Exploring the Evolution of Software Quality with Animated Visualization [0] *

Guillaume Langelier Houari Sahraoui Pierre Poulin
Université de Montréal

Abstract

Assessing software quality and understanding how events in its evolution have lead to anomalies are two important steps toward reducing costs in software maintenance. Unfortunately, evaluation of large quantities of code over several versions is a task too time-consuming, if not overwhelming, to be applicable in general.

To address this problem, we designed a visualization framework as a semi-automatic approach to quickly investigate programs composed of thousands of classes, over dozens of versions. Programs and their associated quality characteristics for each version are graphically represented and displayed independently. Real-time navigation and animation between these representations recreate visual coherences often associated with coherences intrinsic to subsequent software versions. Exploiting such coherences can reduce cognitive gaps between the different views of software, and allows human experts to use their visual capacity and intuition to efficiently investigate and understand various quality aspects of software evolution.

To illustrate the interest of our framework, we report our results on two case studies.

1. Introduction

Over the years, development of software has gone through many transformations. At first, scientists built small software for their own needs with mainly functional goals in mind. Today's software aims at a larger public, is often more complex, and involves more programmers. Having many people from many locations over a more significant period of time can introduce communication problems and induce extra cost in the project. It becomes therefore important to quickly understand code from others because original developers are not always available. This development model can lead to quality flaws that greatly impact a project and that must be corrected quickly to reduce their costs [10]. In this context, evolution plays a crucial role in understanding quality aspects of software. Evolution can explain the current state of a program and can be used to adjust the development process to better respond to quality expectations.

Therefore it has become necessary to analyze the complex programs present nowadays, both in terms of comprehension and quality. However, the analysis of these programs generates a lot of raw data that must be interpreted in order to extract valuable information concerning its quality. The presence of multiple versions in the analysis of software evolution increases this amount of raw data by adding the dimension of time into the equation. It is hardly possible for an expert to assess this amount of data when the object of study grows over the size of toy programs. This analysis task is also difficult to automate because the exact factors impacting quality and maintenance costs are still unknown or blurry, and general rules are difficult to derive automatically. While machine learning approaches do not require rules from theory, they rely on data assumptions and large learning sets not easily available for software evolution.

Our solution proposes to use visualization as a semi-automatic approach to analyze the quality of programs over many versions. Visualization re-arranges and displays data in a more convenient way for an expert who can manually extract information from these images and make decisions using his knowledge and judgement. We use structural metrics and version control information that we map on graphical entities. The visualization consists in 3D boxes laid out over a 2D plane according to the program architecture, with an algorithm inspired from *Treemap* [15]. This basic visualization is augmented by a representation of evolution.

A number of approaches visually represent software evolution within a single frame for one system or one aspect of a system and all its versions [6, 17, 18, 21, 27, 28]. Other approaches present different snapshots of a system and place them side-by-side for analysis [7, 11, 25, 29, 30]. The first strategy requires aggregating data before presenting it; the other strategy uses a lot of space and prevents the analysis of larger systems. Our solution to these problems uses animation and the metaphor of time . Versions are presented one after the other using animation and navigation features. Placing every item adequately increases coherence

[0]To better appreciate our examples, please view the associated video on *www-etud.iro.umontreal.ca/∼langelig/vlhcc08.wmv*

between represented versions and therefore reduces cognitive gaps when going from one version to another. We use a similar strategy to switch from a structural metric view to a version control view, and reduce the cognitive effort while displaying more information.

The rest of the paper is organized as follows. Section 2 summarizes work related to the present research. Section 3 introduces the different steps of our approach. Section 4 explains the data to be visualized, while Section 5 is dedicated to the relation between graphical coherence, software coherence, and their application in the animation of software evolution. Section 6 describes the techniques used to represent software evolution using visualization. Finally, Section 7 discusses our framework and its applicability, and Section 8 concludes with some remarks.

2. Previous Work

A frequent representation of evolution uses a static frame and place entities in a matrix-like configuration. Usually, rows encode entities and columns versions in chronological order. A matrix cell contains a glyph representing the state of given entity at a given time. For instance, Wu *et al.* [28] represent the evolution of programs with spectrographs. Rows represent files, columns represent versions, and at a cell, a pixel is colored according to the time elapsed since the last modification. Lommerse *et al.* [18] use a similar approach, but they align multiple two-dimensional views one below the other to display multiple metrics for a given time frame (column). Voinea and Telea [27] use the same idea but represent the evolution of files with bands similar to time lines. D'Ambros and Lanza [6] compare bugs and cvs activities of a given system by means of *discrete time figures*, which resemble time lines. They also combine these views in a hierarchical system of files. Lanza and Ducasse [17] draw at each row-column cell a rectangle with its height, width, and color attached to class metrics. They also developed a metaphor based on astronomy to interpret observed phenomena. Finally, Mesnage and Lanza [21] draw 3D representations in the same matrix-like layout to display more metrics simultaneously.

Other work considers different representations for evolution, but still uses static visualization. D'Ambros *et al.* [7] visualize cvs entries and especially contributions of each programmer using figures inspired from fractals. Fischer and Gall [11] study co-changes in files and analyze them in the perspective of the structural information of these files. They use an energy-based graph to represent logical coupling and a time-line representation to compare files within a same time frame. Pinzger *et al.* [25] visualize the evolution of metrics in programs with Kiviat diagrams representing multiple module metrics. Evolution is represented with layers on the Kiviat diagram, and Kiviat diagrams are connected together to represent coupling between modules. Lingu *et al.* [19] present a tool with several simple visualizations to analyze super repositories. Super repositories contain several projects handled by a single company or linked together semantically. Views include plot diagrams, nodes, links, and timelines. Wu *et al.* [29] use ordered Treemap views to represent cvs information. The size and color of each node are mapped to metrics related to the size of changes and the date of the last commit. Links represent relationships. With their tool *cv3D* [30], Xie *et al.* use *polycylinders* to display metrics at different levels : system, class, method, and line. They use a single view to aggregate the whole system or to display modifications between versions. They also use multiple pictures juxtaposed to compare one element at different versions.

Evolution has also been represented with animation and time. D'Ambros and Lanza [5] use a visualization technique called the *evolution radar* to track logical coupling of modules in a system. Diagrams from different time windows are computed on demand, providing a multi-frame visualization. Collberg *et al.* [4] present a visualization based on 3D graphs to represent the hierarchy, the call graph, or the control flow graph of a program. The layout is determined with an attraction-repulsion model with the same nodes from different versions being attached together to reduce movement. Beyer and Hassan [1] present a solution more closely related to ours. It introduces a storyboard concept which consists in a series of frames representing different periods of time, and therefore, their approach uses animation. They also use an energy-based layout and give information through clustering, colors, and links. However, they target cvs information and logical coupling, while we consider also structural metrics. They filter out many nodes from their visualization to reduce occlusion and visual saturation which is not required in our approach.

Amongst these techniques, only the last two [1, 4] use animation to represent evolution. All the other techniques use single images, which are more or less based on graphs or on the matrix principle. These techniques impose a limitation on the number of elements visualized, or on the number of attributes associated with each element.

Even though little research is based on animation to represent software, some applications of this strategy have been used in the more general area of information visualization. Nguyen and Huang [22] present a technique where the node of interest in a 2D graph is exchanged with one of its children with animated movements towards the root. Bladh *et al.* [2] put emphasis on parts of a Treemap, but it effectively acts like a zoom. They do not rearrange or resize the zoomed in parts. Fekete and Plaisant [9] modify the values represented by a Treemap, therefore changing the size of container rectangles present in the visualization. They animate movement and size modifications in two phases to reduce saturation of the human visual system. However, they do not consider the addition or removal of items, which is always the case in software evolution. North [23] also presents an approach for incremental graph drawing that optimizes the stability of node locations. Other approaches use similar principles to incrementally display graphs [12, 24].

3. Approach Overview

The analysis process with our framework can be summarized in five steps. First, metrics and relationships are extracted from ByteCode of Java Programs and version control repositories. The second step automatically creates representations of the extracted data and displays them in our 3D environment. In the third step, experts evaluate the program by performing investigation actions such as navigating through the environment, navigating in time through the different versions, and switching from a structural view to a version control view in order to compare and understand the different elements presented. A fourth step may be necessary to verify observations made in more details. Indeed, our tool is designed to analyze software at the granularity level of classes, and therefore some analysis results must be confirmed with external tools or simply by directly inspecting the code itself. The fifth and final step applies conclusions from the visualization process in order to correct the observed flaws, starts to add new features in a newly understood piece of software, or makes decisions concerning the process of future software projects. Our tool enables quick assessing of programs with thousands of classes modified over dozens of versions.

4. Metrics and Relationships

Various types of data are useful to understand and to evaluate the quality of software. We present in this section three such categories. They are mapped to graphical characteristics in order to be visually interpreted by experts.

Structural metrics represent code well because they simultaneously encapsulate and summarize code, while providing information about its quality. For example, classes with large coupling and high complexity are recognized to lead to poorly maintainable software. While our system can extract several metrics, the examples described in this paper use mainly CBO (Coupling Between Object) for coupling, WMC (Weighted Methods per Class) for complexity and size, and LCOM5 (Lack of Cohesion in Method) for cohesion [3]. Metric values are extracted from ByteCode of Java programs using a homemade tool [14]. Information on different versions is stored independently for each pair (class, version) and is only merged back during visualization.

Our framework represents **UML relationships**. For each class, associations, invocations, interface implementations, and generalization/specialization relationships are reverse-engineered with another tool [13]. These relationships are stored individually for each version.

Our framework also works with **version control metrics** (*cvs*). This information, extracted from *cvs* log files, indicates the author of a modification, the owner[1] of a file (class/interface), the size of a modification, and the number of versions since the last modifications. These metrics are interesting because they represent information about what happened between two versions of a system, instead of being attached to a version in particular. Version control metrics give raw information about the development of the software product, but do not explicitly describe important process decisions that may explain design flaws. However, some information can be abstracted from them to greatly help explaining the quality of the code. Examples of abstracted informations include author changes in a package, phases of refactoring where many classes are modified, large modifications, creation of multiple classes, introduction of new authors, code constantly or never modified, etc.

Most current visualization tools present either structural metrics [4, 17, 25] or version control information [6, 7, 18, 21, 27, 29]. In order to gain a more complete understanding and analysis of the quality of a system as a whole, it is preferable to have easy access to both types of metrics within a single unified tool. However, this is not as simple a task as merging two tools together. Informations from control version and structural metrics are often not aligned together. The former ones are extracted according to individual commits or dates, while the latter ones are usually computed on major releases. In order to display both types of information in a coherent way (see Section 5), it is important to synchronize them so they correspond to each other on a timeline. To achieve this, we convert version control events into metrics to represent what happens between two versions. By keeping in parallel both contexts, the cognitive gap between views is reduced and the analysis of these metrics in conjunction is facilitated.

5. Graphical Coherence Applied to Software Coherence

Graphical temporal coherence resides in the similarity between two images following each other in time, such as in an animated film. With each image drawn similar to the previous one, slight differences will reflect small progression in the scene. Putting all-together all images results in a coherent sequence serving the purpose of a complex story. Because large portions of the images remain unchanged, we do not have to reconstruct their full structure each time. We only need to assess the differences. Since human vision is

[1]The author who committed the larger portion of lines for this file.

inherently attracted by moving elements, the task of following a coherent sequence of images does not require significant cognitive efforts.

Coherence also exists between versions in the evolution of a program. New versions are built on top of previous ones, and therefore modified and new elements affect only a portion of an otherwise unaltered large common portion. Therefore to analyze software evolution, it is more important to concentrate on the modifications rather than to comprehensively analyze each individual version. Although static, *i.e.*, unaltered parts, represent the context that helps better understand a modification, we do not need to re-evaluate this context at each transition between versions.

Hence in a visualization tool exploring the quality of software evolution, it is natural to combine coherence between software versions and coherence of graphical animation, in order to help experts extract information more efficiently. The combination is however not straightforward, as will be discussed in detail in Section 6.2.

6. Visual Representation

6.1 Single Version Representation

Because our multiple version visualization is built on-top of our single version visualization, we first briefly introduce its principles.

To understand our single version visualization, we simply describe the representation of an individual class and how to position it in the environment. More details are available in [16].

6.1.1 Single Class Representation

Classes are represented as 3D boxes arranged over a 2D plane. The 3D box was chosen because it is both efficient to render on graphics hardware and easy to understand by humans. Since we are working with Java Programs, interfaces are differentiated from classes by using cylinders. A set of graphical characteristics are mapped to metrics: a color scale from blue to red or a set of discrete colors to represent nominal data, the box's height, and the box's rotation around the up axis (twist). These characteristics do not introduce important perception bias on each others, but it would be difficult to add many more characteristics without impacting on the clarity of existing ones. Metrics are associated by a linear mapping between their values and corresponding graphical characteristics. The maximum value for each graphical characteristic corresponds to a *practical* maximum value for the metric. Values higher than this threshold are mapped at this maximum value. We generally use the following associations for metrics and graphical characteristics: color and coupling, twist and cohesion, size and height. These associations help experts in their analyses because of related metaphors [20]. Red often means danger and a high coupling is recognized as a problem in the software engineering literature, height relates naturally to size, and finally a cohesive class can be seen as going straight to a goal and a non-cohesive class as going in all directions, so it appears twisted. We also map size of modifications to height, time since the last modification to twist, and discrete colors to authors in a second view on version control information.

6.1.2 Layout

The layout of entities (classes/interfaces) follows their full name path and the package hierarchy it represents. This adds new information without extra dimensions. The goal is to represent this hierarchy while using space as efficiently as possible. To do so we use a variant of the Treemap [15] algorithm, which is a space filling visualization representing trees and using a starting rectangle to subdivide it recursively. However, this technique must be adapted because as seen in Section 6.1.1, each box needs a given amount of space on the plane, while the original Treemap is based on continuous values. Our solution enlarges the original rectangle when necessary in order to fit the discrete elements. It is not a problem since we are in a 3D environment and there are no strict constraints on the resulting rectangle.

6.1.3 Navigation

Users can navigate in our 3D environment according to the information they are looking for. The camera rotates on an hemisphere, can smoothly move the center of the hemisphere, as well as zoom in and out. The camera is always pointing toward the layout plane to prevent confusion. Users can also directly access the metrics numerical values or the code itself by clicking on a given class. Another mode allows users to click on a class to fetch information about its relationships. Instead of drawing links between entities, we simply reduce the saturation of classes not concerned by relationships without altering the others.

6.2 Multiple Versions

Our representation for multiple versions is based on our representation for a single version. We display each version one after the other, with different strategies to increase visual coherence. The navigation in time (forward and backward) between versions is controled by the user.

6.2.1 Animation of a Single Class

During the animation of an individual class, we always consider only two of its states: at version v_i and version v_{i+1}.

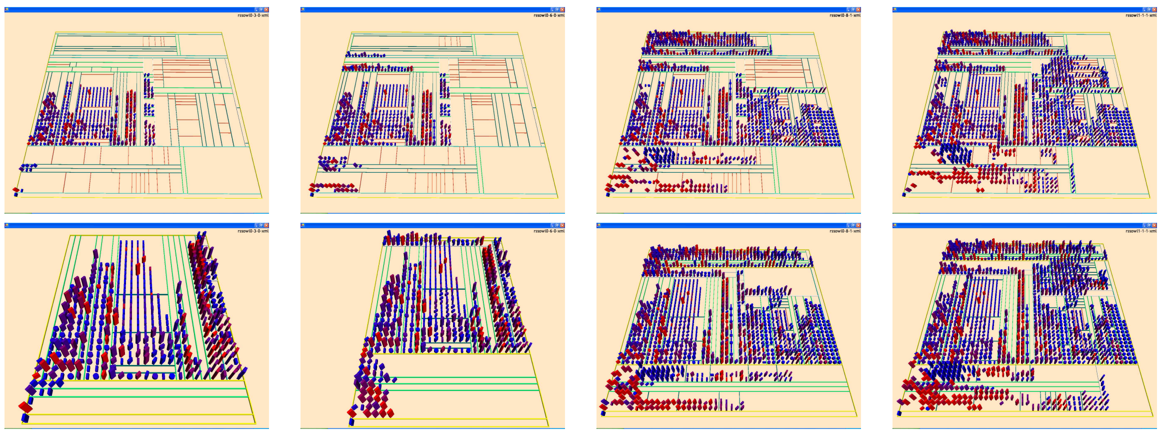

Figure 1. Four frames of RSSOwl using (top) the static position layout animation and (bottom) the relative position layout animation.

At the beginning of the animation, the class representation at v_i is displayed. A linear interpolation with the use of in-between frames is then displayed to reach the state of its next version. All three graphical characteristics are simultaneously incremented. When position must be modified, classes are translated in a first phase of animation and their characteristics are changed in a second phase. These animations last only a second, but attract the visual attention of the expert while being efficient for the understanding of program modifications [26]. These smooth transitions between representations of two subsequent versions are one of the ways we use to achieve more coherence in the system.

6.2.2 Animation of the Layout

In order to achieve coherence during layout animation, we need to reduce unnecessary movements of classes as much as possible. Classes that are not moving or moving less are much easier to track. Therefore, it is not a good idea to use the Treemap layout generated for each independent version and interpolate between consecutive versions. Because the Treemap algorithm is recursive and not iterative, the addition of only a few classes can dramatically change the layout. The two next paragraphs present alternatives that increase coherence.

Static Position Animation. In this layout animation approach, all classes remain at the same position during the visualization of all versions. The position of each entity is computed for all versions at the beginning of the visualization. To do so, all classes that ever existed in the system and their package hierarchy are merged in a virtual tree for which we compute a Treemap layout. Classes in this animation approach are easily followed because they remain static. However this creates holes where classes were deleted in a previous version or where classes are not yet created. This can result in an important loss of space for early versions where only few classes are present. Figure 1 shows four frames of the evolution of RSSOwl using this layout animation.

Relative Position Animation. Relative Position Animation is built on top of Static Position Animation, but uses a post-processing step to reduce the space lost in the early versions where only a few classes are present. We simply compute the Treemap algorithm for the virtual Tree as mentioned above, and then take each version individually and try to shrink them in order to use less space, but with the added constraint that all classes must keep their relative position. This means that if class A was placed to the left of class B, class B will never be able to go to the left of class A. In order to achieve this, we simply try to move classes left or down, and see if constraints are violated. If it is not the case, we move it to the new position. We continue the process until no classes can be moved. Using this technique, classes usually move in groups and are easier to track visually than in the independently computed Treemap. The space reduction is significant for very large programs, or in presence of major renamings where large portions of the system move in the visualization. Figure 1 shows again RSSOWL, but using this layout animation technique.

6.3 Switching Context

Our switching context feature is useful to represent more metrics without having to add more visual characteristics. Switching context transfers the visualization from one set of metrics to another, while navigating in the visualization. Any set of metrics can be mapped to both contexts but usually, structural metrics are used in the first context and version control metrics in the second context. With the click of a button, the mapping changes instantly, allowing rapid flips. Between the two images associated with the switch, only graphical characteristics are modified; the point of view and the current version remain the same. This switching is more coherent because users do not have to re-evaluate from a new point of view, but only fetch new information while in the midst of their navigation process. Figure 2 shows two contexts from the same view of the program Freemind.

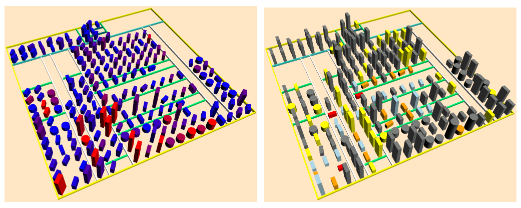

Figure 2. Two contexts from the same view angle in the program Freemind.

7. Applications and Evaluation

Since our approach is semi-automatic, the examples presented in this section are the result of explorations. The whole investigation took a short amount of time (a few hours to explore more than 15 systems) and required some movements backward and forward in time to be executed. However, considering the size of the explored systems, we think this effort is reasonable compared to the results obtained. These examples were found by experimented users, although novice users were able to achieve similar results to find phenomena on static visualization [8, 16]. We believe that this learning curve should translate well to our multiple version environment.

7.1 Applications

Exploration. Free exploration simply maps interesting metrics and lets the user navigate freely in the code and its versions. The visual aspect of our approach is well suited to attract the eye on suspicious events, anomalies, and recurrent patterns. It can also help to give an idea of the general quality of a system.

Verification of Automatic analysis. In order to calibrate or evaluate automatic analysis, it is necessary to verify its output against data that are verified and accurate. Manually reviewing results of automatic analysis can be tedious, if not impossible. Our semi-automatic approach of visualization can reduce the effort required by this manual inspection, especially for the quick verification of false positives.

Study of Evolution Patterns. Some evolution patterns are known to be suspicious, *e.g.*, constantly growing classes, quick birth and death of classes, and explosions in complexity in a short span of time. These patterns are difficult to define precisely and their evolution can differ from the expected behavior. Therefore their detection is difficult to automate. With our framework, an expert can recognize them easily and deal with such differences.

Context and Evolution of Known Anomalies. Several researches have already exposed anomalies from different programs. These anomalies are difficult to understand and correct because they represent only symptoms. Our framework allows following problematic classes over their evolution and their program context to give more information on reasons why they became problematic.

Applications mentioned above are only examples of the capabilities of our framework. In fact, it presents raw data to users so they can freely interpret them. Mappings are fully customizable and determined dynamically so multiple sources of different data can be used to create visualizations not restrained to software. Our framework is flexible and ranges from careful inspection of characteristics from each entity in a system, to an overview of several systems in a few minutes.

7.2 Case Study of Evolution Patterns

In order to evaluate our framework on real software, we have investigated Azureus, a well-known open source peer-to-peer program, for evolution phenomena and particularities. We have studied four main versions of Azureus, which contain over 2500 classes. Table 1 shows our findings using our framework during an exploration of about 90 minutes.

Figure 3 shows an example taken from the case study where two classes (UpdateChecker and UpdaterUpdateChecker) are shrinking simultaneously even though they are in different packages. After more careful observation, we see that these two classes are highly inter-related and that the importance attached to their behavior in the code has reduced between these two versions.

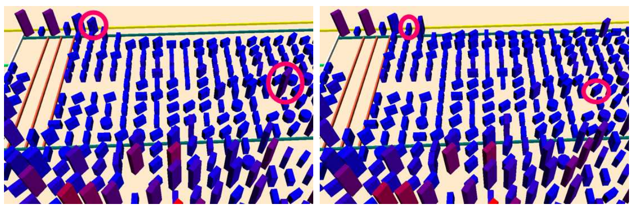

Figure 3. An example of rapidly shrinking classes found in Azureus.

Figure 4 shows an example where a class disappears while its neighbor class becomes suddenly complex. In

Type	Nb.	Concerned Classes
Responsibility Overload	3	TorrentUtils, FileUtil, ThreadPool
Class Renaming	* [2]	LGlogger → Logger, LGloggerImpl → LoggerImpl, BTProtocolMessage → BTMessage, devices → services
Responsibility Transfer	1	TRTrackerUtilsImpl → TRTrackerUtils
Rapidly Growing Classes	4	DiskManagerImpl, NetworkManager, IncomingMessageQueue, ConfigurationDefaults
Rapidly Shrinking Classes	2	UpdateChecker, UpdaterUpdateChecker
Classes → Interfaces	1	BufferedTableItem

Table 1. Examples from the case study on evolution patterns concerning Azureus.

fact, between versions 2.4 and 2.5, developers decided they no longer needed a class and its implementation in separate files, and transferred the code to one file.

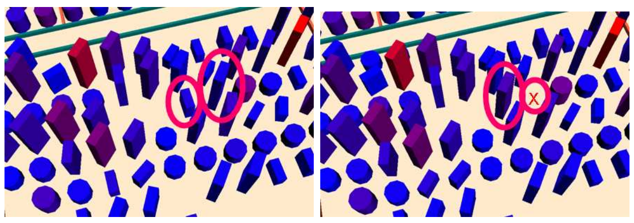

Figure 4. An example of responsibility transfer found in Azureus.

7.3 Case Study of Context and Evolution of Known Anomalies

Studying the evolution of known anomalies helps to correct them in later phases and gives some ideas on patterns that should be avoided in future development. We studied more than five systems containing a total of almost 80 anomalies. Observed patterns show that some anomalies appear as problematic, while others suffer from a gradual degradation. Other patterns showed an up-and-down cycle toward the anomaly. We will now present two specific observations among these. The class Controller in Freemind presented in Figure 5 can be considered a God Class. While observing its metrics, it is obvious that this class follows a responsibility overload pattern because it is constantly growing to become a problem. There are two authors attached to this file and both have participated in this continuous growth in terms of coupling and complexity. The second author has modified the class many times upon his arrival, without any observable benefits in coupling or complexity. Secondly, in Lucene, the class IndexReader was considered a potential case of Shotgun Surgery according a study [8]. The class is in fact constantly growing in coupling, even though it starts very high. Moreover its complexity follows and up-and-down pattern, but in the long run, the class is always gaining in complexity, as if developers were failing to keep good practices after a refactoring.

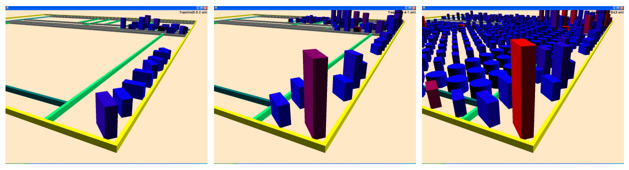

Figure 5. A controller class grows gradually out of proportion in Freemind.

8. Conclusion

We have presented in this paper a framework able of visualizing software evolution and its quality aspects for systems of thousands of classes over dozens of versions. To do so we use two parallel views to represent both structural and control version metrics. Classes represented as 3D boxes are arranged according to their full path containment hierarchy. Animations between static frames represent several versions, and in-between frames and evolution-specific layouts help transform the coherence already present between the different versions of a program into coherence between their graphical representations. We have presented a number of applications for our tool and demonstrated its usefulness in two case studies.

Other tools present structural metrics and version control information separately, but we consider that their unification is both useful and a new challenge. The use of animation and coherence is also important because it allows the verification of large programs that are difficult to analyze in a single image. Our short animations and careful use of navigation features reduces the cognitive effort required to analyze evolution and enables rapid overview of systems.

As future work, we need to conduct a comparative experiment for the usefulness of our tool on precise software

[2] We found many renaming occurrences between versions 2.2 and 2.4, but only list a few.

engineering tasks. We would like to add even more information in our framework by integrating a semantic zoom in the visualization in order to explore different granularity levels. We want to develop a new layout responding well to expansion to prevent major modifications of the Treemap layout when adding classes. We would also like to develop an intuitive and efficient metaphor to represent both the software product and the software process.

References

[1] D. Beyer and A. E. Hassan. Animated visualization of software history using evolution storyboards. In *WCRE '06: Proc. Working Conf. on Reverse Engineering*, pages 199–210, 2006.

[2] T. Bladh, D. A. Carr, and M. Kljun. The effect of animated transitions on user navigation in 3D tree-maps. In *Proc. Intl. Conf. on Information Visualization*, pages 297–305, 2005.

[3] S. R. Chidamber and C. F. Kemerer. A metric suite for object oriented design. *IEEE Trans. on Software Engineering*, 20(6):293–318, June 1994.

[4] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proc. ACM Symp. on Software Visualization*, pages 77–86, 2003.

[5] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *WCRE '06: Proc. Working Conf. on Reverse Engineering*, pages 189–198, 2006.

[6] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *CSMR '06: Proc. Conf. on Software Maintenance and Reengineering*, pages 229–238, 2006.

[7] M. D'Ambros, M. Lanza, and H. Gall. Fractal figures: Visualizing development effort for cvs entities. In *Proc. IEEE Intl. Workshop on Visualizing Software for Understanding and Analysis*, pages 46–51, 2005.

[8] K. Dhambri, H. Sahraoui, and P. Poulin. Visual detection of design anomalies. In *CSMR '08: Proc. Conf. on Software Maintenance and Reengineering*, pages 279–283, 2008.

[9] J.-D. Fekete and C. Plaisant. Interactive information visualization of a million items. In *INFOVIS '02: Proc. IEEE Symp. on Information Visualization*, pages 117–124, 2002.

[10] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 1998.

[11] M. Fischer and H. Gall. Evograph: A lightweight approach to evolutionary and structural analysis of large software systems. In *WCRE '06: Proc. Working Conf. on Reverse Engineering*, pages 179–188, 2006.

[12] Y. Frishman and A. Tal. Dynamic drawing of clustered graphs. In *INFOVIS '04: Proc. IEEE Symp. on Information Visualization*, pages 191–198, 2004.

[13] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: putting icing on the UML cake. In *Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314, 2004.

[14] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *WCRE '04: Proc. Working Conf. on Reverse Engineering*, pages 172–181, 2004.

[15] B. Johnson and B. Shneiderman. Treemaps: A space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization Conf.*, 1991.

[16] G. Langelier, H. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proc. IEEE/ACM Intl. Conf. on Automated Software Engineering*, pages 214–223, 2005.

[17] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Langage et modèles à objets*, pages 135–149, 2002.

[18] G. Lommerse, F. Nossin, L. Voinea, and A. Telea. The visual code navigator: An interactive toolset for source code investigation. In *Proc. IEEE Symp. on Information Visualization*, pages 25–32, 2005.

[19] M. Lungu, M. Lanza, T. Girba, and R. Heeck. Reverse engineering super-repositories. In *WCRE '07: Proc. Working Conf. on Reverse Engineering*, 2007.

[20] L. Mason. Fostering understanding by structural alignement as a route to analogical learning. *Instructionnal Science*, 32(6):293–318, November 2004.

[21] C. Mesnage and M. Lanza. White coats: Web-visualization of evolving software in 3D. In *Proc. IEEE Intl. Workshop on Visualizing Software for Understanding and Analysis*, pages 40–45, 2005.

[22] Q. V. Nguyen and M. L. Huang. A space-optimized tree visualization. In *Proc. IEEE Symp. on Information Visualization*, pages 85–92, 2002.

[23] S. C. North. Incremental layout in dynadag. In *GD '95: Symp. on Graph Drawing*, pages 409–418. Springer, 1996.

[24] G. M. Oster and A. J. Kusalik. Icola — incremental constraint-based graphics forvisualization. *Constraints*, 3(1):33–59, 1998.

[25] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proc. ACM Symp. on Software Visualization*, pages 67–75, 2005.

[26] M. Shanmugasundaram, P. Irani, and C. Gutwin. Can smooth view transitions facilitate perceptual constancy in node-link diagrams? In *GI '07: Proc. Graphics Interface*, pages 71–78, 2007.

[27] S. L. Voinea and A. Telea. A file-based visualization of software evolution. In *Proc. ASCI: Advanced School for Computing and Imaging*, 2006.

[28] J. Wu, R. C. Holt, and A. E. Hassan. Exploring software evolution using spectrographs. In *WCRE '04: Proc. Working Conf. on Reverse Engineering*, pages 80–89, 2004.

[29] X. Wu, A. Murray, M.-A. Storey, and R. Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *WCRE '04: Proc. Working Conf. on Reverse Engineering*, pages 90–99, 2004.

[30] X. Xie, D. Poshyvanyk, and A. Marcus. Visualization of cvs repository information. In *WCRE '06: Proc. Working Conf. on Reverse Engineering*, pages 231–242, 2006.