Animated Visualization of Software History using Evolution Storyboards

Dirk Beyer [*]
EPFL, Switzerland

Ahmed E. Hassan
University of Victoria, Canada

Abstract

The understanding of the structure of a software system can be improved by analyzing the system's evolution during development. Visualizations of software history that provide only static views do not capture the dynamic nature of software evolution. We present a new visualization technique, the Evolution Storyboard, which provides dynamic views of the evolution of a software's structure. An evolution storyboard consists of a sequence of animated panels, which highlight the structural changes in the system; one panel for each considered time period. Using storyboards, engineers can spot good design, signs of structural decay, or the spread of cross cutting concerns in the code. We implemented our concepts in a tool, which automatically extracts software dependency graphs from version control repositories and computes storyboards based on panels for different time periods. For applying our approach in practice, we provide a step by step guide that others can follow along the storyboard visualizations, in order to study the evolution of large systems. We have applied our method to several large open source software systems. In this paper, we demonstrate that our method provides additional information (compared to static views) on the ArgoUML project, an open source UML modeling tool.

1. Introduction

Large software systems have a rich development and maintenance history, which is filled with 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 for several years. Such information is relayed from one developer to the next through anecdotes and other informal communication.

Many future decisions in a project are affected by lessons that can be learned from the software's history. Projects risk losing this rich history over time, because senior developers tend to move on to other projects, or eventually leave the organization. As these developers depart, the project would lose the developers' undocumented wisdom and knowledge. However, version control systems (VCS) contain valuable historical information about a project, and mining the VCS repository may reveal interesting events in the development and maintenance of long-lived projects.

The software engineering literature contains many approaches for visualizing and studying the evolution of software systems. Most of these visualizations are static — static in the sense that a single visualization (e.g., graph or numerical plot) is used to summarize the various periods in the lifetime of a system. Static visualizations often cannot capture the dynamic nature of software evolution. We attempt to overcome this shortcoming with our new visualization concept *evolution storyboard*.

A storyboard is traditionally produced beforehand to help directors and cinematographers to study movie scenes to uncover potential problems before they occur [3, 11, 13]. In our work about software engineering, we define the concept of evolution storyboards to replay and study the history of a software system, retrospectively. Practitioners can use our evolution storyboards to better understand the rationale behind the current structure of the software system, and to uncover problems (e.g., structural decay) and possible improvements (e.g., refactoring and code re-organization) to the software structure. In contrast to static visualizations, the evolution storyboard presents dynamic views, which depict consecutively important events and periods in the lifetime of long-lived software systems. However, we do not propose a full movie visualization since developers watching such a movie would have difficulties controlling it. Developers would likely miss interesting events, and are not able to easily focus on particular periods or parts of the system. In short, the evolution storyboard visualization strikes a balance between static and movie-like visualization to permit developers to effectively study the dynamics of software evolution.

An evolution storyboard consists of a series of dynamic panels. Each panel represents the dependency information of a particular period in the lifetime of a studied software system. Each panel contains software artifacts (e.g.,

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

files or functions) that are placed 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. Each panel highlights changes of the software artifacts' dependency degree, and optionally showcases the movement of them through animated arrows. The evolution storyboard assists developers in spotting artifacts that are becoming more or less dependant on others.

The evolution storyboard visualizes changes of the system structure over time based on dependency graphs. For the case studies mentioned in this paper, we used co-change graphs —an abstract representation of the change transactions— as dependency graphs. However, the method is parametric in the kind of dependency graph. A co-change graph can be extracted from the VCS repository of a software system using a simple and efficient extraction process. The panels in this paper represent the co-change information: artifacts are positioned closely together if they were often changed together. The approach is programming-language independent, and the software artifacts are not restricted to program source but can also represent artifacts such as build scripts, documentation, and test cases.

Storyboard visualizations help practitioners to understand the evolution of the structure of a software system, complementing other existing techniques. In comparison to static visualizations of evolution, our animated visualization can monitor how the structure of dependencies in a software system has changed over time. More concretely, our work provides support for the following reengineering subtasks (selection of possible applications of our approach):

- **Explaining decay symptoms.** Storyboards help explaining symptoms and clarify misconceptions. E.g., although a static visualization may reveal symptoms of decays, the storyboard could reveal that these symptoms have existed over the years since the beginning of the project and are more likely design decisions rather than decay symptoms.

- **Highlighting refactoring candidates.** Storyboards help uncover artifacts that are good candidates for refactorings. Artifacts that are responsible for a variety of concerns in a software system and tend to change with many different groups of artifacts that represent particular concerns, are examples of such candidates. Such refactoring candidates are highlighted in the panels of a storyboard since they frequently move as they change often with different artifacts over time.

- **Spotting good structure.** Storyboards highlight well designed sets of artifacts. Such artifacts tend to change together, and rarely change with other artifacts. Our clustering layout causes artifacts with such characteristics to separate from the rest of the system and to form their own cluster. The storyboard panels help us to visualize the emergence of such clusters over time.

Many useful software visualization approaches lack to provide a guideline that explains how to use the visualization and what insights can be obtained from the pictures. We provide a **step by step guideline** for using our method. The guide aims at helping to get started when analyzing the software's evolution, in order to gain a better understanding of the current structure of the system. Practitioners can follow this simple guideline to understand, visualize and animate their system and its structure. We evaluate our own technique along the proposed guideline in the last section of the paper, and showcase our new storyboard visualization on a large open source project. We demonstrate how to address the above reengineering tasks for these systems.

Contributions. The contributions of this work can be summarized as follows:

1. We define a new dependency graph model, which generalizes the co-change graph model from [5] to arbitrary software dependency graphs.

2. We introduce the evolution storyboard, a new concept for animated visualizations of historical information about the software structure, and the storyboard panel, which is the building block for highlighting structural differences between two versions of a system.

3. We highlight several obvious applications of our technique to typical reengineering tasks, and show on large example systems how it provides solutions.

4. We formulate a guideline for the usage of our visualization, in order to make the approach applicable by and useful for non-experts, and to make the evaluation repeatable on other subject systems.

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. 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 [10]. 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 [6]. 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[7], and Lanza used matrices to represent evolution data [14]. Beyer and Noack introduced energy-model based visual clustering of software systems using co-change graphs [5]. We use their method to compute the positions of artifacts in our panels.

Outline. Section 2 defines the graph model that our method is based on. Section 3 defines how the static layouts are computed and how they are combined to form a dynamic storyboard. We use a clustering layout technique to position artifacts in the storyboard panels. Then we compute the panels for each period, add the animation to each panel, and apply several filtering heuristics to avoid pollution of the visualization with unimportant information, to make the presentation clear and easy to understand. In Section 4 we propose a usage guideline for our approach and report our findings from applying the technique to three large open-source software projects. We conclude the paper with summarizing and discussing our approach.

2. Graph Model

An evolution storyboard is constructed by combining visualizations of multiple dependency graphs for different periods. This section defines the graph model that is used as input data for the method; these graphs can be automatically extracted from version control repositories (such as CVS), however this is not detailed in this paper (cf. [16, 17, 8, 4, 12]).

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. 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 a *dependency node*, which captures the reason for the dependency.

A *dependency graph* is a weighted, bipartite, undirected graph $G = (V, E, w)$, where V is the set of nodes, E is the set of edges, and $w : E \to \mathbb{R}$ is a total function that assigns a weight to each edge. 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 $\{d, a'\} \in E$. The weight of an edge can be used to model the importance of the dependency (if the weight is not mentioned explicitly, we assume an edge weight of 1). Software artifacts are, e.g., subsystems, files, classes, or functions. Dependencies can be induced by, e.g., calls, subtype relations, or co-changes.

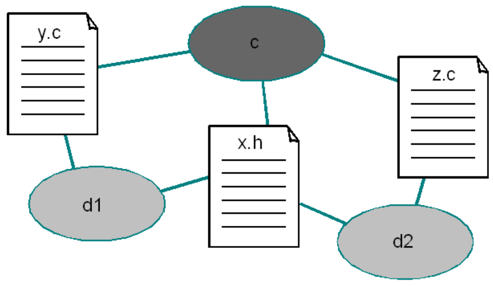

Figure 1. Combined dependency graph

Example 1. (Dependency graph) 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 $G = (\{d_1, d_2, \text{x.h}, \text{y.c}, \text{z.c}\}, \{\{d_1, \text{x.h}\}, \{d_1, \text{y.c}\}, \{d_2, \text{x.h}\}, \{d_2, \text{z.c}\}\}, w)$, $w : V \to \{1\}$.

Co-change graphs. The dependency graphs that we use in the examples in this paper are *co-change graphs* [5]. In this case the graph represents the change history of a 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.

Weighted combinations of different graphs. Since we model the abstract reason for a dependency in a separate graph node, it is possible to represent several different kinds of dependencies between two artifacts in the same graph. E.g., two artifacts can be connected by an inclusion dependency node of an inclusion graph and by a change transaction node of a co-change graph. The combination of two different dependency graphs is the union of the two graphs. If different dependency graphs are not considered equally important, then the dependency graphs can be weighted with different edge weights in the combination.

Example 2. (Combined dependency graph) Figure 1 shows a combined dependency graph that includes the facts from the previous example, and additionally a co-change dependency c between all three artifacts.

3. Visualization

An evolution storyboard visualizes a sequence of dependency graphs for several periods in the development of a software system. Therefore the storyboard consists of several panels — each panel visualizing the dependency graph at a particular period. In this section, we first present our technique to layout a single dependency graph for a panel. Then we discuss three alternatives for instantiating the method to co-change graphs. We proceed with defining the evolution storyboard and how it is drawn in the visualization. In the visualization we need to ensure that we filter out unimportant information that otherwise blur the generated picture. Finally, we discuss our tool implementation for generating storyboards, and highlight the benefits and limitations of the current implementation.

Visualization of single panels. A *layout* of a graph (V, E, w) is a function $p : V \to \mathbb{R}^d$, which maps each node from V to a position in the d-dimensional real space ($d \in \{2,3\}$). We construct the function p using *energy-based graph layout* (cf. [9]). 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* [5, 15]. The algorithm that computes a layout with minimal energy is called a minimizer. To efficiently compute an approximation of the best layout of a dependency graph for a single panel, we run the graph layout tool CCVISU[1].

Visualization of a sequence of graphs. Instead of visualizing a single dependency graph which represents the full history of a software system (as done in [5]), we are here interested in visualizing how that particular dependency graph *evolves* over time. This visualization helps to gain insights into changes to interdependencies between artifacts and subsystems during the development of the system.

For the instantiation of our method to *co-change graphs*, we considered the following three alternatives for the choice of the co-change graphs at every panel:

1. Time-based co-change graphs: We consider graphs after a constant time period. In our case studies, we chose to create a new panel after every 3 months.

2. Change-count-based co-change graphs: We consider graphs after a constant number of additional dependencies for each panel. In our case studies, we chose to create a new panel after every 1000 dependencies.

3. Release-based co-change graphs: We consider graphs at certain release points. We create a new panel for each release of the software system.

Each alternative has its benefits and shortcomings; the time-based graphs quickly reveal in their corresponding storyboard panels quiet time periods in the lifetime of a software system, where little or no changes have occurred to the software's dependency structure. On the other hand, using the change-count-based or release-based alternative, the user is not aware of the amount of time that has elapsed between two panels in the storyboard. For example, one panel may capture 1000 changes that occurred in a single day whereas another panel may capture the same amount of changes over a few weeks or months.

[1]http://mtc.epfl.ch/∼beyer/CCVisu

Evolution storyboards. The evolution storyboard divides the lifetime of a software system into several periods, and shows a storyboard panel for each of them. Every storyboard panel is based on a layout of the dependency graph to visualize the structure of the software system.

Let $G_t = (V_t, E_t, w_t)$ 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 marks $t_0, t_1, ..., t_n$ is a sequence of n panels $P_1, P_2, ..., P_n$, one for each period between two subsequent marks. The panel P_t consists of the layout p_t and a set $M_t \subseteq V_t \cap V_{t-1}$ of *animated nodes*. We use the set M_t to restrict the animation of a panel to nodes that reflect the change of structure, to filter out negligible change of nodes, to avoid clutter in the panel. The conditions for a node of being considered as animated node are detailed below.

The visualization of a single panel P_t is implemented 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 degree $deg_{G_t}(a) = \sum_{\{a,d\} \in E_t} w_t(\{a,d\})$, i.e., the sum of the weights of dependency edges to the artifact node. For every artifact node $a \in M_t$, we draw a grey filled circle 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 bubbles that move from $p_{t-1}(a)$ to $p_t(a)$. Furthermore, we visualize the change in the degree of dependency since the previous panel (i.e., the last period): we draw a red ring within the circle 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 $deg_{G_t}(a)$ otherwise. This means: large nodes depend on many other nodes, and nodes with large rings have changed their degree of dependence a lot during the last period.

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. 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 the user to animate and study closely changes in the layout and in the structure of a software system over time.

Stabilization and filtering techniques. To achieve a sequence of layouts that are similar to each other (*stable*), but emphasize the change in the structure of the system, we feed our graph-layout algorithm when computing the layout for panel p_i with the positions of the artifacts in panel p_{i-1}. I.e., the layout algorithm starts with the positions of the previous layout, and adopts the positions of the artifacts in the current layout according to the current graph, in order to produce a layout with minimal energy.

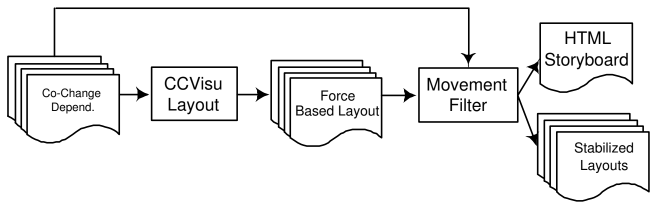

Figure 2. Approach overview

In the following we explain in more detail how we filter out animation of negligible movement, to avoid that unimportant information blurs the visualization. An artifact node $a \in V_t \cap V_{t-1}$ is included in the set M_t of animated nodes if all of the following conditions hold:

1. The Euclidean distance $||p_t(a) - p_{t-1}(a)||$ of the node's current and previous position is larger than a certain threshold (e.g., 5 % of the panel's width). The animation of artifacts not fulfilling this condition would clutter the visualization by many very small changes of positions.

2. The degree of dependency $deg_{G_t}(a)$ (i.e., the edge degree) is above a certain threshold (i.e., the dependency of the artifact changed often). Artifacts not fulfilling this condition are not important for the overall structure of the system and its evolution.

3. The change of the degree of dependency $deg_{G_t}(a) - deg_{G_{t-1}}(a)$ in the last period (i.e., the difference of the edge degrees) is above a certain threshold (i.e., the dependency of the artifact changed a considerably in the last period). Artifacts not fulfilling this condition are not important for the change of the structure during the last period.

Note that restricting the animation to a subset of artifacts does not affect the positions of the other artifacts in the panel. If, e.g., an artifact changes its dependencies slowly over a long period of time, it would always change its position a bit, but we would not emphasize it by animation.

Tool implementation. The concepts of the evolution storyboard are implemented as a Java application, in order to be platform independent. Clustering graph layouts are efficiently computed by calling functions from CCVISU's layout library[1] [4], which is based on the best known algorithm for computing energy-based graph layouts (Barnes-Hut). Figure 2 gives an overview of the tool implementation. The dependency graphs for each panel are generated through either the time-based, change-count-based, or release-based alternative. These graphs are feed to the graph-layout tool. The tool uses the layout from previous panels to generate the layout for the following panel. For the first panel, the tool uses a random layout. The generated layouts along with the dependency graphs are feed into the movement filter. The filter uses the aforementioned thresholds and heuristics to reduce the animation of unimportant nodes. The complete evolution storyboard is displayed using standard, vendor-independent Java technology.

Alternatively, the storyboard can be written to several SVG files (one for each panel), which are embedded in a single HTML document for navigation. The use of standard web technology makes the tool easy to use and adopt by software practitioners. To permit users to move quickly between panels, all SVG files are loaded in memory using HTML layers. Once a specific panel is to be shown, its particular HTML layer is brought to the front. This technique permits very fast interaction with the storyboard with no lag due to loading up an SVG file corresponding to a panel. Users can zoom into a particular area of a panel to monitor closely the interactions of a limited set of artifacts. Although each panel is represented as a separate SVG file, all the SVG files communicate together through JavaScript functions to ensure that they are synchronized. For example, once a user zooms into a particular panel then the viewpoint of all other panels in the storyboard are adjusted to reflect the same viewpoint.

Coloring schemes. Each generated storyboard supports two coloring schemes by default. Additional coloring schemes are possible. One coloring scheme is based on an authoritative decomposition of the studied software system. In this coloring scheme, all artifacts (i.e., nodes) in a subsystem are colored using a particular color which represents the subsystem. This coloring scheme highlights how interdependencies between different subsystems change over time. The second coloring scheme is a heat-based coloring scheme (called HeatMap): artifacts 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 panels are colored grey. This heat coloring scheme permits us to note artifacts that are continuously moving due to permanent change of dependencies.

4. Application

4.1. Step by Step Guideline

To get started with using evolution storyboards, we propose the following step by step guideline, which we derived from our own experience in analyzing large software systems. We hope that this guideline provides help for others who are interested in using storyboards on their own software systems.

1. **Data preprocessing.** The input of the method is a set of dependency graphs for different versions. These need to be extracted from the system's version control repository (e.g., co-change graphs with CCVISU).

2. **Single-panel analysis.** Analyze various static panels individually. Use an authoritative decomposition or the package/directory structure to color all artifacts of one subsystem with the same color. The area of an artifact circle indicates the degree of dependency (e.g., how often it was changed). If the dependency graph contains loosely coupled, cohesive groups of artifacts, then these groups appear in the layout as clusters. If these clusters consist of nodes of mostly the same color, then the dependency graph 'matches' the authoritative decomposition. Watch for the following characteristics:

   (a) **Large nodes.** Artifacts represented by large nodes have a high degree of dependency in the graph (co-change graphs: were changed often in the lifetime of the project). These artifacts can be good candidates for reengineering.

   (b) **Separated subsystem components.** Artifacts that have the same color but are not closely positioned, are considered to be in the same subsystem, but do not strongly depend on each other. These nodes can be candidates for reassignment into different subsystems.

   (c) **Independent clusters.** A group of artifacts that is separated from the rest in the layout represents a very cohesive subsystem according to the system's dependency structure. The artifacts in such a cluster strongly depend on each other (change often together), and are not very dependant on (often changed together with) other artifacts.

   Such (groups of) artifacts are good candidates for closer investigation.

3. **Animated analysis.** Animate the dependency graph using the evolution storyboard. Watch for the following characteristics in the animation:

   (a) **Growing nodes.** Observe how nodes grow over time. In particular, focus on the large nodes that were noticed in the single-panel analysis. This will explain if artifacts had particular periods of heavy dependencies to other artifacts (were frequently changed) and quiet periods, versus artifacts that continuously had a stable degree of dependency. Also look for groups of nodes that do not change their size, these are 'stable' artifacts/subsystems, which do not change their dependencies to other artifacts.

   (b) **Traveling nodes.** Observe how nodes change their position over time. In particular, look for nodes that are frequently moving reasonable distances. This indicates that these artifacts are changing not only their degree but also their dependency *partners* excessively. Such artifacts that frequently move across the storyboard are good candidates for refactoring by breaking the artifact into smaller artifacts, according to the different responsibilities (each representing a particular concern).

   (c) **Cluster movement.** The single-panel analysis gives us a static glimpse of how the different subsystems (colors) are related. It does not show how the dependencies between these subsystems change over time. Concentrate on the following three particular aspects:

      i. **Independent clusters.** Examine how each cluster is moving over time in the storyboard. In particular, look for clusters that could be identified in the single-panel analysis as being well separated from the rest of the artifacts. If we observe that such clusters have moved slowly over time to become separated, then this can be the result of a restructuring activity and indicates that these clusters are highly cohesive and have low coupling to the rest of the system.

      ii. **Co-cluster movement.** Compare how each cluster has moved relative to the others. If two clusters move closer together, this indicates that these clusters are becoming more and more dependant on each other.

      iii. **Emerging clusters.** If a cluster moves out of the rest of the system or out of another cluster, this means that over time the dependencies were concentrated within that cluster. This effect can be a consequence of successful restructuring activities. The clustering energy model ensures that the a cohesive group of artifacts that is otherwise loosely coupled with the rest gets separated.

4. **Consult additional sources.** Compile a list of noteworthy artifacts and group of artifacts, and consult the system documentation, look up the source code or API, or consult a system expert, in order to get additional information about the components to select and refine the findings obtained using the evolution storyboard. We hope that this consultation of additional sources (which a tool cannot do for us) can be done more efficiently after exploring the evolution storyboard, since the user knows already which questions to ask.

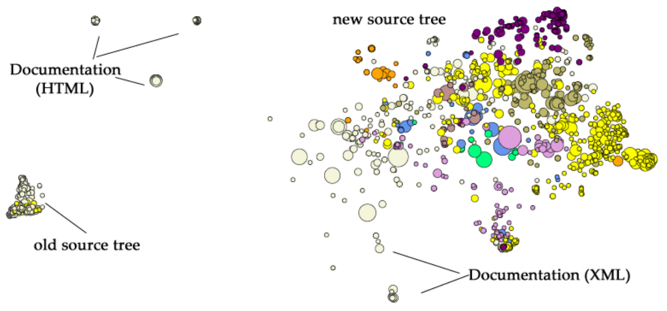

Figure 3. A storyboard panel for ArgoUML showing old (left cluster) and new (right cluster) source code files and documentation files

4.2. Case Studies

Our example storyboards were created by applying the approach to the information stored in the version control repository (CVS in this case) of three systems. We chose to use co-change graphs as input, because they are simple and efficient to extract from the version repository. We do not claim that this is the best choice of dependency graph, and our method is not limited to co-change graphs. For clear presentation in the paper (limited resolution when printed), we omit all artifacts with dependency degree less than 3.

We use both coloring schemes. The HeatMap coloring scheme is helpful in our animated analysis since it highlights frequently moving nodes and subsystems. The coloring by authoritative decomposition helps to observe how the dependency structure between the different subsystems changes over time. Our Java implementation and SVG JavaScripts permit us to switch between both coloring schemes on the fly, to not lose context. Once we observe an interesting set of nodes (e.g., frequently moving or growing fast), we can zoom and observe them more closely over time.

We have applied the evolution storyboard approach to the three large open source software projects POSTGRESQL, ARGOUML, and MOZILLA. POSTGRESQL is an open source relational database system with more than 15 years of active development. ARGOUML is an open source UML modeling tool which includes support for all standard UML 1.4 diagrams. MOZILLA is an open source web browser and mail client. The evolution storyboards highlighted several interesting historical events and information about the three systems. Panels in the POSTGRESQL storyboard were created using the time-based technique, whereas the panels in the storyboards of ARGOUML and MOZILLA are change-count-based. Due to space limitation, we discuss only the ARGOUML storyboard in this paper, but we encourage the reader to download and explore also the other storyboards (we refer to the online supplement).

ArgoUML. The ARGOUML repository contains the development history of 10,139 files over the last 8 years. Our largest extracted co-change graph, which is used in the last storyboard panel, models 10,108 commits (resulting in a total of 20,247 nodes with 57,036 single changes).

Single-panel analysis. Figure 3 shows a panel from our first storyboard for ARGOUML. The figure shows two large separated clusters. The different panels in the storyboard show how one large cluster was becoming more active with many changes applied to it and artifacts moving, while the other large cluster had very few changes (node sizes not growing) and no artifacts moving. A closer examination of the version control system for ARGOUML revealed that the system contains an old (no longer maintained) source tree and another actively enhanced and maintained tree with moving nodes and changing dependencies. Our generated storyboard offered us a dynamic view into how the team moved their development from their old source tree to the new source tree.

To increase the clarity of the generated storyboard, we decided to consider only the active source tree when we built our next storyboard for ARGOUML. By dropping the artifacts of the old tree from our visualization, the new artifacts could expand to fill the panel instead of using only half of the panel's space.

Figure 4 shows the first and last panel in the ARGOUML storyboard. The first panel indicates the colors assigned to each subsystem. Utility files and other files that are not assigned to any subsystem are drawn in white. Following our step by step guide, we note all the large nodes in the last panel. The names of the large nodes are indicated in the figure. These large nodes are good candidates to observe in the next step using animation. Moreover, we can investigate their overall movements using the HeatMap coloring scheme. We also observe that the UI subsystem (yellow) has grown considerably over the years and seems to be highly dependant on several other subsystems. Nevertheless, we notice in the lower right corner of the last panel the formation of a mini-cluster of the UI subsystem. Using the animated analysis, we can closely look at the formation of this cluster over time to gain a better understanding of the rationale behind its formation. Finally, we notice that the I8N-internationalization subsystem (orange) has remained over the years as a reasonably independent cluster that is separated from the rest of the system. This is a good indication that the internationalization concerns within ARGOUML are well contained except for a few orange files that are placed close to the other subsystems.

Animated analysis. Our single-panel analysis highlighted the formation of a mini-cluster of the UI subsystem (yellow) in the lower right corner in Fig. 4 (last panel). For closer investigation, we zoom into the area that is greyed in Fig. 4. Figure 5 shows four different panels from the

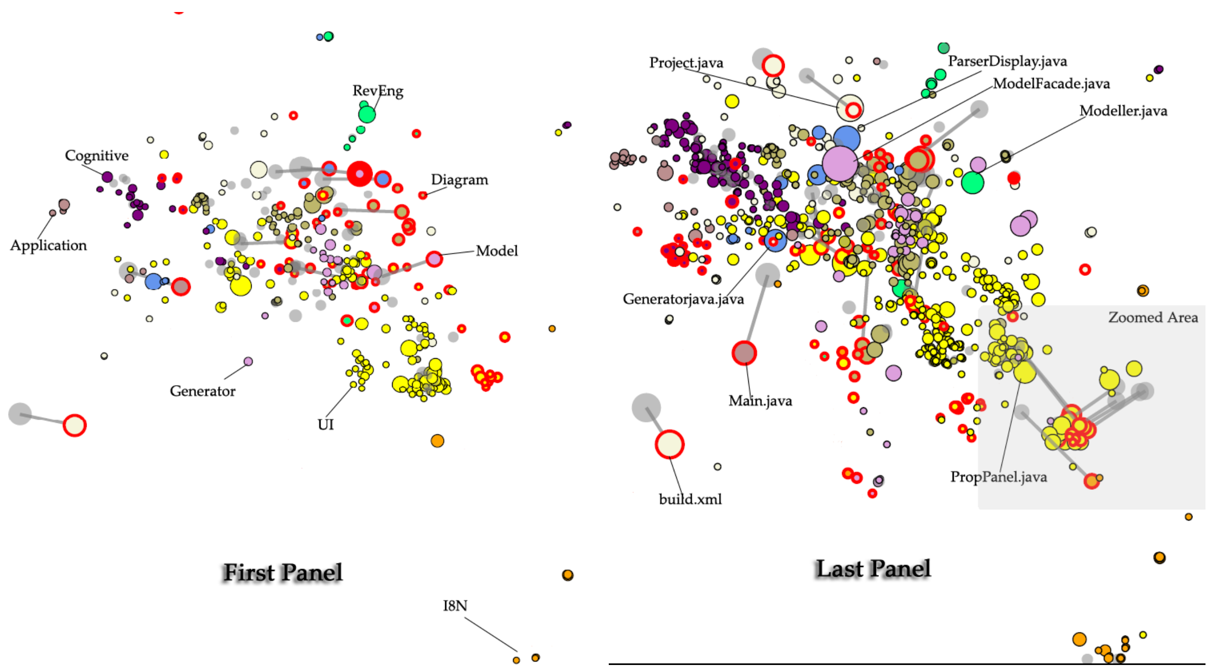

Figure 4. First and last panel (ArgoUML)

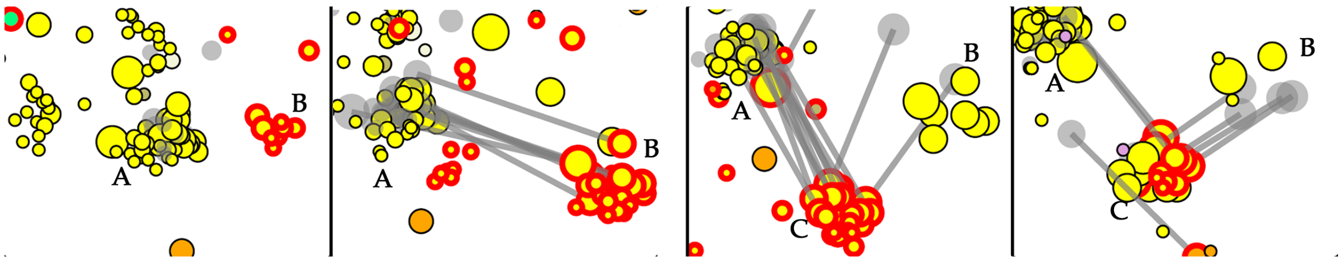

Figure 5. Animated analysis (ArgoUML)

zoomed ARGOUML storyboard at different moments in time (from left to right). We observe the existence of two yellow mini-clusters at the beginning of the project, labeled A and B in the figure. As time progresses, a new mini-cluster C is formed due to many nodes separating from cluster A and B. These separated nodes tend to be modified together and are rarely modified with other artifacts, therefore the energy layout separates them from the rest of the yellow subsystem. Recently (i.e., in the last panel), most of the nodes in B have moved into C and we are back to having two mini-clusters. The storyboard permits us to closely observe the creation of the new cluster C and the slow disappearance of cluster B. At this moment, the reason for the movement and creation of these mini-clusters is still not clear to us.

Our analysis (detailed below) reveals that the storyboard has highlighted a peculiar evolution path of the design of ArgoUML. During our large-node single-panel analysis, we marked the file `PropPanel.java`. This file is the largest file in that area of interest in the storyboard panels. We use this file to start our investigation.

Studying ARGOUML's JavaDoc documentation, we learn that `PropPanel` is an abstract class that provides basic layout and event dispatching support for all property panels. The class has three subclasses, `PropPanelModelElement`, `PropPanelDiagram`, and `TabDocumentation`. The class `TabDocumentation` is a simple class and it is placed close to `PropPanel`. The class `PropPanelDiagram` represents the property panel for a diagram. The classes that implement UML diagrams, such as Use Case, Activities Diagram, inherit from it. Most classes that inherit from `PropPanelDiagram` are placed beside the `Diagram` subsystem classes (at the top of the storyboard panels), except for a few classes which are more UI dependent and are located in cluster A throughout the lifetime of the project.

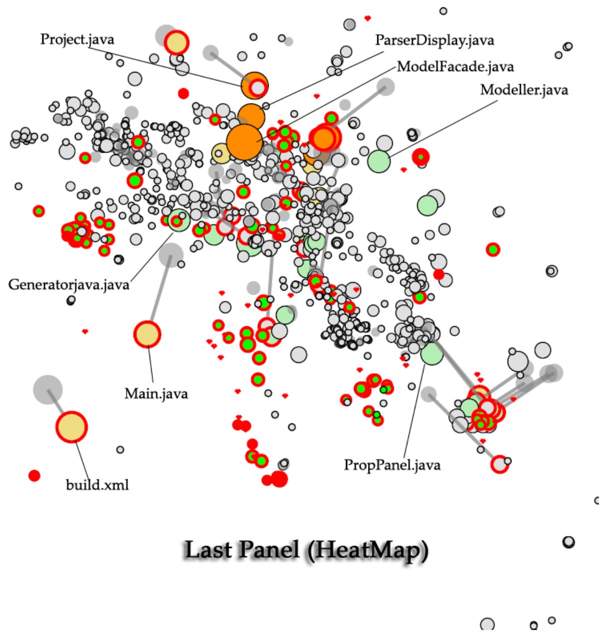

Figure 6. A HeatMap storyboard panel (ArgoUML)

We now focus on the class `PropPanelModelElement`, which holds the key in revealing the rationale for the mini-clusters. The class `PropPanelModelElement` is an abstract class which contains the property panel for a model element. The class `PropPanelModelElement` has several abstract classes that inherit from it. These abstract classes in turn have many concrete classes that implement the desired features. The three mini-clusters that are shown in Fig. 5 are due to movements of abstract and concrete classes in the `PropPanelModelElement` family.

In the left most panel in Fig. 5, cluster B contains most of the concrete classes in the `PropPanelModelElement` family. This is expected as these concrete classes tend to change together since they are related. The abstract classes on the other hand should not be changing frequently since they represent well-defined and stable interfaces. Nevertheless, as the project evolves, the abstract classes need to change and changes to these abstract classes must be reflected in their concrete implementation classes. This evolution of the design is shown in the second panel (from the left) with many of the abstract classes joining cluster B along with any concrete classes that had not joined the cluster yet. Now, in the third panel, the implementation of new functionality, which is likely caused by the changes to the abstract classes, has caused the formation of a new independent mini-cluster (cluster C in Fig. 5). Finally in the last panel, we observe that cluster B is slowly losing many of its nodes to cluster C. The remaining nodes in cluster B are the nodes representing the abstract classes that have again become more stable and are not changing with their concrete implementation classes.

As we noted in our single-panel analysis (Fig. 4), we found several large nodes that we planned to study using our animated analysis. For the animated analysis, we use the HeatMap storyboard for ARGOUML (shown in Fig. 6) and we notice that many of these large files are continuously moving, such as `build.xml`, `ModelFacade.java`, `Modeller.java`, and `Main.java`.

A closer analysis of these files and consulting the project's documentation indicate that these files are responsible for cross-cutting concerns that are spread throughout the source code. Changes to these files cause changes to a variety of other files that are spread across the system. For example, the file `build.xml` is responsible for the build aspect of the software system. The addition of a new file requires updating this file to ensure that the new file becomes part of the build. The class `ModelFacade` acts like an abstraction layer (e.g., facade and interface) to different parts of the system. The implementation for these abstraction layer files is spread throughout the code, which causes these files to change with a variety of files over time, instead of having a consistent set of files which they change with. Even though the file `Modeller.java` is part of the Reverse Engineering (RevEng) subsystem, it interacts heavily with the Model subsystem and hence it tends to oscillate its position between both subsystem based on recent co-changes. The file `Main.java` is the application's main file. The addition of features in any subsystem usually causes changes to this file to enable these features through the command line or through the user interface.

As we began studying the ARGOUML project using the evolution storyboard technique, we had little knowledge about the software system and its historical development. Throughout our case study, we found that the storyboard's animation was valuable in highlighting interesting artifacts that are worth investigating. In this section, we gave a brief overview of a few notable discoveries regarding the evolution of ARGOUML. The evolution storyboard has guided us in understanding a variety of other events in the lifetime of ARGOUML, and other large projects.

5. Conclusion and Discussion

Existing techniques for evolution visualization offer static views of the software history. Static views do not support the understanding of the dynamics of software development and maintenance. We introduced a new visualization technique for software evolution that offers dynamic views. Evolution storyboards emphasize the history of a project using a sequence of panels, each representing a particular period in the life of the project. Each panel shows how artifacts (e.g., classes) change their dependencies over time. The movement of an artifact is an indication of changes to the interdependencies of the artifact with its environment.

Our technique ensures that the visualization is not blurred by animation of unimportant artifacts using a set of simple heuristics. Although we have evaluated the method only for co-change graphs, it is applicable to any dependency relation due to our general dependency graph model.

Several typical reengineering tasks were suggested for which the new technique can provide helpful information, and we outlined how to use the evolution storyboards by providing a guideline. In our case studies, the storyboard visualization revealed many interesting events in the lifetime of the considered large projects. For ARGOUML, we reported how the storyboard highlights the team's migration from the old code tree to the new code tree over time, and other interesting design observations.

We proposed two different artifact coloring schemes for the storyboard. Through our case studies, we showed that the heat-colored layout usually reveals files that implement cross-cutting concerns. These files tend to frequently move over time since they change with a large number of files. The coloring technique based on authoritative decomposition (each subsystem is given a particular color) is helpful in showing how the structure of a software system decayed or remained stable over time.

Performance issues and scalability. The production of one single panel for a large system like ArgoUML takes several minutes of processor time. Therefore, it is not possible to adjust the predefined panel periods interactively and show the resulting storyboards immediately. Also, all kinds of (slider or moving-window based) time-line zooming features are prohibitive expensive for a clustering energy-model based method like ours. However, panels are precomputed ahead of time using an efficient algorithm, and zooming within panels for particular fixed periods is fast.

Evaluation. The storyboard has highlighted to us many unknown and interesting details and facts about several large projects. We believe that the storyboard is helpful in revealing various notable and historical events in the lifetime of a project. However, more detailed case studies would be necessary to prove the benefits of storyboards for the system's developers and reverse engineers, especially including actual developers of the software system to achieve a more authoritative validation of our visualization technique. We believe that the storyboard may prove useful even for experienced developers who might have forgotten various details about the project, and the storyboard can help to remind them of various interesting events.

Online supplement. The static nature of the paper medium in which this article is written may hinder the reader's understanding of the operations on a storyboard. Storyboards for the three systems mentioned in this article are available on the supplementary web page (search the internet for the string "Evolution-Storyboard-Supplementary-Material"). Once the storyboard is loaded, the user can navigate through the storyboard pressing the following keys: '$\rightarrow$': go to next panel, '$\leftarrow$': go to previous panel, 'A': animate the panel (moving bubbles emphasize changes to the location of nodes), 'P': pause or resume the animation, 'C': switch between HeatMap and authoritative decomposition coloring schemes. A user can use the viewer's zoom features in a single panel and all the other panels will be updated to reflect this new view. If the user holds either the '$\rightarrow$' or '$\leftarrow$' button, the sequence appears in a movie-like animation as the storyboard rapidly switches between the different panels.

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] M. Begleiter. *From Word to Image: Storyboarding and the Filmmaking Process*. Michael Wiese Productions, 2001.

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

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

[6] 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.

[7] 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.

[8] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. ICSM*, pages 23–. IEEE, 2003.

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

[10] 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.

[11] J. Hart. *The Art of the Storyboard*. Focal Press, 1998.

[12] A. E. Hassan, R. C. Holt, and A. Mockus. Proc. MSR, 2004.

[13] S. Katz. *Film Directing — Shot by Shot : Visualizing from Concept to Screen*. Michael Wiese Productions, 1991.

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

[15] A. Noack. Energy-based clustering of graphs with nonuniform degrees. In *Proc. GD'05*, LNCS 3843, pages 309–320. Springer, 2006.

[16] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. MSR*, pages 2–6, 2004.

[17] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng.*, 31(6):429–445, 2005.