Software Evolution Storylines

Michael Ogawa*
VIDI Lab
University of California, Davis

Kwan-Liu Ma†
VIDI Lab
University of California, Davis

ABSTRACT

This paper presents a technique for visualizing the interactions between developers in software project evolution. The goal is to produce a visualization that shows more detail than animated software histories, like code_swarm [15], but keeps the same focus on aesthetics and presentation. Our software evolution storylines technique draws inspiration from XKCD's "Movie Narrative Charts" and the aesthetic design of metro maps. We provide the algorithm, design choices, and examine the results of using the storylines technique. Our conclusion is that the it is able to show more details when compared to animated software project history videos. However, it does not scale to the largest projects, such as Eclipse and Mozilla.

Categories and Subject Descriptors

H.5.m [**Information Interfaces and Presentation**]: Miscellaneous

General Terms

Design

Keywords

Software Visualization, Software Evolution, Storylines

1. INTRODUCTION

Animated visualizations of version control system (VCS) data, such as code_swarm and Gource [15, 5], have appeared in recent years. Though characterized as non-analytical, they have been useful for presenting the scale of development to viewers not familiar to the particular project, as well as project-specific events to those who are. A problem with animation is that, while it provides an overall mental picture of a project's evolution, the glimpses into the data at individual timesteps are fleeting.

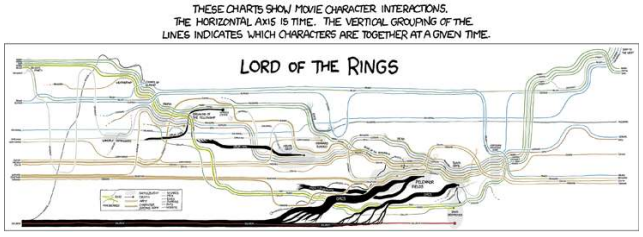


Figure 1: Exceprt from the XKCD comic "Movie Narrative Charts" [14].

Considering these animations, we wondered if the same information could be represented in a static image, so that a viewer can see all timesteps at once. This approach could potentially add analysis capabilities, since a viewer is able to explore an image at leisure.

Previously, we found that the most important aspect of the data to present was the developer interactions. Our visualization needs to show how developers work together on a software project over time. In our search for techniques that accomplish this, we came across the webcomic XKCD's "Movie Narrative Charts" [14] (an excerpt is shown in Figure 1). This chart shows the interactions between movie characters as lines going from left to right. When characters are in the same film location, their lines are grouped together. The XKCD chart was drawn by hand, and it is likely that considerable time was spent planning the layout.

So the question is: Can we apply the idea of a narrative line chart to software project data to produce visualizations automatically? In this paper we explain the technique and show the results of software evolution storylines.

2. THE STORYLINES TECHNIQUE

In this section we describe the data processing requirements, visual design and layout, and results of the technique.

2.1 Data Processing

The raw input data for our visualization comes from software projects' version control systems. There are many version control systems that projects may choose from, such as CVS, Subversion, Git, Mercurial, Bazar, and so on. Each of these systems have their own log output format. Therefore, being able to read the histories of different projects will likely result in having to parse different log formats.

*msogawa@ucdavis.edu
†ma@cs.ucdavis.edu

SOFTVIS'10, October 17–21, 2010, Salt Lake City, Utah, USA.

In order to minimize the complexity of programming for all the different systems and their log formats, we use one separate format. This common XML format is the same as the one accepted by code_swarm, with data on each commit, its author, time and modified files. The parser is also separate from the visualization program.

2.2 Layout Heuristic

From studying the XKCD chart, we derived general rules for an aesthetic layout of lines:

1. Clustered developers must be placed in contiguous, adjacent lines.
2. Clusters should be spaced apart from each other.
3. Existing tubes should change y-position very little, if at all.
4. Tube crossings are inevitable, but avoid them if possible.

What follows is a concise description of the algorithm. Some details, such as specific data structures, have been left out. (We emphasize that this paper is about the *design* of the visualization technique and not about its implementation. The heuristics changed much during the prototyping process and there is no reason to believe that this is the *best* way to achieve the desired layout.)

```
Divide the y-axis into discrete tube slots.
(e.g. A 100-pixel axis may be divided into
twenty slots each five pixels high.)
For each timestep in the dataset,
  Obtain a set of clusters of that timestep's
    developers through the co-commit graph.
  For each cluster,
    Based on already-placed developer tube
    positions, find the weighted average
    y-position (desired position) of the developers.
    Add the cluster and its desired
    position to a list.
  Sort the list of clusters by desired position.
  For each cluster in the list,
    Attempt to assign it its desired place,
    s.t. most existing lines do not shift position
    and it is spaced away from already-placed
    clusters.
    If the desired place is taken,
      search around the desired place until a
      suitably empty area is found.
    For each "veteran" developer in the cluster,
      Place developer as close to their previous
      timestep position as possible.
    For each "newcomer" developer in the cluster,
      Place where there is an empty tube in the
      cluster's defined region.
```

This is for the general case, where there a previous timestep has been laid out. For the intial case, we simply place the first timestep's developers in spaced-out positions centered on the middle of the y-axis.

The timestep duration we use is one month. This unit was chosen by experimenting with different durations and seeing what effect they had on the visualization. Shorter durations lead to developers appearing to drop in and out of the project, while longer durations lead to everyone appearing in one large cluster.

Notice that we do not specify the specific clustering method used for clustering each timestep's developers. It is not important, so long as the clustering makes sense from a data perspective. In our prototype implementation, we use JUNG's [22] Weak Component Clusterer algorithm. It was chosen because it does not require an arbitrary edge removal number or set number of clusters to generate. On the other hand, it tends to generate large superclusters with software project-specific data.

We considered other possible classes of algorithm for this layout problem.

Force-Directed Graph Layout.

One may consider the developer storylines as a graph. Each developer appearance in a timestep is a node, and nodes are connected in the x-direction by author and connected in the y-direction by co-commits with other authors. By constricting the node positions to their respective timesteps on the x-axis, but leaving them free to move in the y-direction, we can use a force-directed graph layout algorithm to attempt to reduce the number of edge crossings. This idea was not followed because of the force-directed algorithms' tendency to settle in local minima. For aesthetic reasons, we wanted to keep the distance between inter-cluster lines uniform, which a force-directed algorithm cannot guarantee.

Genetic Algorithm.

One may also consider the developer storyline positions as a combinatoric problem. At each timestep there is an ordering of lines such that there is a minimum of changed positions considering the previous and next timestep orderings. A genetic algorithm can be used to find an efficient ordering across all timesteps, where there is a minimum of changed positions and line crossings. This is the fitness metric. We chose not to use this class of algorithm because of the programming complexity, but it is a good candidate for future work.

Although some reasons are given for not using these particular algorithm classes, that is not to say that they will not provide a good solution. We only wanted to produce a proof of concept and using a heuristic approach was the fastest. We encourage others to explore the results from force-directed, genetic, and other algorithms.

2.3 Visual Elements

Tubes and Colors.

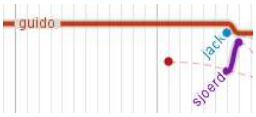
Initially, we tried to copy the pencil line drawing style of the XKCD chart. That is, the developer lines were drawn as colored thin curves separated by whitespace. While that style worked for the movie chart, it did not scale for the larger amount of lines and crossings. The thinness of the lines made it difficult to follow each line, as in the "hairball" problem that network visualizations experience.

Instead of thin lines, we were inspired by *metro maps* (i.e. schematic diagrams of public transportation routes) to thicken the lines and use bold colors. The amount of space between connected lines was decreased, to mimic the metro map convention showing collinear routes.

The default color scheme for storylines is a hash of a developer's VCS username, transformed into a hue. An alternative scheme is to color each storyline by its developer's combined file modifications, using a user-defined map of file type to color. As an example, a core developer may be colored red while a lead documenter may be colored blue.

Labels.

Each developer tube is labeled with their repository username. There are two types of labels: angled and inline. Angled labels appear at the beginning of most tubes. They are angled at 45 degrees to distinguish them from the mostly-horizontal tubes. In-line labels appear inside the tubes when there is enough horizontal room. The appearance of these labels is controlled by a minimum spacing distance, to ensure that they do not become too cluttered. (Although in times when there are many new developers, as seen in Figure 3, the cluttering is unavoidable.)

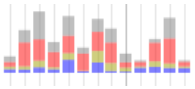


Furlough Lines.

A fairly common occurrence in open source development is when a developer does not commit during a timestep, but resumes work in a future one. These furloughs from activity ought to be visually differentiated from a developer who leaves the project permanently. We use dashed lines to connect develoeprs' timesteps during their temporary absence (pictured below). The XKCD chart uses these lines before a character's first appearance to imply their prior locations are unknown.

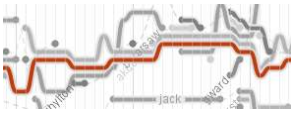
Commit Histogram.

As the storylines indicate the number of developers but not the amount of commits, we place a commit histogram at the bottom. This shows the number of file-commits (the sum of files in each commit). Each bar is one timestep and is divided into color categories, defined by the user. In the histograms in this paper, the colors are red for core source code, yellow for modules, and blue for documents.



Interaction.

The storylines visualization is designed to be viewable as a static image. However, unlike metro maps which usually have under fifteen lines, the number of developers in a given project may number in the tens or hundreds. The potentially large number of developers can lead to visual clutter and difficulty tracking individual developer storylines. In our prototype implementation, when the user mouses over a storyline, only that developer's storyline is colored and the rest are turned to grayscale (pictured below). This significanly reduces the visual burden and allows the user to easily follow a developer's activity. In addition, the selected developer's activity in the commit histogram is highlighted through this interaction.



2.4 Results

We produced visualizations for a variety of open source projects. These include Python, Apache, Ant, Eclipse and PostgreSQL. The results for the Python dataset can be seen in Figures 2 and 3. We do not include high-resolution images of other project results in this paper for space reasons, but we will make them available online at `http://vis.cs.ucdavis.edu/~ogawa/storylines`. Implementation was done in the Processing[1] language. It displays an interactive visualization window and can export to PNG and PDF formats. (We are currently working on a feature to export to SVG so that web users can experience the interactive elements.)

Looking at the result for Python (Fig. 2), we see that Guido (van Rossum, in red) starts the project in 1990. He commits alone for two years, then is joined by a few other developers. They "weave" in and out, sometimes working together, sometimes not. For Jack (Jansen, in blue), this pattern is understandable because he created the MacPython port. From the histogram, we see that documentation is given more attention right before 1994 and at the beginning of 1995.

What we did not expect to see, based on code_swarm's depiction, is Fred Drake connected to Guido most of the time. As the lead documenter, we expected Fred to be separate from the core developers *and* not work on any code. By interacting with the visualization, we find that Guido commits a fair amount of documentation and that Fred commits a fair amount of code. This was not obvious in the Python code_swarm video[2].

In the year 2000, a large number of newcomers enter the project. This did not happen organically; the project migrates to SourceForge and many people are now able to access the VCS. This pattern again happens in 2005, when the project migrates to Subversion.

Post-migration to SourceForge, development activity is dominated by a large core cluster with small, specialist contributions on the periphery. The Python code_swarm video also showed the large core, but the peripheral contributions were lost in the flurry of activity.

2.4.1 Scale

We found that the storylines method works best with small- to medium-sized projects. Apache, Python, and PostgreSQL can be considered medium-sized. Larger projects, like Eclipse, Mozilla and Linux, create problems for cluster separation, which in turn creates problems for readability and aesthetics. We think that when a project has a large

[1] http://processing.org

[2] `http://vis.cs.ucdavis.edu/~ogawa/codeswarm`

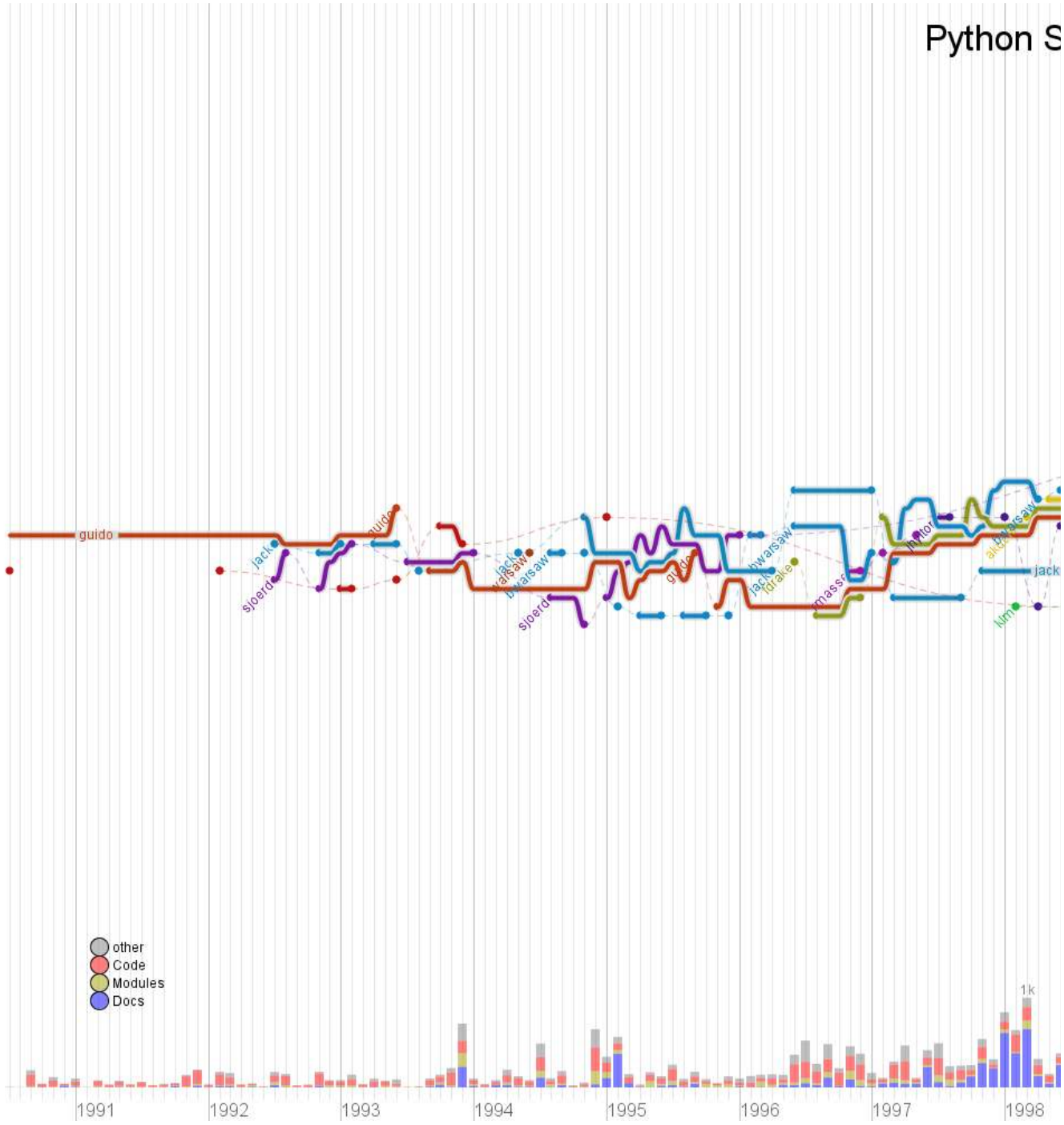


Figure 2: Python's Storylines (left half). Guido van Rossum (red) begins the project in 1990. He is joined later by a few developers. Development often fluctuates between cooperation and separation.

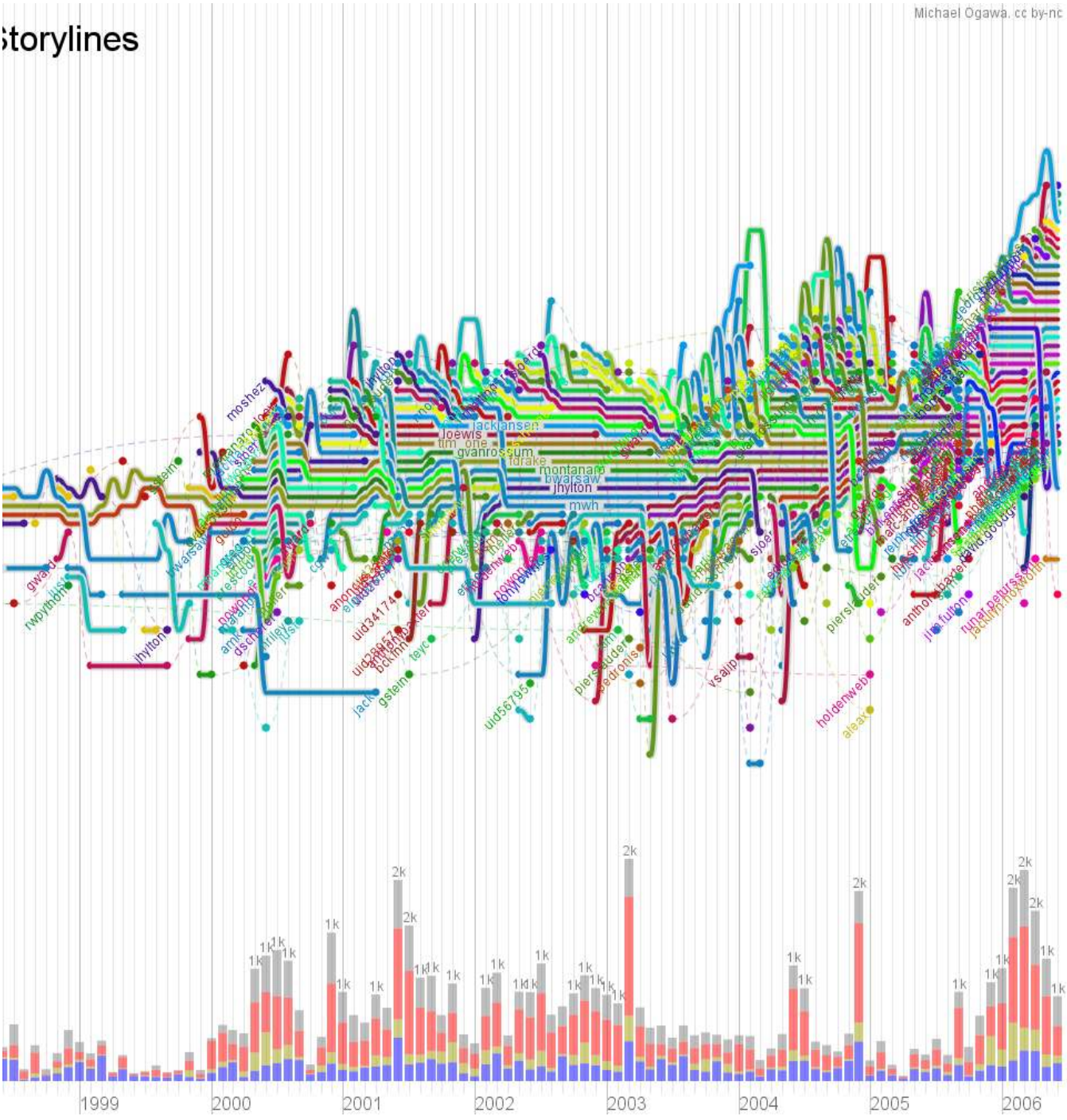


Figure 3: Python's Storylines (right half). Project hosting is moved to SourceForge (CVS) in 2000 and development speed picks up. They migrate to Subverion in 2005. There are plans to migrate to a distributed VCS (likely Mercurial) in 2010.

number of developers, they are more likely to "overlap" each other's file modifications. Since our method relies on file modifications to determine developer clustering, the developers are grouped into one large cluster. While it may be the ground truth that everyone is connected to everyone else in a particular project, it does not make for an interesting visualization rich with insights.

As an example, the Eclipse storylines result (Fig. 4) contains large "ribbons" of developers. The result is not particularly engaging, as everyone seems to be doing the same thing as everyone else.

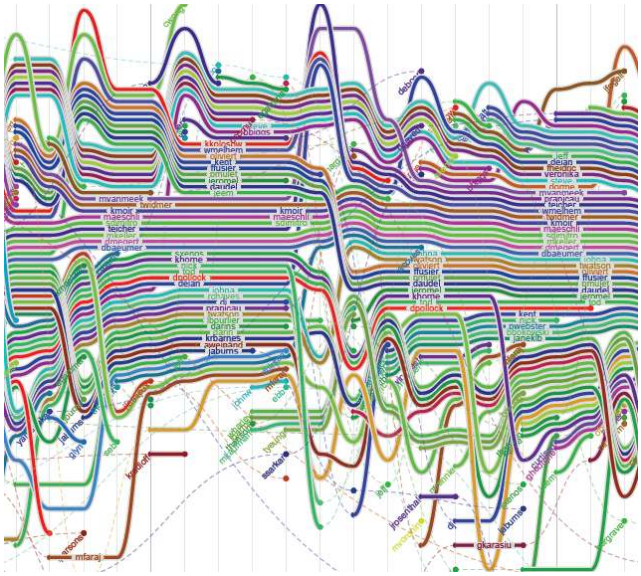


Figure 4: A small section of the Eclipse Storylines result, showing around 50 active developers per timestep. Large, single clusters are a problem when visualizing large software projects. We believe this is because having more developers in a project increases the likelihood of overlapping file changes.

2.4.2 *Timing*

The time to generate these visualizations is not problematic. The Python result takes twenty seconds to produce. A larger project like Eclipse takes roughly two and a half minutes. This time includes loading and processing the data, computing the layout and displaying the result. Interaction with the visualization is real-time, because all layout information is saved in memory. Processing was done on a Windows 2.4 Ghz dual core laptop with Sun's 32-bit Java VM.

3. RELATED WORK

As mentioned previously, we were motivated by our work on code_swarm [15] and inspired by the XKCD chart [14]. This section examines other related work.

Other visualizations have plotted developer activity on a timeline. A handfull of papers explaining the Visual Code Navigator suite [11, 20, 19, 21] use a History Flow-like [18] approach to show changes in sets of files and lines of code. The Ownership Map [6] represents files as straight lines that are colored according to the calculated owner of the file. The activity of only about twelve distinct developers can be shown due to color perception limitations. Code Flows [17], more recently, uses a tube aesthetic to show changes between versions of files. These above visualizations differ from Storylines in that their primary, spatial focus is on the files or code while ours is on the developers and their relationships.

ThemeRivers [7] and Streamgraphs [4] have thus far not been applied to software evolution, but are worth mentioning. They show some measure of frequency of popularity of things over time, such as political text, movie box office returns, and music listening habits. Using stream width to represent some magnitude may be applicable to our Storylines, though the tube aesthetic may be lost.

The project hosting site Github [1] has an "Impact" visualization as part of their project graphs collection. It shows a timeline of developer contributions, sorted by amount. Developers are differentiated by color, sized by contribution amount, and connected across timesteps. However, it does not show any relationships between developers.

Like the Github visualization, Lee et al.'s system [9] uses sized connected bars to show trending topics on the del.ici.ous website. Unlike Github, relationships between topics are shown as adjacent clusters. The "general view" also allows topic clusters to overlap.

"Temps," by Bestiario and Arts Santa Monica [2], is a Flash-based visualization of scientific publications. Its lower view shows trending topics in science from year to year. It does not, however, cluster topics or order them in an aesthetic way.

Metro Maps as Visualization Metaphors.

Metro Maps, first designed for the London Underground in 1931 by Harry Beck, have been used as a visualization metaphor in the past.

Burkhard et al. and Stott et al. [3, 16] use the metro map metaphor to display a timeline for project tasks. In the first paper, layout and drawing is done in Adobe Illustrator, but the second paper computes the layout automatically. Martinez et al. [13] use the same metaphor to show more detailed aspects of the software development process. Unlike [3] and [16], there are no parallel routes. These visualizations are different from ours because they are meant to guide future project organization and development rather than show the project history.

There are also more *artistic* metro maps, such as one of the human body [10], web trends [8], and music [12]. These maps are manually laid out.

4. DISCUSSION AND CONCLUSION

We have presented a technique for visualizing software developer interactions as a single image. The technique, called software evolution storylines, shows many features of software project histories. It uses data from the VCS, which is readily available and inherent in software projects.

While it has advantages over animated techniques, such as being able to drill down and discover more details about developers (demonstrated in Section 2.4), it does not show complete details like the exact names of files committed, number of lines changed, or the intents of developers. We would like to incorporate more information in the design, such as commit messages, bug reports and mailing list topics. The problem is difficult, as space is limited and there can be thousands of messages to distill.

Aesthetically, we would like to see various groups of developers forming along the lines of their separated file assignments, occasionally coalescing and reforming. The XKCD chart has this aesthetic because epic movies do not often feature all characters on screen at once. We discovered software development is a more practical process, where the separation of assignments is not followed closely at all. This leads to the large cluster of developers seen in the latter half of the Python storylines (Figure 3) and throughout the Eclipse and Mozilla projects. To alleviate this monocluster problem, a more liberal clustering algorithm can be used, or one where a minimum number of partitions is set. Fortunately, the majority of open source projects are smaller in scale than Eclipse or Mozilla.

We would like to see the results of different classes of layout algorithm, as featured at the end of Section 2.2. Can force-directed and genetic algorithms produce aesthetically pleasing results in a timely manner?

Finally, an evaluation is necessary to determine if this visualization technique engages casual viewers. Will it have the same response as code_swarm and Gource, or will viewers be underwhelmed by its static nature? Will people familiar with the projects find more insights than before? To do a public evaluation with interactive features intact, we will need to program for the web (Flash or Javascript), a process that will take some time. After releasing the visualization to the public, we will collect data from users' online feedback, both spontaneous and surveyed. If public feedback is positive, we will prepare and release the code as open source.

5. ACKNOWLEDGMENTS

This research was supported in part by the U.S. National Science Foundation through grants CCF-0938114, CCF-0808896, CNS-0716691, and CCF-1025269, the U.S. Department of Energy through the SciDAC program with Agreement No. DE-FC02-06ER25777, and HP Labs and AT&T Labs Research.

We would like to thank the SoftVis reviewers who offered their helpful suggestions – chief among these the use of SVG to bring our visualizations to the web. We would also like to thank Tom I. Li for his feedback.

6. REFERENCES

[1] Github. http://github.com.

[2] Bestiario and Arts Santa Monica. Temps. http://culturesdelcanvi.com/temps.

[3] R. A. Burkhard and M. Meier. Tube Map Visualization: Evaluation of a Novel Knowledge Visualization Application for the Transfer of Knowledge in Long-Term Projects. *Journal of Universal Computer Science*, 11(4):473–494, 2005.

[4] L. Byron and M. Wattenberg. Stacked Graphs – Geometry & Aesthetics. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1245–1252, 2008.

[5] A. Cauldwell. Gource. http://code.google.com/p/gource.

[6] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How Developers Drive Software Evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 113–122. IEEE, 2005.

[7] S. Havre, E. Hetzler, P. Whitney, and L. Nowell. ThemeRiver: Visualizing Thematic Changes in Large Document Collections. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):9–20, 2002.

[8] Information Architects, Inc. Web trend map. http://informationarchitects.jp/wtm.

[9] T.-Y. Lee, C. Jones, B.-Y. Chen, and K.-L. Ma. Visualizing data trend and relation for exploring knowledge. In *Posters Proceedings of the IEEE Pacific Visualization Symposium*, 2010.

[10] S. Loman. Underskin. http://www.just-sam.com.

[11] G. Lommerse, F. Nossin, L. Voinea, and A. Telea. The Visual Code Navigator: An interactive toolset for source code investigation. In *Symposium on Information Visualization*, pages 24–31. IEEE, 2005.

[12] D. Lynskey. Music on the tube map. http://blogs.guardian.co.uk/culturevulture/archives/2006/02/03/going_underground.html.

[13] A. A. Martínez, J. J. D. Cosín, and C. P. García. A metro map metaphor for visualization of software projects. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 199–200. ACM, 2008.

[14] R. Munroe. XKCD #657: Movie Narrative Charts. http://xkcd.com/657.

[15] M. Ogawa and K.-L. Ma. code_swarm: A Design Study in Organic Software Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1097–1104, 2009.

[16] J. M. Stott, P. Rodgers, R. A. Burkhard, M. Meier, and M. T. J. Smis. Automatic layout of project plans using a metro map metaphor. In *IV '05: Proceedings of the Ninth International Conference on Information Visualisation*, pages 203–206. IEEE Computer Society, 2005.

[17] A. Telea and D. Auber. Code Flows: Visualizing Structural Evolution of Source Code. *Computer Graphics Forum*, 27(3), 2008.

[18] F. B. Viégas, M. Wattenberg, and K. Dave. Studying cooperation and conflict between authors with history flow visualizations. In *CHI*, pages 575–582. ACM, 2004.

[19] L. Voinea, J. Lukkien, and A. Telea. Visual assessment of software evolution. *Sci. Comput. Program.*, 65(3):222–248, 2007.

[20] L. Voinea and A. Telea. CVSgrab: Mining the history of large software projects. In *EuroVis*, pages 187–194, 2006.

[21] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: Visualization of Code Evolution. In *SOFTVIS*, pages 47–56. ACM, 2005.

[22] S. White, J. O'Madadhain, D. Fisher, and Y. B. Boey. JUNG: Java Universal Network/Graph Framework. http://jung.sourceforge.net.