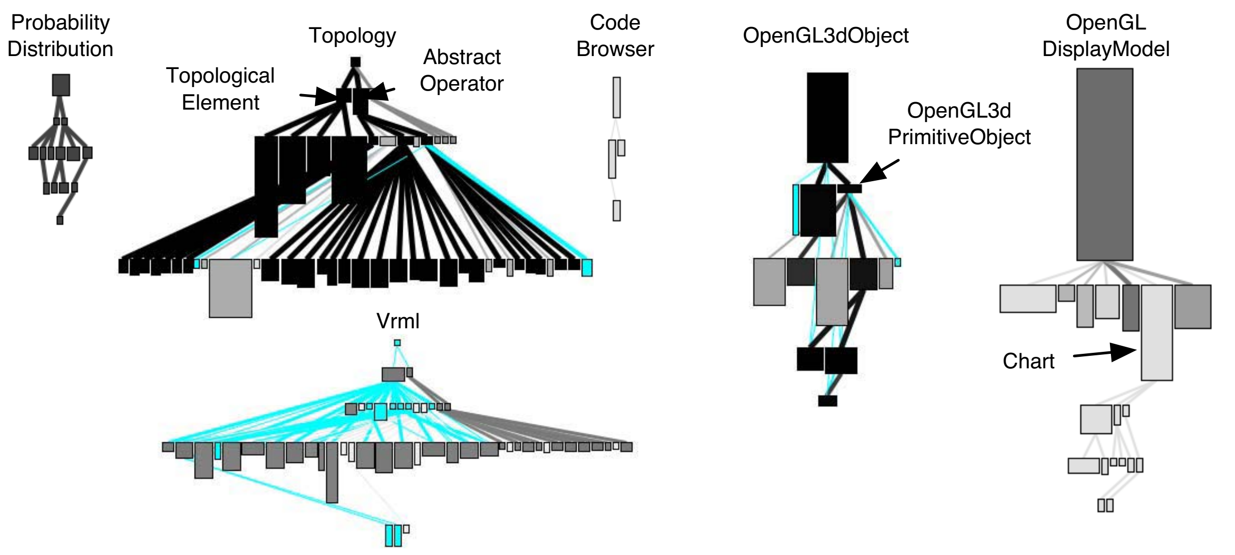


Figure 6. A modified *Hierarchy Evolution Complexity View* of the evolution of six hierarchies from the Jun case study (node width = NOM instability; node height = last NOM).

tion to extract finer-grained and more useful information, *e.g.,* accessing the source code.

Example. In Figure 5 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.

Figure 6 shows a modified *Hierarchy Evolution Complexity View* applied on the Jun hierarchies. In this view we used for the width and the height of the nodes other measurements, namely the last number of methods (NOM) as the height and the instability of the number of methods (INOM) as the width. The instability of the number of methods is computed as the number of versions in which the number of methods changed over the total number of versions in which it could have changed.

While in the original view the nodes dimensions show the correlation between implementation effort and the behavior addition and removal effort, this view shows the correlation of the actual size in terms of number of methods and the number of times methods were added or removed in the classes. Thus, in this view we can detect whether the instability is correlated with actual size.

For example, in the OpenGLDisplayModel hierarchy there is a correlation of the instability of number of methods and the size of the root class because the node is tall and wide. On the other hand, the Chart hierarchy is newborn and appears small in the original view, while in this view the root is tall meaning that it is a large class.

6.3 History Meta-Model Design Decisions

The history meta-model on which the work presented in this paper is built, is called *Hismo* [5]. The novelty of this meta-model is the introduction of the notion of history as a first class entity.

Figure 7 shows how a meta-model centered around the notion of history can be built: each cell in the matrix is a Class Version which makes each line represent a Class History. Moreover, the whole matrix is actually a line formed by SystemVersions, which means that the whole matrix can be seen as a SystemHistory. In the right side of the figure we built a small meta-model which shows that a SystemHistory has more ClassHistories. If we have a structural meta-model in which we have other entities defined (*i.e.,* other than System and Class), we can build in the same way the corresponding history entities and their relationships. In our implementation, Hismo is based on *FAMIX*, a language independent meta-model [4].

The use of a meta-model allows one to express rules at a high-level of abstraction as shown in Section 3. The current analysis and the way it is expressed is an illustration of the power given by such a meta-model.

7 Implementation: CodeCrawler, Van, and Moose

The focus of our tools is on dynamic queries and on interactivity rather than on static report generation.

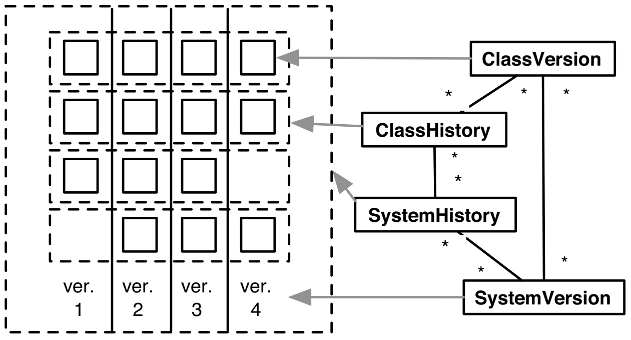


Figure 7. Example of building Hismo.

The presented visualizations are generated using *CodeCrawler* [14]. CodeCrawler supports reverse engineering through a lightweight combination of metrics and software visualization. In its latest implementation CodeCrawler has become a general-purpose information visualization tool. The underlying evolutionary model is implemented in *Van* which is our history analysis tool.

Both Van and CodeCrawler are based on the *Moose* [6] reengineering environment, which is an implementation of the FAMIX meta model. Both the FAMIX and the Hismo metamodels are language independent. Our complete toolset is implemented in Smalltalk and is open-source and available at: http://www.iam.unibe.ch/∼scg/Research/Moose/index.html.

8 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 [16]. Lehmann *et al.* explored the implication of the evolution metrics on software maintenance [15] [17]. 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.* also employed the same kind of metrics while analyzing the continuous evolution of the software systems [8].

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

Lanza's Evolution Matrix [13] visualized 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 [11] by using colors to depict the changes. Jingwei Wu *et al.* used the spectograph metaphor to visualize how changes occur in software systems. [23]. Rysselberghe and Demeyer use a simple visualization based on information in version control systems to provide an overview of the evolution of systems [22].

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 authors already used the notion of history to analyze how changes appear in the software systems [9]. The drawback of our approach consists in the inherent noise which resides in compressing large amounts of data into numbers.

Taylor and Munro [21] visualized CVS data with a technique called *revision towers*. Ball and Eick [1] developed visualizations for showing changes that appear in the source code. These approaches reside at a different granularity level, *i.e.,* files, and thus does not display source code artifacts as in our approach.

Holt and Pak [10] proposed a detailed visualization of the old and new dependencies between modules. Collberg *et al.* used graph-based visualizations to display the changes authors make to class hierarchies. However, they did not give any representation of the dimension of the effort and of the removals of entities.

Gall *et al.* [7] analyzed the history of changes in software systems to detect the hidden dependencies between modules. Xiaomin Wu *et al.* also visualized [24] the change log information to provide for an overview of the active places in the system as well as of the authors activity. 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. These approaches, are based on information that is outside the code, while our analysis requires only the code.

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.

9 Conclusions and Future Work

This work set to answer four questions: (1) How old are the classes of a hierarchy?, (2) Were there changes in the inheritance relationship?, (3) Are classes from one hierarchy modified more than those from another one?, and (4) Are the changes evenly distributed among the classes of a hierarchy?

The history of a system holds the information necessary for answering the above questions, 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 we defined history measurements which summarize the evolution of an entity.

Based on the questions we formulated a vocabulary of terms and we used the measurements to formulate rules to characterize the evolution of class hierachies. Furthermore, we displayed the results using a new polymetric view of the evolution of class hierarchies called *Hierarchy Evolution Complexity View*.

We applied our approach on two large open source projects and showed how we could describe the evolution of class hierarchies.

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

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004) and "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

References

[1] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, 1996.

[2] 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.

[3] 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.

[4] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[5] S. Ducasse, T. Gîrba, and J.-M. Favre. Modeling software evolution by treating history as a first class entity. In *Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 71–82, 2004.

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

[7] 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.

[8] 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.

[9] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM 2004 (International Conference on Software Maintenance)*, pages 40–49, 2004.

[10] R. C. Holt and J. Pak. GASE: Visualizing software evolution-in-the-large. In *Proceedings of WCRE '96*, pages 163–167, 1996.

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

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

[13] 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.

[14] M. Lanza and S. Ducasse. Codecrawler an extensible and language independent 2d and 3d software visualization tool. In *Reengineering Environments*. tba, 2004. to appear.

[15] M. Lehman, D. E. Perry, J. F. Ramil, W. M. Turski, and P. D. Wernick. Metrics and laws of software evolution - the nineties view. In *Metrics '97, IEEE*, pages 20 – 32, 1997.

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

[17] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of evolution metrics on software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 208–217, 1998.

[18] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[19] D. Raţiu, S. Ducasse, T. Gîrba, and R. Marinescu. Using history information to improve design flaws detection. In *Proceedings of CSMR 2004 (European Conference on Software Maintenance and Reengineering)*, pages 223–232, 2004.

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

[21] 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.

[22] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004. to appear.

[23] J. Wu, R. Holt, and A. Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89. IEEE Computer Society Press, Nov. 2004.

[24] X. Wu, A. Murray, M.-A. Storey, and R. Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 90–99. IEEE Computer Society Press, Nov. 2004.