Evolution Storyboards: Visualization of Software Structure Dynamics

Dirk Beyer *
EPFL, Switzerland

Ahmed E. Hassan
Research In Motion, Canada

Abstract

Large software systems have a rich development history. Mining certain aspects of this rich history can reveal interesting insights into the system and its structure. Previous approaches to visualize the evolution of software systems provide static views. These static views often do not fully capture the dynamic nature of evolution. We introduce the Evolution Storyboard*, a visualization which provides dynamic views of the evolution of a software's structure. Our tool implementation takes as input a series of software graphs, e.g., call graphs or co-change graphs, and automatically generates an evolution storyboard. To illustrate the concept, we present a storyboard for PostgreSQL, as a representative example for large open source systems. Evolution storyboards help to understand a system's structure and to reveal its possible decay over time. The storyboard highlights important changes in the structure during the lifetime of a software system, and how artifacts changed their dependencies over time.*

Keywords: Software evolution, reengineering, software visualization, software clustering, software structure analysis, dependency analysis, force-directed graph layout

1 Introduction

The history of large software systems contains noteworthy events (e.g., major refactoring or re-architecting) and interesting time periods (e.g., bug fixing, or active and quiet development periods). Such information is rarely documented, instead it is kept in the minds of senior developers who have been working on the system since its initial release, and is relayed from one developer to the next through personal communication. Large projects risk losing such historical information as developers depart from the project. However, historical information is important since future decisions may be affected by lessons learned over time.

Information about the evolution of a system's structure can be partially recovered from version control repositories, and may reveal interesting events in the lifetime of long lived projects. The software engineering literature contains several studies that propose tools and techniques to visualize historical information about projects. Previous visualization techniques are static in the sense that a single graph is used to summarize the various periods in the lifetime of a software system. Static views are not capable of capturing the dynamic nature of software evolution. On the other hand, a movie visualizing the evolution of a software system has also its shortcomings. Developers watching such a movie have little control over it. They are likely to miss interesting events, and are not able to easily focus on particular time periods or parts of the system.

To complement existing approaches, we propose the *Evolution Storyboard* visualization, which strikes a balance between a static image and a full blown movie. The evolution storyboard presents dynamic panels, which depict consecutively important events and periods in the history of a software system: for a series of different versions of a system —given as dependency graphs—, the method produces a series of visualizations of the structure of the system. This series of visualizations is then combined and extended in order to form an animated storyboard. The method is parametric in the type of dependency graph that is used.

The visualization of a single dependency graph (panel) is a placement of software artifacts in a two-dimensional space, where the positions of the artifacts are obtained by an energy-based graph clustering algorithm. These placements have the property that artifacts that are dependent have close positions, and artifacts that are independent have distant positions. The particular instantiation of the method for the experiments in this paper uses as dependency graph the co-change graph, which is an abstract representation of the change transactions during the development of a system. The input data for this instance of the method can be extracted from the version repository of a software system using a simple and efficient extraction process. The method is then completely programming-language independent, and the software artifacts are not restricted to program source but can also represent, e.g., documentation and test cases.

*Supported in part by the MICS NCCR of the SNSF.

An evolution storyboard consists of a series of panels. Each panel represents the dependency information of a particular time period in the lifetime of a software system. The panels are displayed in sequence for the purpose of visualizing and animating the history of the dependency structure. Each panel optionally indicates the movement of artifacts over time using animated arrows. If co-change graphs are used as dependency graphs, the panels represent the co-change information, and the artifacts are positioned closely together if they were often changed together.

Related work. Ball et al. mined and visualized graphs based on common source code changes from the version control repository [2]. Baker and Eick used animated visualizations of software metrics to observe the growth of software systems [1]; they did not use co-change information from version repositories. The visualization of release histories by Gall et al. [8] is produced by generating two-dimensional pictures and combining them to a layered structure (system–subsystem–module) for different versions of the system over time; several attributes are used to color the visualizations. Collberg et al. proposed a method that is limited to source code objects and the programming language Java, to produce sequences of static layouts of call and inheritance graphs [5]. The method uses energy models that are not designed for clustering, but for aesthetic layout of non-software graphs. Fischer and Gall visualized the dependencies between features [6], and Lanza used matrices to represent evolution data [9].

2 Concepts

Dependency graphs. Previous approaches to visualize dependency structures are based on graph models where a dependency between two artifacts is modeled as an edge between the two corresponding artifact nodes. If the strength of a dependency was important, weights were assigned to the edges. In contrast to this 'condensed' graph model, we prefer to keep more information in the graph. If the reason for the (assumed) dependency is a syntactical coupling of three artifacts, then we want to keep this fact in our model. For this purpose, we introduce a new model for dependency graphs with a new type of nodes, called *dependency node*, which captures the reason for the dependency.

A *dependency graph* is a bipartite, undirected graph $G = (V,E)$, where V is the set of nodes and E is the set of edges. A node $v \in V$ is either an *artifact node* or a *dependency node*. An edge $\{d,a\} \in E$ between a dependency node $d \in V$ and an artifact node $a \in V$ exists if node d models the abstract reason that makes artifact a dependent on all other artifacts a' with $\{d,a'\} \in E$. Software artifacts are, e.g., subsystems, files, classes, or functions. Dependencies can be induced by, e.g., calls, subtype relations, or co-changes.

Example. Let x.h be a header file, and let y.c and z.c be two implementation files that contain both a preprocessor directive to include file x.h. This fact of a syntactical inclusion dependency can be modeled by the following (sub-) graph $G = (\{d, \mathsf{x.h}, \mathsf{y.c}, \mathsf{z.c}\}, \{\{d, \mathsf{x.h}\}, \{d, \mathsf{y.c}\}, \{d, \mathsf{z.c}\}\})$.

We restrict ourselves to unweighted graphs for clear presentation. The extension to weighted graphs is natural (cf. [3]). The dependency graphs that we use in the application section are *co-change graphs* [4]. In this case the graph represents the change history of the software system in the following way: the dependency nodes represent version-control change transactions, and an edge $\{d,a\}$ between a change transaction node d and an artifact node a exists if artifact a was changed by change transaction d.

Energy-based graph layout. A *layout* of a graph (V,E) is a function $p : V \to \mathbb{R}^d$, which maps each node from V to a position in the d-dimensional real space ($d \in \{2,3\}$). An *energy model* is an evaluation function U that assigns to each layout p a real number u. The layout p is the *best layout* if $U(p)$ is the global minimum of function U. This means that the energy model encodes the desired properties of the layout. Since we are interested in grouping a dependency graph into groups that represent subsystems, we use energy models with clustering properties. The algorithm that computes a layout with minimal energy is called a minimizer, and the concept is called *energy-based graph layout* (cf. also [7]).

To efficiently compute an approximation of the best layout of a dependency graph for a single version of the software system, we run the graph layout tool CCVISU[1] [3], and use its default energy model (for the details, cf. the work on clustering co-change graphs [4]). The tool CCVISU is also used to extract the co-change graph from the version control repository, in cases where we use a co-change graph as our dependency graph.

Evolution storyboards. The evolution storyboard divides the lifetime of a software system into several time periods. For each time period, a layout of the dependency graph is used to visualize the structure of the system in one panel.

Let $G_t = (V_t, E_t)$ be the dependency graph of the software system at time t, and let p_t be the best layout of the graph G_t. An *evolution storyboard* for the sequence of time stamps $t_0, t_1, ..., t_n$ is a sequence of n panels $P_1, P_2, ..., P_n$, one for each time period. The panel P_t consists of the layout p_t and a set M_t of *moving nodes* $v \in V_t \cap V_{t-1}$ that moved a lot, i.e., the Euclidean distance $||p_t(v) - p_{t-1}(v)||$ of the node's current and previous position is larger than a certain threshold. The threshold is used to ignore negligible movement, to avoid clutter in the visualization.

The visualization of a single panel P_t is done as follows: For every artifact node $a \in V_t$, we draw a filled circle at position $p_t(a)$, with the circle area proportional to the edge

[1] http://mtc.epfl.ch/∼beyer/CCVisu

degree $deg_{G_t}(a) = |\{\{a,d\} \mid \{a,d\} \in E_t\}|$, i.e., the number of dependencies the artifact is involved in. For every artifact node $a \in M_t$, we draw a filled circle in grey at the node's previous position $p_{t-1}(a)$, with a circle area proportional to the previous edge degree $deg_{G_{t-1}}(a)$, and a grey line from the previous position $p_{t-1}(a)$ to the current position $p_t(a)$. This line is animated by moving arrow heads that move from $p_{t-1}(a)$ to $p_t(a)$. Furthermore, we visualize the change in the degree of dependency since the *last time stamp*: we draw a red ring for the artifact node a, with the area of the ring proportional to the difference of the edge degree between graph G_t and the previous graph G_{t-1}, i.e., $deg_{G_t}(a) - deg_{G_{t-1}}(a)$ if $a \in V_{t-1}$, and 0 otherwise. This means: large nodes depend on many other nodes, and nodes with large rings changed their degree of dependence a lot.

In the evolution storyboard, these panels are displayed in a sequence for the purpose of visualizing and animating the historical changes in the dependency of the nodes. To ensure the stability of the layouts in the storyboard, we feed the layout minimizer when computing layout p_i with initial positions of layout p_{i-1} of the previous time stamp.

Tool implementation. The generation of evolution storyboards is implemented as an extension of the tool CCVISU[1] [3]. The output visualization, i.e., the storyboard, is dumped as a collection of SVG files, which are embedded in a single HTML document for navigation. The use of standard web technology makes the tools easier to use and adopt by software practitioners.

3 Application

In this section we present the evolution storyboard for POSTGRESQL. The evolution storyboard is created by applying our visualization approach to the information stored in the version control repository (CVS in this case) of the open source software system POSTGRESQL. The version control repository contains the version history of more than 9 years of system development and maintenance. We instantiate our dependency graph with the extracted *co-change graph*. The co-change graph at time t contains all changes up to time t. We have extracted the change history of 4,114 files, and our largest extracted co-change graph models 20,107 commits (resulting in a total of 24,221 nodes) with 87,301 single changes (i.e., edges).

The complete generated storyboard for POSTGRESQL contains 23 panels. Each panel represents a snapshot of the system after 3 months of development. The graph layout in each panel has the property that nodes are positioned closely together if the corresponding files were changed often together, and at distant positions if the nodes were rarely changed together. (Only artifact nodes are drawn.) For POSTGRESQL, we identified 9 major subsystems and assigned a color to each subsystem: Port (green), Executor (red), Optimizer (blue), Parser (cyan), Storage Manager (magenta), Query Evaluation Engine (yellow), Nodes (dark magenta), Access (dark cyan), and ADT (light blue). The rest of the files (for example files which are utilities) are assigned to color pink. Figure 2 shows several panels, which correspond to different time periods in the lifetime of POSTGRESQL. The interface of the tool permits the user to move quickly between consecutive panels. The ability to move quickly between panels offers a motion-like animation, which permits to animate and study closely the changes in the layout and in the structure of a software system over time. A cursory look at Fig. 2 shows that over time the Executor (red) and Optimizer (blue) subsystems are moving closer to each other, indicating that they are changing more often together, and are likely becoming more dependent on each other. The Query Evaluation Engine (yellow) is the subsystem with the most instable dependency structure; many file nodes moved over long distances, and the files were more and more often changed together with the Parser subsystem.

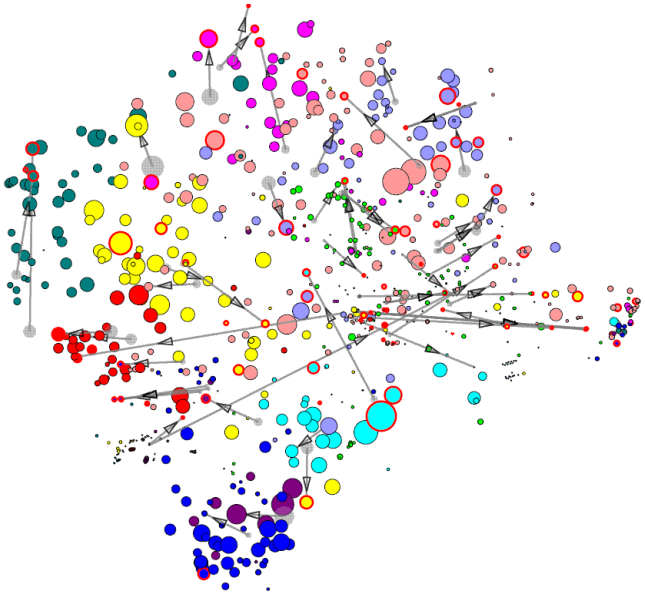

Figure 1. A storyboard panel for PostgreSQL for the period 2005-04-01 to 2005-07-01

Figure 1 shows a panel for one time period. For each panel, the user can animate the panel. The animation shows the movement of nodes between consecutive time stamps, which are 3 months apart in Fig. 1. The old location of a node is shown in grey, and a grey arrow points to its new location. The size of a node indicates how many changes have occurred to that node during its lifetime, and a red ring highlights nodes which have changed during the last 3 months. The size of the red ring is proportional to the amount of changes during the period of the storyboard panel. The user can also zoom into a particular area of the graph and closely watch the evolution animation for that area of the graph.

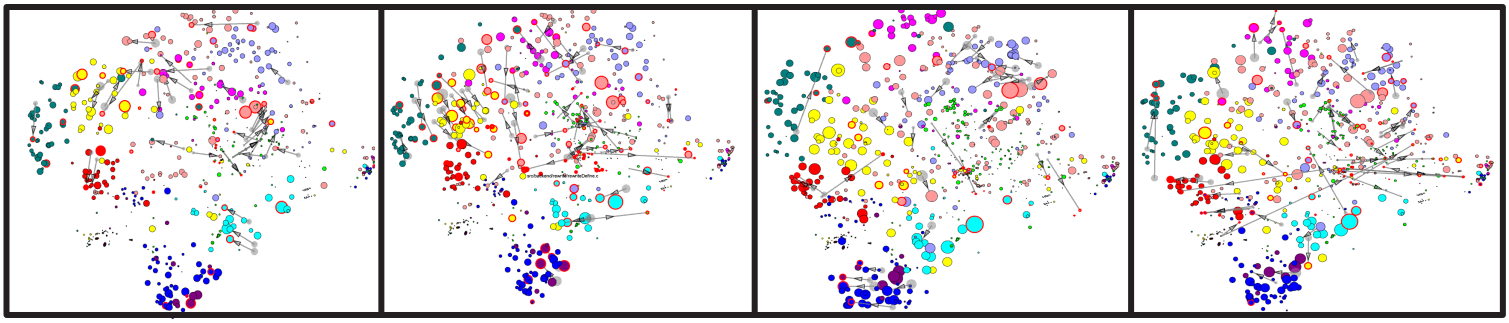

Figure 2. Storyboard panels for PostgreSQL's co-change information

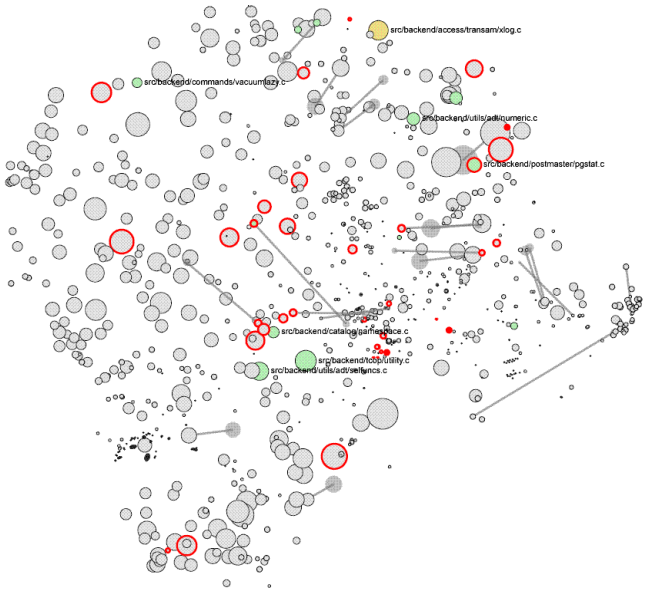

Figure 3. A HeatMap storyboard panel

Instead of using the colors from the authoritative subsystem-decomposition, the storyboard tool can alternatively color the nodes based on their movement over time. This visualization permits users to notice nodes (files) that may require refactoring, since they were co-changed with many different files over time and are moving too much between the different storyboard panels. Figure 3 shows an example of this HeatMap view of the storyboard. Files which have moved in more than 40 % of the panels are colored orange; files which have moved in more than 30 % of the panels are colored yellow; and files which have moved in more than 20 % of the panels are colored green. Finally files that have moved in than less than 20 % of the evolution panel are colored grey. We note that in Fig. 3 the POSTGRESQL transaction log manager (`access/transam/xlog.c`) is yellow, indicating that this file tends to change with many different files over time. This is likely to be a good indication that the transaction logging feature is spread out across the code and not localized in a particular subsystem.

4 Summary

A storyboard is traditionally produced beforehand to help directors and cinematographers to study movie scenes to uncover potential problems before they occur. In this work about software engineering, we propose the concept of *evolution storyboards*, to be used by developers of software to replay and study the history of a software system. Practitioners can use our evolution storyboards to better understand the rationale behind the current structure of the software system, and to uncover problems and possible improvements to the software structure.

References

[1] M. Baker and S. Eick. Visualizing software systems. In *Proc. ICSE*, pages 59–67. IEEE, 1994.

[2] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk ... In *Proc. Workshop Process Modelling and Empirical Studies of Software Engineering*, 1997.

[3] D. Beyer. Co-change visualization. In *Proc. ICSM'05, Industrial and Tool volume*, pages 89–92, Budapest, 2005.

[4] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proc. IWPC*, pages 259–268. IEEE, 2005.

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

[6] M. Fischer and H. Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *J. Software Maintenance and Evolution: Research and Practice*, 16(6):385–403, 2004.

[7] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21(11):1129–1164, 1991.

[8] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Proc. ICSM*, pages 99–108. IEEE, 1999.

[9] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proc. VISSOFT*, pages 37–42. ACM, 2001.