Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes

Tudor Gîrba, Stéphane Ducasse, Michele Lanza
Software Composition Group - University of Bern, Switzerland
{girba, ducasse, lanza}@iam.unibe.ch

Abstract

[1]*Knowing where to start reverse engineering a large software system, when no information other than the system's source code itself is available, is a daunting task. Having the history of the code (i.e., the versions) could be of help if this would not imply analyzing a huge amount of data. In this paper we present an approach for identifying candidate classes for reverse engineering and reengineering efforts. Our solution is based on summarizing the changes in the evolution of object-oriented software systems by defining history measurements. Our approach, named* Yesterday's Weather, *is an analysis based on the retrospective empirical observation that classes which changed the most in the recent past also suffer important changes in the near future. We apply this approach on two case studies and show how we can obtain an overview of the evolution of a system and pinpoint its classes that might change in the next versions.*

Keywords: software evolution, reverse engineering, object-oriented programming, program understanding

1 Introduction

When starting a reverse engineering effort, knowing where to start is a key question. When only the code of the application is available, the history of a software system could be helpful. However, analyzing a software system's history is difficult due to the large amount of complex data that needs to be interpreted. Suppose we had as case study 40 versions of a software system's code, each version consisting on average of ca. 400 classes. We would have to analyze ca. 16000 classes, *i.e.,* class versions, which would make the analysis more difficult. Still, the history of the system contains valuable information about the life of the system, its growth, decay, refactoring operations, and bug-fixing phases.

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.

The basic assumption of this work is that the parts of the system which change are those that need to be understood first. We can find about the tendencies of changes by looking at the history of the system. However, not every change in the history of the system is relevant for the future changes. For example, the parts of a system which were changed in its early versions are not necessarily important[2] for the close future: Mens and Demeyer suggested that the *evolution-prone* parts of a system are those which changed a lot lately [16].

In this paper we aim to measure how relevant it is to start reverse engineering from the parts of the system which changed the most in the recent past. Based on historical information, we identify the parts of the system that changed the most in the recent past and check the assumption that they are likely to be among the most changed classes in the near future. If this assumption held many times in the system history, then the recently changed parts are good candidates for reverse engineering. Our experiments showed that important changes do not necessarily imply that they only occur in big classes (in terms of lines of code, methods or attributes). Therefore identifying the big classes in the last version of a software system is not necessarily relevant for its near future.

In this paper we analyze the evolution of object-oriented systems. We analyze classes as they are the primary abstractions from which applications are built. We identify classes that are likely to change by defining evolutionary measurements which summarize the history of object-oriented systems. We show the relevance of these measurements in our

[1]Proceedings of ICSM 2004 (International Conference on Software Maintenance), 2004, pp. 40–49

[2]By *important* we denote the fact that these classes will be affected by changes.

approach which we named *Yesterday's Weather*. It is similar to the historical observation of the weather: a good way of guessing what the weather will be like tomorrow is to think it will stay the same as today, which depending on the location can have very high success rates. However, this probability is not the same in all places. For example, in the Sahara desert the chance that the weather stays the same from one day to the next is higher than in Switzerland, where the weather can change in a few hours. Therefore, before we can use such a predictive and intrinsically fuzzy approach for "successful weather forecasts", we need historical information about the climate of the place we are interested in.

Yesterday's Weather is a measurement applied on a system history and it characterizes the "climate" of a software system. More specifically, *Yesterday's Weather* provides an indication that allows one to evaluate the relevance of starting reverse engineering from the classes that changed the most recently.

We start by presenting an overview of our approach in Section 2. In Section 3 we define the measurements needed to compute the *Yesterday's Weather*. In Section 4 we present the results obtained on two case studies, and then discuss the variation points of our approach. Before concluding, we discuss related work. In the appendix we show key aspects of the implementation of the tools we use.

2 *Yesterday's Weather* in a Nutshell

We define *Yesterday's Weather* (*YW*) to be the retrospective empirical observation of the phenomenon that at least one of the classes which were heavily changed in the recent history is also among the most changed classes in the near future.

Our approach consists in identifying, for each version of a subject system, the classes that were changed the most in the recent history and in checking if these are also among the most changed classes in the successive versions. We count the number of versions in which this assumption holds and divide it by the total number of analyzed versions to obtain the value of *Yesterday's Weather*.

Example. Suppose that for a system *YW* yields a value of 50%. This means that the history of the system has shown that in 50% of the cases at least one of the classes that was changed a lot in the recent past would also be among the most changed classes in the near future.

YW characterizes the history of a system and is useful from a reverse engineering point of view to identify classes that are likely to change in the next version. On one hand we use such information to make out progressive development phases in the evolution of the system (*e.g.,* what is/was the current focus of development?). In phases where the developers concentrate on a certain part of the system, the evolution would be fairly easy to predict. During repairing phases due to bug reports or little fixes the developers change the software system apparently at random places which leads to a decrease of predictability (*i.e.,* the weather becomes unstable). On the other hand it also gives a general impression of the system (*i.e.,* how stable is the climate of the whole system?). By interpreting the *YW* we identify that the changes are either focused on some classes over a certain period of time, or they move unpredictably from one place to another.

Example. If the *YW* value of a software system S_1 is 10%, this implies that the changes in the system were rather discontinuous – maybe due to new development or bug fix phases. If the *YW* yields an 80% value for a system S_2, this implies the changes in the system were continuous. In such a system, we say it is relevant to start the reverse engineering from the classes which were heavily changed lately, while this is not the case in system S_1.

3 *Yesterday's Weather* in Detail

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.).

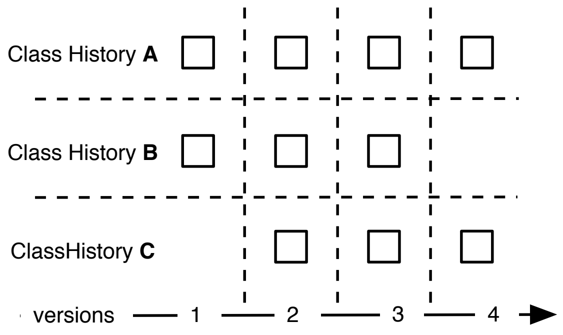


Figure 1. Example of a system history displayed in an Evolution Matrix.

Example. In Figure 1 we use a simplified example of the Evolution Matrix [12] to display a system history with four versions. A cell in the matrix is marked by a square and represents a class version. A line in the matrix represents a class history and a column represents a system version. In Figure 1, 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.

3.1 History Measurements

A class is the primary unit of design and structure in object-oriented systems. The functionality of the system is defined by classes and their methods. We consider as change to a class an addition or a removal of at least one method. Therefore, to measure the changes in the history of a class we measure the differences in their number of methods in different versions. In the following paragraphs we define 3 measurements that are necessary to compute the *YW* value of a software system, namely the (1) *Evolution of Number of Methods* (*ENOM*), the (2) *Latest Evolution of Number of Methods* (*LENOM*), and the (3) *Earliest Evolution of Number of Methods* (*EENOM*).

1. **Evolution of Number of Methods (*ENOM*)**

   We define $ENOM_i$ as being the difference in the number of methods between version $i-1$ and i of the class C:

   $$(i > 1) \quad ENOM_i(C) = |NOM_i(C) - NOM_{i-1}(C)| \quad (1)$$

   $ENOM_{j..k}$ is the sum of the number of methods added or removed in subsequent versions from version j to version k out of n versions of a class C:

   $$(1 \leqslant j < k \leqslant n) \\ ENOM_{j..k}(C) = \textstyle\sum_{i=j+1}^{k} ENOM_i(C) \quad (2)$$

   We use this measurement to show the overall changes performed during the lifetime of a class.

2. **Latest Evolution of Number of Methods (*LENOM*)**

   When computing $ENOM_{j..k}$ all the changes have the same importance. With $LENOM_{j..k}$ (see Equation 3) we favor the recent changes over the changes further in the past by applying a weighting function 2^{i-k} which decreases the importance of a change as the version i in which it occurs is more distant from the latest considered version (k).

   $$(1 \leqslant j < k \leqslant n) \\ LENOM_{j..k}(C) = \textstyle\sum_{i=j+1}^{k} ENOM_i(C) 2^{i-k} \quad (3)$$

3. **Earliest Evolution of Number of Methods (*EENOM*)**

   As opposed to *LENOM*, we define *EENOM* to favor the changes closer to the first version of the history over the changes that appear closer to the latest version of the considered history (see Equation 4).

   $$(1 \leqslant j < k \leqslant n) \\ EENOM_{j..k}(C) = \textstyle\sum_{i=j+1}^{k} ENOM_i(C) 2^{k-i+1} \quad (4)$$

3.1.1 Interpretation of the Measurements

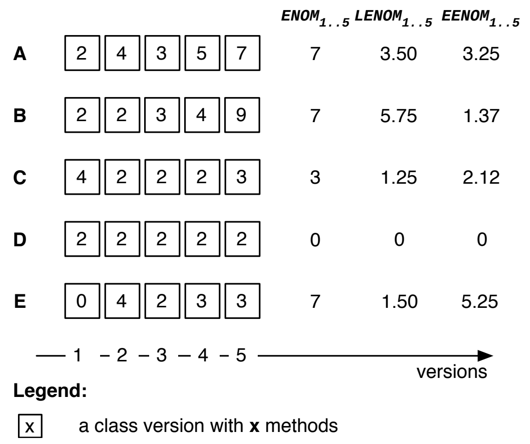
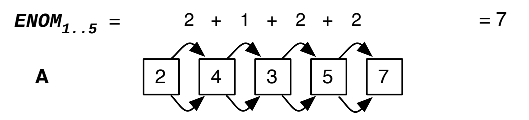


Figure 2. Examples of the computation of the *ENOM*, *LENOM* and *EENOM* class history measurements. Figure 3 shows in more details how we obtained the values applied on the history of class A.



$LENOM_{1..5}$ = 2*2^{-3}+ 1*2^{-2}+ 2*2^{-1}+ 2*2^{0} = 3.50

$EENOM_{1..5}$ = 2*2^{0} + 1*2^{-1}+ 2*2^{-2}+ 2*2^{-3} = 3.25

Figure 3. Detailed computation of *ENOM*, *LENOM* and *EENOM* for class history A.

Figure 2 presents an example of a system history displayed with an Evolution Matrix view [12] in which each square represents a class version and the number inside the square represents the number of methods of that particular class version. On the right side of Figure 2 we display the computed values of *ENOM*, *LENOM*, and *EENOM* for the 5 class histories (A, B, C, D and E). Figure 2 shows the following:

- During the displayed history (5 versions) of class D the number of methods remained 2. We consider that no methods were added or removed, therefore the values of *ENOM*$_{1..5}$, *EENOM*$_{1..5}$ and of *LENOM*$_{1..5}$ of this class history are 0.
- In the histories of class A, of class B and of class E, 7 methods were detected as being added or removed. The class histories differ in their *LENOM*$_{1..5}$ and *EENOM*$_{1..5}$ values which means that (i) the changes are more recent in the history of class B, (ii) in class E the changes occurred in its early history, and (iii) in the history of class A the changes were scattered through the history more evenly.
- The histories of class C and E have almost the same *LENOM*$_{1..5}$ value, because of the similar amount of changes in their recent history. The *ENOM*$_{1..5}$ values differ heavily because class E was changed more throughout its history than class C.

3.2 Measuring *Yesterday's Weather*

Before defining the *YW* function, we introduce the notion of *top n* of entities out of an original set *S* of entities with the highest *M* measurement value:

$$(0 < n < 1) \quad Top_M(S,n) = S' \left| \begin{array}{l} S' \subseteq S, \\ |S'| = n \\ \forall x \in S', \forall y \in S - S' \\ M(x) > M(y) \end{array} \right. \tag{5}$$

For a system version i, we compare the set of class histories with the highest *LENOM*$_{1..i}$ values (the *candidates* set) with the set of the class histories with the highest *EENOM*$_{i..n}$ values (the *really-changed* set). The *Yesterday's Weather* assumption holds if the intersection of these sets is not empty, that is at least one class history belongs to both sets. This means that for the classes in version i at least one of the recently most changed classes is among the most changed classes in the near future relative to version i. If the assumption holds for version i we have a hit.

We formally define the *Yesterday's Weather* hit function applied on version i of a system history S and given the two threshold values t_1 and t_2 as follows:

$$(i > 1; t_1, t_2 \geq 1)$$
$$YW_i(S,t_1,t_2) = \begin{cases} 1, & Top_{LENOM_{1..i}}(S,t1) \cap \\ & Top_{EENOM_{i..n}}(S,t2) \neq \emptyset \\ 0, & Top_{LENOM_{1..i}}(S,t1) \cap \\ & Top_{EENOM_{i..n}}(S,t2) = \emptyset \end{cases} \tag{6}$$

Yesterday's Weather is computed by counting the hits for all versions and dividing them by the total number of possible hits. Thus, we obtain the result as a percentage with values between 0% and 100%.

We formally define the *Yesterday's Weather* applied on n versions of a system history S given two threshold values t_1 and t_2 as in Equation 7.

$$(n > 2; t_1, t_2 \geq 1)$$
$$YW_{1..n}(S,t_1,t_2) = \frac{\sum_{i=2}^{n-1} YW_i(S,t_1,t_2)}{n-2} \tag{7}$$

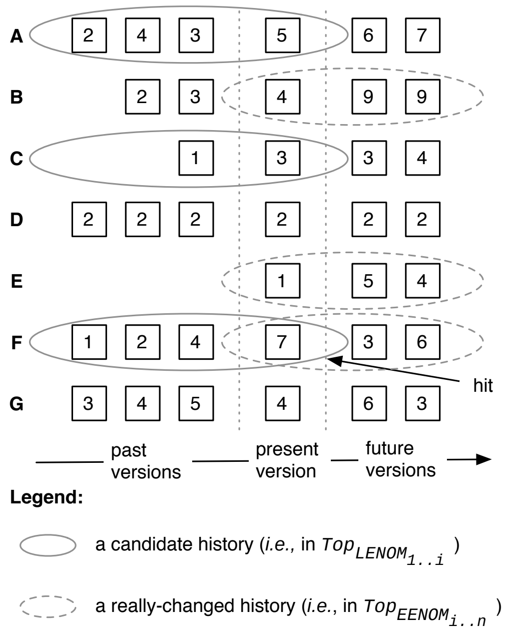


Figure 4. The detection of a *Yesterday's Weather* hit.

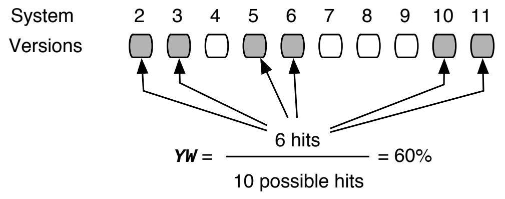


Figure 5. The computation of the overall *Yesterday's Weather*.

Example. In Figure 4 we present an example of how we check *Yesterday's Weather* with respect to a certain version. We display 6 versions of a system with 7 classes (A-G). We want to check *Yesterday's Weather* when considering the 4th version to be the present one. Therefore, the versions between 1 to 3 are the past versions, and the 5th and 6th are the future ones.

We also consider the dimensions of the *candidates* and the *really-changed* set to be 3, that is, we want to check the assumption that at least one of the top three classes which were changed in the recent history will also be among the top three most changed classes in the near future. We draw circles with a continuous line around the A, C and F classes to mark them as being the top three classes which changed the most in the recent history with respect to the 4th version. A, C and F are therefore, candidates for a *Yesterday's Weather* hit. While B changed recently, it is not a candidate because A, C and F changed more than B and in this case we only concentrate on the top three most changed classes. We marked with a dotted circle the classes which change the most in the next versions after the 4th one. We get a hit if the intersection between the continuous line circles and the dotted circles is not empty. In the presented example we get a hit because of class F.

In Figure 5 we show how we compute the overall *Yesterday's Weather* for a system history with 10 versions. The grayed figures show that we had a hit in that particular version, while the white ones show we did not. In the example we have 6 hits out of possible 10, making the value of *Yesterday's Weather* to be 60%.

3.3 *Yesterday's Weather* Interpretation

Suppose we have to analyze a 40 version system history, each version consisting on average of 400 classes and suppose we compute *YW*(S_1,*20,20*) and get a result of 10%. That means that the "climate" of the system is unpredictable with respect to the important changes. In such a "climate" it is not relevant to consider the latest changed classes to be important for the next version.

If, on the other hand, we compute the *YW*(S_2,*5,5*) and get a result of 80%, it means that during the analyzed period in 80% of the versions at least one of the 5 classes that changed the most in the recent past is among the 5 classes that change the most in the near future. Therefore, we have a great chance to find, among the first 5 classes which were heavily changed recently, at least one class which would be important (from a reverse engineering point of view) for the next version.

The value of *YW* depends on the dimensions of the sets we want to compare. For example, on each line in Table 1 we display different results we obtained, on the same history, depending on the sizes of the sets. For 40 versions of Jun, when we considered the *LENOM*$_{1..i}$ set size to be 5 and the *EENOM*$_{i..n}$ set size to be 5, the *YW* was 50% (*i.e.,* *YW*(Jun_{40},*5,5*) = 50%). When the *LENOM*$_{1..i}$ set size was 10 and the *EENOM*$_{i..n}$ set size was 10, the *YW* was 79% (*YW*(Jun_{40},*10,10*) = 79%).

In *YW(S, candidates, really-changed)* the dimensions of the *candidates* and *really-changed* sets represent thresholds that can be changed to reduce or enlarge the scope of the analysis. Thus, using higher thresholds increases the chance of a hit but also increases the scope, while by using lower tresholds we reduce the scope, but we also reduce the probability to have a hit. Both thresholds have specific interpretations:

1. The *candidates* set threshold represents the number of the classes which changed the most in the recent past. The lower this threshold is the more accurate the assumption is. For example, imagine that for one system we choose a *LENOM* threshold of 1 and an *EENOM* threshold of 5 and we get a *YW* value of 60% (*i.e.,* *YW*(S_1,*1,5*) =60%). For another similar system we get *YW*(S_2,*3,5*) =60%. It means that in the first system you have a 60% chance that the class identified as changing the most in the recent past will be among the 5 classes that change the most in the near future, while in the second system, we have to investigate three candidate classes to have a 60% chance to find one important class for the near future.

2. The size of the *really-changed* set is the second threshold and it shows how important – *i.e.,* how affected by changes – the candidates are. The lower this threshold is, the more important the candidates are. Suppose we have *YW*(S_1,*5,5*) = 60% in one system and *YW*(S_2,*5,1*) = 60% in another system. It means that in the first system we have a 60% chance that one candidate will be *among* the first 5 important classes in the next version, while in the second system we have a 60% chance that one of the candidates will be *the most* important class in the next version.

4 Validation - *Yesterday's Weather* Jun and CodeCrawler

Our approach is difficult to validate. A real validation would require repeatedly reverse engineering a real-system over a long period of time with access to internal development information. The validation would then be the comparison of the results we obtain with the reverse engineering results and with the next development facts. Instead, we compute our approach on two case studies, Jun and CodeCrawler, and discuss the the results from multiple perspectives.

Jun[3] is a 3D-graphics framework currently consisting of more than 700 classes written in Smalltalk. The project started in 1996 and is still under development. We have access to over 500 of its versions. 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 the first analyzed version there were 160 classes, in the last analyzed version there were 740 classes. In total there were 814 different classes which were present in the system over this part of its history, and there were 2397 methods added or removed.

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

History sample	*YW(3, 3)*	*YW(5, 5)*	*YW(10, 10)*
40 versions of Jun	40%	50%	79%
20 versions of Jun	39%	55%	77%
10 versions of Jun	37%	37%	87%
40 versions of CC	68%	92%	-
20 versions of CC	61%	94%	-
10 versions of CC	62%	100%	-

Table 1. *YW(3,3)*, *YW(5,5)* and *YW(10,10)* computed on different sets of versions of Jun and CodeCrawler.

CodeCrawler[4] is a language independent reverse engineering tool which combines metrics and software visualization. In the first analyzed version there were 92 classes and 591 methods, while in the last analyzed version there were 187 classes and 1392 methods. In the considered history, there were 298 different classes present in the system over the considered history and 1034 methods added or removed in subsequent versions.

Jun has been developed by a team of developers while CodeCrawler is a single developer project.

4.1 *Yesterday's Weather* in Jun and CodeCrawler

Table 1 presents the results of the *YW* for Jun and CodeCrawler for different number of versions while keeping the thresholds constant. High values (*e.g.,* 79% for Jun or more than 90% for CodeCrawler) denote a stable climate of the case studies: the changes either went slowly from one part to another of the system, or the changes were concentrated into some classes.

When we choose more distance between releases, we take into consideration the accumulation of changes between the releases. Therefore the candidate classes were not necessarily heavily changed in one version, but they were constantly changed over more versions.

Jun. When we doubled the thresholds when analyzing 40 versions of Jun, we obtained 29% more in the *YW* value. Moreover, when we doubled the thresholds when analyzing 10 versions, we more than doubled the *YW* value. These facts show that in Jun there were classes which were changed over a long period of time, but these changes are not identified when we analyze versions which are closer to each other.

To show the relevance of *YW* we display the class histories that provoked a hit when computing *YW(Jun,10,10)* for 40 versions of Jun (see Table 2). We focused on the size of the classes in terms of number of methods and determined that *YW* predicts changes in classes which are not necessarily big classes (*e.g.,* JunOpenGLPerspective). We marked with a "+" the 5 hit classes which were also in the top 10 of the biggest classes in the last version. There are 5 classes in the first 10 classes in terms of number of methods, which did not provoke a hit in the *YW*. Thus, we say that in Jun, a big class is not necessarily an important class in terms of future changes.

YW(Jun,10,10) hit class history	NOM
JunOpenGLDisplayModel +	150
JunWin32Interface +	104
JunBody +	85
JunOpenGL3dObject +	75
JunOpenGL3dObject_class +	71
JunOpenGL3dNurbsSurface	55
JunLoop	55
Jun3dLine	51
JunOpenGLProjection	48
JunUNION	47
JunOpenGL3dCompoundObject	41
JunPolygon	34
JunBody_class	31
JunVertex	30
JunOpenGL3dVertexesObject	23
JunOpenGL3dCompoundObject_class	21
JunOpenGL3dVertex	19
JunUNION_class	19
JunOpenGL3dPolygon	15
JunOpenGLPerspective	12
JunOpenGLTestController *	9
JunOpenGLTestView *	1

Table 2. The class histories that provoked a hit in Jun when computing *YW(Jun,10,10)* and their number of methods in their last version. (Legend: the "*" classes were not present in the system's last version; the "+" classes were in the top 10 of number of methods in the last version).

CodeCrawler. CodeCrawler is a project developed mainly by one developer, and as such can be considered a system with a focused and guided development with little external factors. This assumption is backed up by the data which reveals very high *YW* values for low thresholds, resulting in a "stable climate" of the system. Note that CodeCrawler is much smaller than Jun, the thresholds must thus be seen as relatively lax.

Table 3 displays, using the same notation as in Table 2, the class histories that provoked a hit in *YW(5,5)*. As in the case of Jun, the hits were not necessarily provoked by big classes and not all big classes provoked a hit. This shows that in CodeCrawler there is not always a relation-

[4]See http://www.iam.unibe.ch/∼scg/Research/CodeCrawler/.

YW(CC,5,5) hit class history	NOM
CCDrawing +	123
CCAbstractGraph +	99
CCGraph +	69
CCNode +	47
CodeCrawler +	42
CCConstants_class +	39
CCEdge *	36
CCControlPanel	29
CCGroupNodePlugIn	25
CCModelSelector *	24
CCRepositorySubcanvas	17
CodeCrawler_class	15
CCService	0

Table 3. The class histories that provoked a hit in CodeCrawler when applying *YW(CC,5,5)* and their number of methods in their last version. (Legend: the "*" classes were not present in the system last version; the "+" classes were in the top 10 of number of methods in the last version).

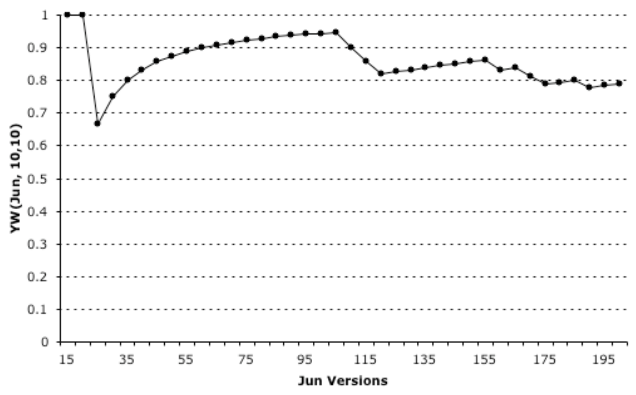


Figure 6. The values of *YW(Jun,10,10)* over 40 versions of Jun. The diagram reveals phases in the which the predictability increases and during which changes are more focused (*e.g.,* the first part of the history) and phases in which the predictability decreases and changes are more unfocused (*e.g.,* the second part of the history).

ship between changes and size. Therefore, identifying the big classes from the last version, is not necessarily a good indicator for detecting classes which are important, in terms of change, for the next versions.

4.2 The Evolution of *Yesterday's Weather* in Jun

In Figure 6 we represent a chart which shows Jun's evolution of *Yesterday's Weather* over time. The points in the chart show the value of *Yesterday's Weather* until that version: in version 15 *Yesterday's Weather* is 100%, drops in version 25, grows again until version 100 and then finally has an oscillating descending shape.

Based on this view we can detect phases in the evolution where the changes were focused and followed by other changes in the same part of the system (the ascending trends in the graph) and phases where the changes were rather unfocused (the descending trends in the graph). In the first half of the analyzed versions, in 90% of the cases at least one class which was in the top 10 of the most changed classes in the last period was also in the top 10 of the most changed classes in the next version. In the last 20 versions that we analyzed, the probability drops. Therefore, in the first half of the analyzed period the development was more continuous and focused than in the second half.

5 Analyzing the *Yesterday's Weather* Approach

In this section we explain the impact of the decisions we took when defining the *YW* measurement.

About the Impact of the Weighting Function. The *LENOM* measure weighs each change using the function 2^{i-k} (see Equation 3 and Figure 3). This function actually acts like a window over the complete history of the changes by considering as relevant only the last four versions. This window is important as it lowers the impact of early development. For example, if a big class was developed in the early versions but now suffers only bug-fixes, it will not be selected as a candidate for future important changes. Increasing the window length favors the candidacy of the large classes in the system, even if they are not changing anymore, and reduces the relevancy of the prediction. Note that although the value of *LENOM* takes into account only the last four versions, the *YW* measurement is computed over the complete history.

About the Impact of the *At Least* Condition. With the current *YW* assumption we consider to have a hit if we have *at least one* class which was heavily changed recently and which also gets changed a lot in the next versions. If we have *YW(S,10,10)* = 60%, we do not know if the assumption held for 10 out of the 10 candidate class histories or just for one of them. *YW* gives relevant results in two cases:

1. *High value of* YW *when considering low thresholds.*

Low thresholds mean low scope (both of candidates or of the importance of the *really-changed* entities), and if for such low thresholds we obtain a high *YW* value, we can characterize the changes as being continuous, and therefore it is relevant to look at the most recently changed classes to detect one which will probably undergo an important change during the near future, *e.g.*, the next versions.

2. *Low value of* YW *when considering high thresholds.* When obtaining low *YW* values for high thresholds, we can characterize the changes as being discontinuous, and therefore looking at the most recently changed classes is not necessarily relevant for the future changes in the system.

A possible alternative would be to compute an average of the number of class histories that matched the YW_i assumption. The result of this average would complement the *YW* value, by showing its overall accuracy.

About the Impact of the Release Period. Another variation point when computing *Yesterday's Weather* is the release period. If we consider the release period of one week, we focus the analysis to immediate changes. If, on the other hand, we consider the release period of half a year, we emphasize the size of the changes that accumulate in the class histories.

Example. Suppose that when we consider the release period of a big system of one week we obtain *YW(S,5,5)* = 60% and when we consider the release period of half a year we obtain *YW(S,5,5)* = 20%. It means that from one week to another the development is quite focused, and the bigger parts of the system tend to stabilize over a long period of time, thus leading to apparently unexpected changes, *e.g.*, bug-fixing, patching, small functionality increase all over the system.

The variation of *YW* allows one to fine-tune the information. It is the combination of short and focused releases and the fact that *YW* drops that allows one to conclude that the system stabilizes. Note that, by considering longer release periods, the additions and removals of methods from the same class between consecutive releases will not show in the history measurements.

About the Impact of the Number of Versions. The number of versions makes another variation point when computing *YW*. Increasing or decreasing the number of versions affects the overall *YW*, but has little effect on the value of individual YW_i. The longer the considered history, the less important is a hit/non-hit. By increasing the number of versions while keeping the same period between versions, we let the early changes affect the overall *YW*. Therefore, by increasing the number of versions we obtain a long-term trend, while by decreasing the number of versions we concentrate on the short-term trend.

6 Related Work

Metrics have traditionally been used to deal with the problem of analyzing the history of software systems.

Lehmann used them starting from the 1970's when he analyzed the evolution of the IBM OS/360 system [14]. Lehmann, Perry and Ramil explored the implication of the evolution metrics on software maintenance [15] [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.* [8] 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 [1].

Gold and Mohan defined a framework to understand the conceptual changes in an evolving system [9]. Based on measuring the detected concepts, they could differentiate between different maintenance activities.

Visualization proved to be an effective technique to analyze the history of software systems.

Lanza's Evolution Matrix [12] displays the system's history in a matrix in which each row is the history of a class (see a simplified version in Figure 2). 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.

Rysselberghe and Demeyer used a simple visualization based on information in version control systems to provide an overview of the evolution of systems [18].

Grosser, Sahraoui and Valtchev applied Case-Based Reasoning on the history of object-oriented system as a solution to a complementary problem to ours: to predict the preservation of the class interfaces [10]. They also considered the interfaces of a class to be the relevant indicator of the stability of a class. Sahraoui *et al.* employed machine learning combined with a fuzzy approach to understand the stability of the class interfaces [17].

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

Gall *et al.* [7] 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 level making the results finer grained. Zimmerman *et al.* placed their analysis at the level of entities in the meta model [19]. Their focus was to provide a mechanism to warn developers that: "Programmers who changed these functions also changed.…". Their approach differs from ours because they only look at syntactic changes, while we identify changes based on the semantics of the changes.

Fischer *et al.* [6] analyzed the evolution of systems in relation with bug reports to track the hidden dependencies between features. Demeyer *et al.* [3] 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.

7 Conclusions and Future Work

When starting a reverse engineering effort, knowing where to start is the key problem. When only the code of the application is available, the history of a software system could be of help. However, analyzing the history is difficult because of the interpretation of large quantities of complex data. In this paper we presented our approach of summarizing the history by defining history measurements.

We used the term *Yesterday's Weather* to depict the retrospective empirical observation that at least one of the classes which were heavily changed in the last period will also be among the most changed classes in the near future. We computed it on two case studies and showed how it can summarize the changes in the history of a system. We use the approach to pinpoint classes in the latest versions which would make good candidates for a first step in reverse engineering. We looked at how the big changes are related with the size of the classes, and validated our approach by showing that big changes can occur in classes which are not big in terms of size (*i.e.,* number of methods). Thus, our approach is useful to reveal candidates for reengineering which are otherwise undetectable if we only analyze the size of the classes of the system's last version.

In the future, we would like to correlate our approach with information from outside the source code. Conway's law [2] states that "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." Therefore, when reverse engineering, we should take the auxiliary development information into account. For example, we could correlate the shape of the evolution of *Yesterday's Weather* with the changes in the team or with the changes in the development process.

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] 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] M. E. Conway. How do committees invent ? *Datamation*, 14(4):28–31, Apr. 1968.

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

[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, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[6] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99, Nov. 2003.

[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] N. Gold and A. Mohan. A framework for understanding conceptual changes in evolving source code. In *Proceedings of International Conference on Software Maintenance 2003 (ICSM 2003)*, pages 432–439, Sept. 2003.

[10] D. Grosser, H. A. Sahraoui, and P. Valtchev. Predicting software stability using case-based reasoning. In *Proceedings of the 17th IEEE International Conference on Automated Software Engienering (ASE '02)*, pages 295–298, 2002.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings IWPSE2001 (4th International Workshop on Principles of Software Evolution)*, pages 83–86, 2002.

[17] H. A. Sahraoui, M. Boukadoum, H. Lounis, and F. Ethève. Predicting class libraries interface evolution: an investigation into machine learning approaches. In *Proceedings of 7th Asia-Pacific Software Engineering Conference*, 2000.

[18] 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.

[19] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, 2004.

A Implementation - Van and Moose

Van is our history analysis tool which is based on the *Moose* [5] reengineering platform. Van is an implementation of the *HisMo* history meta model which is an extension of the *FAMIX* [4] meta model. The FAMIX and the HisMo metamodels are both language independent.

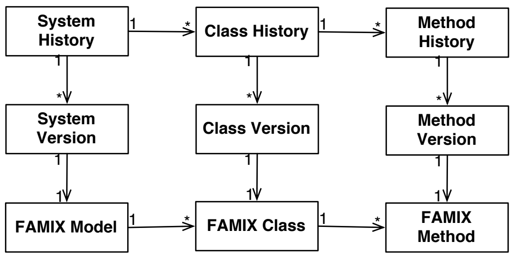


Figure 7. The HisMo meta model is an extension of the FAMIX meta model (This a reduced schema of the meta model).

In Figure 7 we schematically present the relationship between HisMo and FAMIX. HisMo recognizes the history as being a first-class entity which is formed by multiple versions, each version having a one-to-one relationship with a FAMIX entity.

B Weighted Evolution of a Version Property (*WE*)

We define a generic evolutionary measurement *WE* (Equation 8) which takes as arguments the history (H), a weight function (w) and a property P. *WE* is the sum of the changes in a property P in subsequent versions, where each change is given a weight according to the version i in which the change occurred compared to the analyzed history (*i.e.*, from version j to version k, out of n total versions).

$$(1 \leqslant j < k \leqslant n)$$
$$WE_{j..k}(H, P, w(i,j,k,n)) = \sum_{i=k}^{k} |P_i(H) - P_{i-1}| w(i,j,k,n) \tag{8}$$

ENOM, *LENOM* and *EENOM* are instances of *WE*. *ENOM* is an instance of *WE* where the weight function is 1, and where $P = NOM$.

$$(1 \leqslant j < k \leqslant n)$$
$$ENOM_{j..k}(C) = WE_{j..k}(C, NOM, w(i,j,k,n) = 1) \tag{9}$$

The *LENOM* measurement is an instance of the *WE* measurement where $w(i,j,k,n) = 2^{i-k}$.

$$(1 \leqslant j < k \leqslant n)$$
$$LENOM_{j..k}(C) = WE_{j..k}(C, NOM, w(i,j,k,n) = 2^{i-k}) \tag{10}$$

The *EENOM* measurement is an instance of the *WE* measurement where $w(i,j,k,n) = 2^{k-i+1}$.

$$(1 \leqslant j < k \leqslant n)$$
$$EENOM_{j..k}(C) = WE_{j..k}(C, NOM, w(i,j,k,n) = 2^{k-i+1}) \tag{11}$$