A File Based Visualization of Software Evolution

Keywords: Software evolution, Software visualization.

S.L. Voinea
l.voinea@tue.nl

A. Telea
alext@win.tue.nl

Technische Universiteit Eindhoven

Abstract

Software Configuration Management systems are important instruments for supporting development of large software projects. They accumulate large amounts of evolution data that can be used for process accounting and auditing. We study how visualization can help developers and managers to get insight in this unstructured history information. To this end, we propose several new techniques for visual mining of software evolution. Central to our approach is a file-based evolution visualization, where each project is shown as a set of horizontal stripes depicting files along the time axis. We propose several mechanisms for interactively building layouts in this display, and for correlating the evolution with the results of various software metrics. We demonstrate the usefulness of our approach on real- life data sets.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.7 [Software Engineering]: Maintenance, Enhancement; H.5.2 [User Interfaces]: Evaluation, Methodology

1 Introduction

Software Configuration Management (SCM) systems are an essential ingredient of effectively managing large-scale software development projects. A main feature of a SCM system is that it maintains a history of changes done in the structure and contents of the managed project. This serves primarily the very specific goal of navigating to and retrieving a specific version in the evolution of the project. However, information maintained by SCM systems enable also many scenarios that fall outside the above very precise goal. The intrinsically maintained system evolution information is an excellent starting point for empirically understanding the software development process and its structure. One area that can benefit from this information is the software maintenance of large projects.

During the maintenance phase of most projects, appropriate documentation misses or is 'out of sync' with the actual code. In such cases, code evolution information maintained by a SCM system (when such a system is used) is the one and only up-to-date reference material available. Effective use of this information can greatly help maintainers understand and manage the evolving project.

In this paper, we propose a set of new techniques for visually assessing the entire evolution of software projects using the evolution information contained in SCM systems. Typical questions we target with our techniques are:

- What is the project-wide activity, i.e. when have been files created, modified, and by who, and how did this activity evolve during the project?
- Which are the project areas of high(est) activity?
- How are development tasks distributed among the programmers?
- Which are the project files that belong and/or are modified together?
- How well does the project conceptual and functional organization match the actual folder structure?

We validate our techniques by implementing them in a tool, CVSgrab, which seamlessly combines SCM data extraction with analysis and visualization.

The structure of this paper is as follows. In Section 2, we review previous work on software evolution visualization. Section 3 outlines the data model we use for the software projects to be visualized. Next, we detail the visual layout mechanisms we use for our evolution visualizations and for correlating them with other results of project evolution analysis. Section 4 presents several use-cases that illustrate the use of our approach for investigating the evolution of industry-size projects. Section 5 summarizes our contribution and outlines open issues for future research.

2 Related work

The research community has only recently acknowledged the huge potential of the information stored by SCM systems as a starting point for empirical studies on software development. The massive growth in popularity and use of SCM systems, influenced by open source projects like CVS and Subversion, opens new possibilities for project accounting, auditing and understanding. Efforts have been focused so far in two research directions: data

mining and data visualization. *Data mining* research focuses on processing and extracting relevant information from the evolution data stored into SCM systems. However, most data mining approaches work by trying to fit an existing 'data model' on the raw information stored by the SCM systems, which is fine if the model is correct and exactly what the user wants to see, but may be of limited use otherwise. Many techniques have been proposed to offer access to higher level, aggregated information about the project evolution [3, 4, 7, 11, 12].

Data visualization, the second research direction, takes the different path of making the large amount of evolution information effectively available to the user. Visualization techniques use a 'weak' data model, as the goal is to let the user discover patterns and trends by himself rather than hard-coding such models in the mining process. Visualization tools try to present data in a way that is as intuitive and familiar as possible to users. SeeSoft [2] is one of the first tools to visualize software change. It uses a direct 'code line to pixel line' visual mapping and color to show code fragments that match given modification requests. UNIX's `gdiff` and its Windows version WinDiff visualize code differences between just two versions of a given file by drawing the line insertions, deletions, and modifications found by the `diff` tool. Such tools can reveal the line-based structure of software systems and change dependencies at given moments in time. However, they do not provide insight into code attributes and higher-level structural changes made throughout an *entire* project with hundreds of versions of thousands of files. Recent efforts try to overcome these limitations. The CVSscan tool [8] offers comprehensive overviews on the evolution of single, or few, files. Code lines are mapped to pixel lines, as in SeeSoft [2]. Next, file versions are arranged along the time axis to visualize evolution. In this way, CVSscan can detect change dependencies inside a small number of files, but doesn't allow correlations across *large* projects. Lanza visualizes in [5] project evolution at class level using a version uniform sampling of the time axis. Classes are drawn as variable size rectangles, laid out one below the other in a vertical stripe in alphabetical order. Closely related, Wu *et al*. [10] visualize evolution of entire projects at file level using a time uniform axis, and focus on the moments of evolution. Such methods scale well for industry-size systems and provide insightful evolution overviews. Still, they do not offer an easy way to find the artifacts that have a similar evolution.

We propose here a set of visualization technique that extend the work mentioned so far and enable evolution correlations across complete projects. We introduce a new mechanism for interactive building of layouts that supports a visually driven data-mining approach to answer the evolution assessment questions stated in Section 1.

3 Visualization model

We use the assumption that developers are comfortable with visualizing code in the same spatial context in which they construct it [2]. Software maintenance is mainly done at code level, so we use a 2D code-centric approach to visualize the software evolution, as in [2, 8]. As a new element, we interactively present the entire evolution of complete projects on one screen. This enables actively using visualization for mining the history of software projects.

3.1 Data model

We use data from the CVS version control management system, one of the most popular SCMs available. However, our data model is generic to all structure-based SCM systems. The central element is a *repository* R that stores all versions of all NF files in a project:

$$R = \{F_i \mid i = 1..NF\}$$

Each file F_i is defined as a set of NV_i versions:

$$F_i = \{V_{ji} \mid j = 1..NV_i\}$$

A version is a tuple containing several *attributes*: the unique version id, the time when it was committed to the repository, the author who committed it, a log message and its source code:

$$V_{ji} = \langle id, time, author, message, code \rangle$$

The first four elements (id, time, author, and message) are unstructured attributes. The code can be structured in different ways, e.g. a set of lines, or set of functions, classes, modules, or other grammar constructs.

3.2 Visualization techniques

The approaches in [5] and [10] are the only ones we are aware of that scale well for visualizing the evolution of industry-size projects. Both techniques use a fixed vertical ordering of the entities (classes and files). This ordering does not specifically help finding entities with similar evolution. We propose a novel approach for visualizing complete projects with a flexible entity layout that can be interactively modified by users to suit specific analysis scenarios.

Similar to [10] we visualize complete projects at file granularity level. Every file is drawn as a fixed

height horizontal stripe made of several segments (Figure 1). Each segment corresponds to a version of that file. Segments are ordered according to creation time and their length is scaled with the lifetime of the respective version. Segments can be colored to show various data. First, we can show the *author* that committed the respective version by mapping the author id to a unique hue (Figure 1 top). This helps evolution correlations based on both activity and the authors' network. Alternatively, color can show the *state* of the version in the context of a complete project, i.e. *file not created yet*, *before last version, last version*. (Figure 1 bottom). This supports evolution correlations based on activity only, but provides simpler image that focus specifically on activity events. For both alternatives, we use geometric shaded cushions [9] to emphasize the version segments and segregate between vertically stacked file stripes. Also, we draw the commit moments themselves as thin vertical yellow lines between the version cushions.

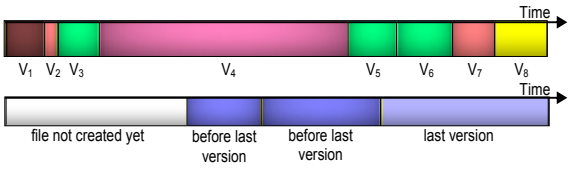

Figure 1: *File evolution representation. Color encodes user identity (top) and activity (bottom)*

We build complete project visualizations of software evolution by stacking individual file stripes on the vertical axis so they share the same time scale and use the same color encoding. In contrast to [5] and [10], we do not fix the vertical axis ordering, but allow (and encourage) users to interactively change the layout to target specific analysis needs. We describe next two mechanisms to achieve this goal: *sorting* and *clustering*.

Sorting allows identifying how a relevant project metric is distributed across a set of files. Files are ordered along the vertical axis according to that metric's values. We propose several such metrics: creation time (similar to [10]), alphabetic order (similar to [5]), activity measure (i.e number of versions), and evolution similarity measure. The last metric works as follows: given some file of interest (the focus), we measure the similarity *S* between its evolution and that of all other files (the context).

To define *S,* we introduce first the notions of commit neighborhood N_K and evolution correspondent τ. Let V_1 be the set of commit moments for all versions of a file F_1 and V_2 be the set of commit moments for all versions of a file F_2. Then $N_K : V_1 \to V_2^*$ is a mapping that assigns to each element t of V_1 a set of elements $V_2^* \subseteq V_2$ that are in a time vicinity of K time-units from t:

$$N_K(t) = \{u \mid u \in V_2, |u - t| < K\}$$

$\tau : V_1 \to V_s \cup \{\infty\}$ is a mapping that assigns to each element t of V_1 the minimum element from $N_K(t)$, if such an element exists, or ∞ (infinity) otherwise:

$$\tau(t) = \begin{cases} \min N_K(t) & N_K(t) \text{is not empty} \\ \infty & \text{otherwise} \end{cases}$$

We define now the evolution similarity $S(F_1,F_2)$ of files F_1 and F_2 as the symmetrized sum of inverses of the time difference between all commit moments in a file and their evolution correspondents in the other file:

$$S(F_1,F_2) = \frac{1}{|V_1|}\sum_{i=1}^{|V_1|}\frac{1}{|t_i - \tau_1(t_i)|} + \frac{1}{|V_2|}\sum_{j=1}^{|V_2|}\frac{1}{|t_j - \tau_2(t_j)|},$$

where $t_i \in V_1$, $t_j \in V_2$, τ_1 is the evolution correspondent from V_1 to V_2, and τ_2 is the evolution correspondent from V_2 to V_1. This measure says that files that are changed at similar moments, are more similar than others from an evolution perspective. Using $S(F_{\text{ref}}, F)$ permits us now to sort all files F according to the relevance (i.e. connection) they have with respect to a given reference file F_{ref}. Why would this assumption be true? The underlying idea, which can be checked as correct in many large software projects, is that files which depend on each other, either via explicit data or call structures or otherwise, must (and will) be changed together to maintain the desired system invariants. Thus, change similarity is correlated with interface or implementation interdependencies.

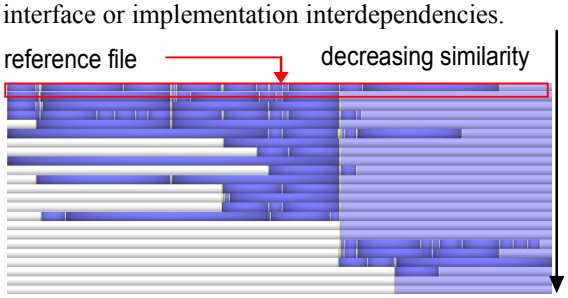

Figure 2: *Sorted files layout based on a similarity measure*

Figure 2 shows an example of the proposed similarity measure used to sort files on the vertical axis. The evolution of 23 files is colored by activity, as described for Figure 1. Yellow lines show commit moments. The topmost file is the reference file F_{ref}; chosen by the user, the other files are vertically sorted on decreasing similarity with respect to F_{ref}. This image allows us to easily find the files that a have a similar evolution with the reference one.

The second generic mechanism we propose for interactive building of layouts is the *clustering* operation. Clustering enables finding groups of strongly related files, i.e. files that have similar computed properties. Two issues must be addressed here. First, we must provide a meaningful similarity measure. Second, we must provide a method for grouping similar files. We use the same similarity measure described before for the sort mechanism, and a bottom-up agglomerative clustering based on average link to group similar files [1]. We start with the individual files and recursively group the two most similar ones in a cluster, until a single cluster is obtained, creating thus a cluster tree. When a new cluster is constructed, it collects all the commit events of its two children. After the tree is constructed, the user can choose to draw the clusters e.g. at some given depth from the root, i.e. view the project at the desired 'level of detail'. Although our clustering may be more computationally intensive than other techniques, e.g. k-means [1], it provides a simple, automatic and deterministic way to identify similar entities.

We visualize the clustering results using colored and shaded cushions. Clusters are rendered as semitransparent rectangles atop of the file stripes, textured with plateau cushions [6]. We use alternating hues, e.g. blue and red, for neighbor cushions. Due to the semi transparency of the cushions, these hues blend with the file stripes (Figure 3, right). The alternating hues effectively help visual segregation of clusters depicted by cushions. For example, Figure 3 compares cluster cushions with and without alternating hues.

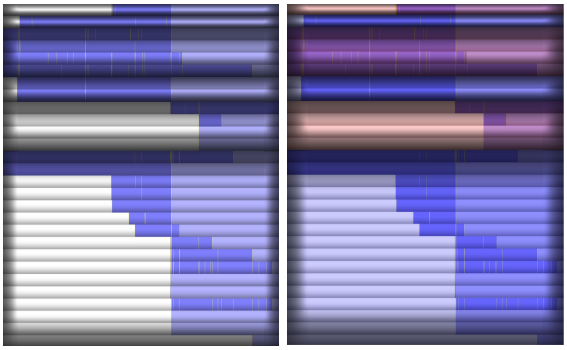

Figure 3: *Cluster segregation: plateau cushions without (left) and with alternating hues (right)*

However, alternating hues alone may not be sufficient for visual segregation. When a rich color encoding is used for the file stripes, e.g. the author-id color encoding, we must minimize its interference with the cushion hues. A too soft cushion hue blending over richly colored file stripes yields a poor visual separation of clusters in the border regions.

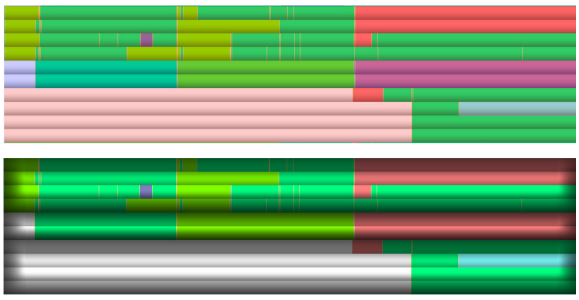

Figure 4: *Cluster segregation: color blending only (top), plateau cushions (bottom)*

Figure 4 presents a relevant example. It depicts the evolution of 10 files with color-encoded author-id. Three clusters are also shown, the first one containing the first four files, the second containing the following two, and the last containing the remaining four. Figure 4 top uses a color-only blending scheme to segregate between clusters. However, the visual transitions between clusters are not obvious. One could easily interpret the color change as author-id change and not as another cluster. In contrast, Figure 4 bottom uses plateau cushions and one can now easily identify the three clusters.

By combining sort and cluster operations, we can interactively build visualizations of project evolution that suit specific analysis needs, as illustrated next.

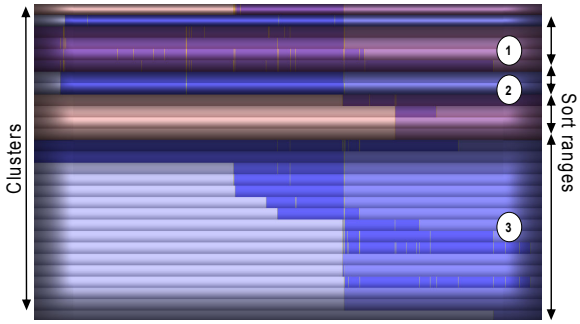

Figure 5: *Interactively built layout using sort and clustering operations*

Figure 5 shows the evolution of 28 files from a real project (the *FreeDesktop*) using such an interactively built layout. The described alternating blue-red hue blending and plateau cushions are used to segregate clusters. Files are colored on activity: white (i.e. pink or light blue after hue blending) = *file not created yet*, dark blue (dark blue or magenta after hue blending) = *regular version*, light blue (light blue or magenta after hue blending) = *last version*. Yellow lines show commit moments. Six clusters emerge, each containing files with similar evolution. Within each cluster, files are sorted according to their creation time. This image immediately shows files with similar behavior. The strongly related files in cluster

1 are: *Glyph.c, Picture.c, Xrender.c, Xrender.h*. At detailed inspection, we discovered that these files contain code of the project's image generation engine. This confirms the correlation between similar evolution and conceptual similarity.

A second important finding is that files with a strongly coupled evolution, i.e. clusters 1 and 2, have also a similar creation time and this time is close to the project beginning. Files that are created later seem to be less connected (cluster 3). This may be an indication that the system's core functionality, developed in the beginning of the project, is found in clusters 1 and 2. Concluding, the interactively built layout in Figure 5 enables user-driven cross-project correlations based on similar evolution and the creation time metric. Such correlations do not address only the development process, but as illustrated by this example, they may bring insight also in the structure and organization of the project.

The interactive layout technique we propose enables the user to combine clustering with a refined sort operation, i.e. equal values in one sort criterion may be further ordered using another metric, to adapt the visual mining process to specific needs. Useful correlations can be obtained by comparing the results of different sort operations.

To further extend the correlation capabilities of our interactive layout in this direction, we use *metric views,* i.e. narrow information bars that decorate the main evolution visualization area. These views use simple encoding techniques, e.g. 1D graphs and color maps, to show one-dimensional metric data in a very small space. To enable correlations, metric views share their main axis with one of the axes of the main visualization. *Vertical views* visualize per-file computed quantities, and *horizontal views* visualize time-related, per-project metrics. Concretely, in the vertical metric view we show the various metrics used for sorting, i.e. the file creation time, alphabetical order, activity measure, and similarity with respect to a reference file. In the horizontal metric view, we visualize the project-wide activity measure, i.e. total number of files updated in a given period. Figure 6 shows the evolution of 68 files from a large project (the *VTK library*) using the same color encoding as in Figure 2, i.e. activity based, and sorted on creation time. The vertical metric view shows the file activity as a 1D bar graph. The horizontal metric view shows the project wide activity. By correlating the main layout with the vertical metric view, we see that file creation time does not fully determine the file activity. Two activity hotspots are identified. They correspond to groups of files that appeared later in the project but had high activity, so they might contain important and/or problematic functionality. We validated this hypothesis against the knowledge of an expert VTK user, and it proved to be consistent with the reality. Concluding, the correlation of the interactively built layout with the metric views enables the user to easily construct pertinent hypotheses about the qualitative aspects of a project based on its evolution.

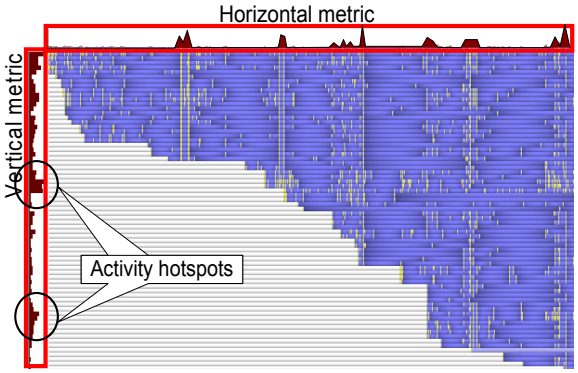

Figure 6: *Metric views. Vertical: file activity. Horizontal: project- wide activity*

While these do not immediately guarantee a valid system assessment, they represent a solid starting point for further investigation and facilitate understanding process during the maintenance phase of software projects.

4 Exploration Scenarios

We analyzed the use of CVSgrab for mining the history of several industry-size projects. Here we present the results of one such exploration for the VTK library, an open source project of over 2700 files written by 40 developers in over 11 years. The project was mined by three experienced C++ developers having, however, no VTK knowledge. They participated first in a 15-minute training in which the functionality of CVSgrab was explained on a small example project, with several generic use cases that could be easily reproduced on other input data. Next, they mined the history of VTK for 2 hours. Finally, their findings were assessed by a developer with over eight years of VTK experience.

Figure 7 depicts various annotated visualizations of the complete project evolution obtained during the study using sort operations. Figure 7.a, 6.c, and 6.d color files on activity, as detailed in Sec. 3.2. Yellow lines show commit moments. Figure 7.b colors files on author id, every hue being a separate author. In Figure 7.a files are sorted alphabetically. Although cluster cushions are not rendered, a vertical metric view (C) shows the clusters to which files belong, using color mapping. The alphabetical sorting of files uses the full pathname and thus nicely groups together files in the same folders. By mouse brushing the evolution area, the users easily identified the major folders of the project, highlighted in (A):

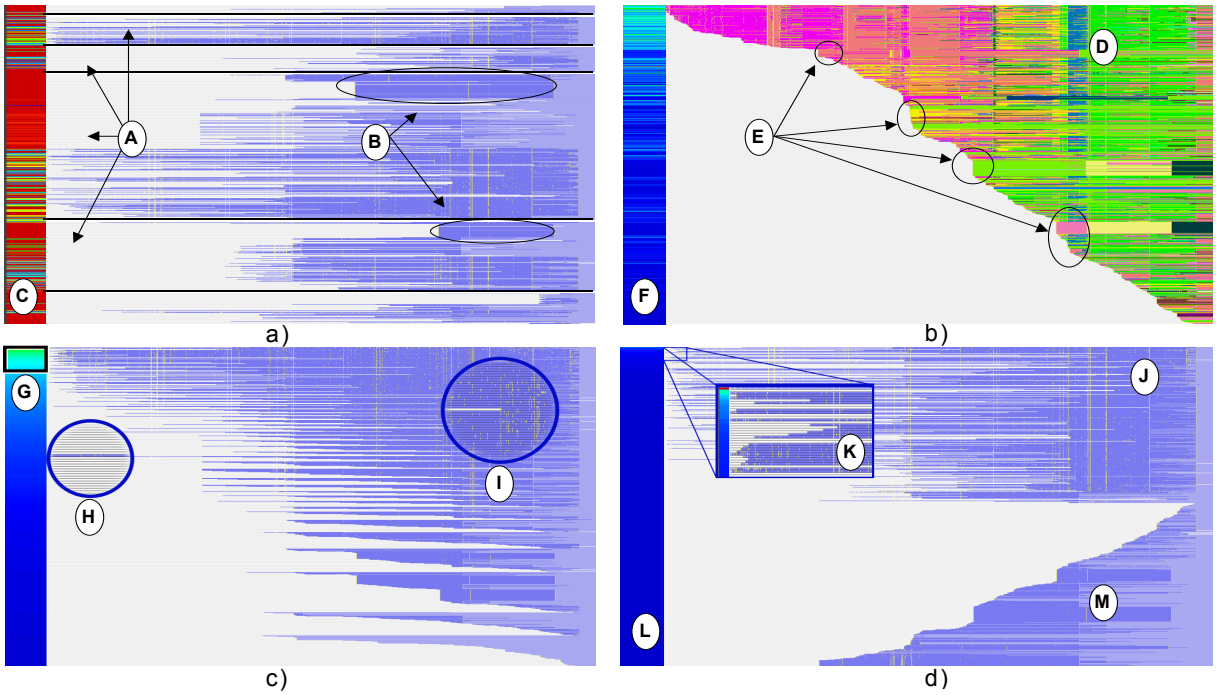

Figure 7: *Interactive built layouts using sort operations*

Imaging, *Graphics*, *Contrib*, and *Common*. Two compact, low-activity evolution regions were also spotted (B). By brushing the corresponding evolution area, the users discovered, via the status bar information, that they refer to VTK code examples in Python.

The vertical metric view (C) helped the users conclude that the project's functional organization does not correspond entirely to its file hierarchy. By sorting on creation, the users found several possible moments of so-called punctuated evolution (E), i.e. moments when large code changes took place in a short time. Via the details-on-demand features of CVSgrab they refined their hypotheses. Of the four moments highlighted in the image (E), three refer to the addition of VTK examples, and just one involves heavy changing of the library functionality. Further, the vertical metric (F) has no smooth transitions. This suggested there is no direct correlation between creation time and file activity. Indeed, the project contains both files that were introduced early but recorded little activity, e.g. stable interfaces and/or implementations, and files that where introduced later but were frequently updated, e.g. problematic and/or unstable implementations. In Figure 7.c files are ordered according to their recorded activity. The vertical metric view (G) depicts also the activity measure using a rainbow color map (red = *high activity*, blue = *low activity*). From this image, the users concluded that most development is concentrated in less than 10% of all files (G). Figure 7.c was also useful to find the activity outliers. The highlighted inset (H) depicts an example of an early outlier, i.e. a stable file during evolution: *vtkRender.h*. The highlighted inset (I) depicts a late outlier, i.e. a file introduced later, but often updated: *vtkDataObject.cxx*. Finally, in Figure 7.d, files are arranged according to their similarity with respect to a selected reference file: *vtkIntArray.cxx*. The vertical metric view (L) uses a rainbow colormap to depict the similarity measure (red = *very similar*; blue = *very different*). The users concluded that the chosen reference file had little in common with most of the other files in the project, as the metric view is almost entirely blue. In the magnification caption (K) a zoomed-in region of the evolution area (J) is displayed. This revealed a small number of files that had a higher similarity value. Via the details-on-demand mechanism the users discovered their identity: *vtkLongArray.cxx, vtkFloatArray.cxx, vtkBitArray.cxx*, etc. Indeed, detailed inspection confirmed these files have a tightly coupled implementation. The files depicted in region (M) are arranged in decreasing order of their creation time. They represent actually files that have no similarity with the reference one and are sorted according to a secondary criterion: creation time.

In Figure 8 the evolution of the complete VTK project is displayed using sort and clustering operations. An activity-based color encoding is used, as in Figure 7.a. The three users relied on the filtering mechanism of CVSgrab to interactively adjust the number of displayed clusters in order to get the desired visual granularity level. In Figure 8 top, from left to right, we display clusters that give a level of detail of 5%, 40%, and respectively 50% of the entire system. Files are sorted in alphabetical order. The proposed filtering mechanism shows here a known

drawback of agglomerative clustering: it generates highly unbalanced trees. This causes both large (A) and very small clusters (C) to coexist, and it is difficult to assess them together. This can be corrected, if desired, by modifying the tree render traversal to return clusters having some size balancing constraints. Still, useful investigations can be done using the actual traversal. On path (A) the users observed that one part of the system behaved like a cluster seed of highly connected files that grew in a large cluster. This part seemed to contain (a part of) the core of the VTK library, which the users localized in the *Graphics* folder. Using the details-on-demand feature, the users found that the cluster seed contains mainly interfaces for a number of rendering related classes (e.g. *vtkVectorDot.h, vtkLineSource.h, vtkWarpTo.h*). On path (B), the users observed that a large part of the system appears to have a separate evolution with respect to the core. Via the details-on-demand mechanism the users discovered this part contains mainly usage examples in three programming languages: C, Tcl, and Python. Further, a subset of the Tcl examples seems to have a different evolution then the rest. Again, at closer inspection of the files themselves and their respective comments, the users could indeed confirm that the examples were structured in a different way, and had a different evolution, from the main core of the VTK proper. In Figure 8, bottom, zoomed-in areas of evolution are displayed at cluster granularity levels that give a 32% (left) and 33% (right) level-of-detail of the entire system. Files are arranged in order of their creation time. The clusters highlighted in (D) and (E) seem to have an interesting evolution. Via details-on-demand, the users discovered that cluster (D) has two evolution 'roots': one that groups generic data description and modeling classes of VTK, e.g. *vtkImageData.cxx, vtkDataObject.cxx*, and one that contains the implementation of various grid classes, e.g. *vtkStructuredGrid.cxx, vtkRectilinearGrid.cxx*. Cluster (E) contains the implementation of various array classes, e.g. *vtkFloatArray.cxx, vtkIntArray*. These suggested that the implementation of the array classes is closely connected to the VTK dataset classes. Indeed, this supposition was confirmed by the experienced user.

At the end of this study, we summarized the three users' observations and checked them again with the knowledge of the expert developer, who validated the largest part of the observations as fully correct. Moreover, he was particularly surprised by the ease with each the users identified the main developers behind VTK, and the problematic / active areas of development, without *any prior knowledge* about this code. One aspect he found interesting was the higher-than-expected ratio between the 'project core' and rest of the project activity. He identified also one misinterpretation concerning the authors. He declared that *schroede* and *will* refer actually to the same contributor (Will Schroeder, one of the parents of the VTK project) who changed his username about mid-project. Still, this correctly matches with the users' observation that *will* replaces *schroede*.

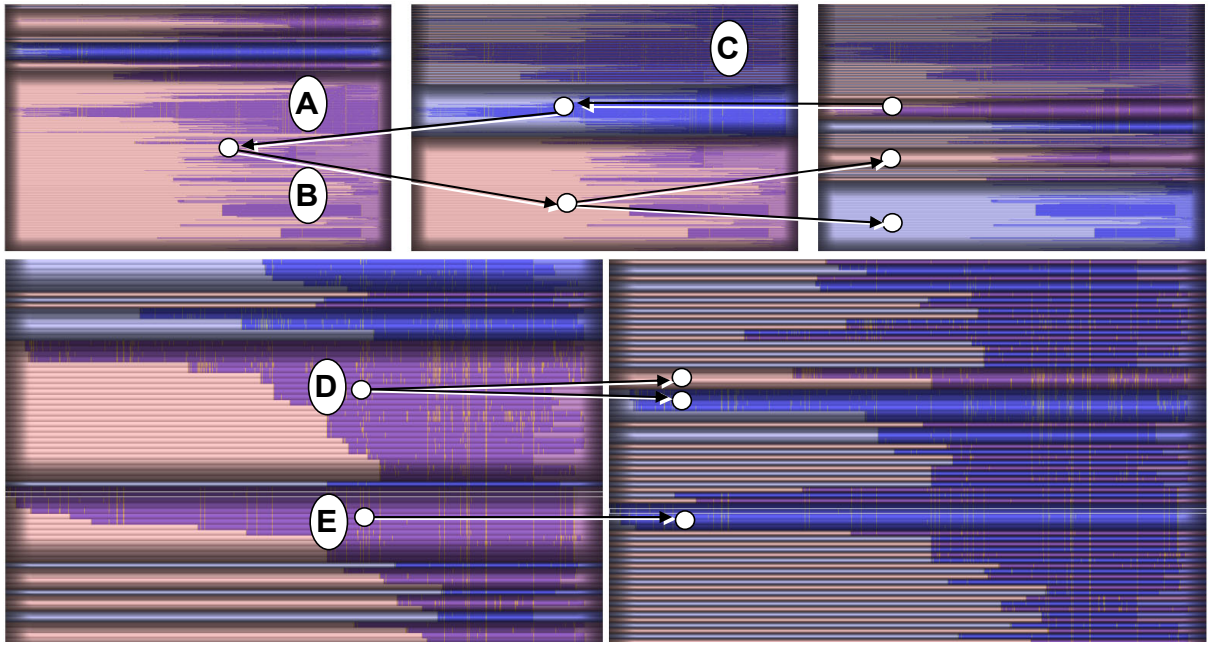

Figure 8: *Visualization layouts: clustering and alphabetical sort (top row); clustering and sort on creation time (bottom row)*

5 Conclusions

This paper presents a set of visualization and interaction techniques that support history mining of large-scale software projects. Our goal is to enable developers and project managers in the software maintenance community to visually and interactively explore the evolution of software projects in a way that facilitates the system and process understanding.

We propose a novel technique for interactive layout of file evolution representations, by interactively mixing and adjusting sort and cluster operations to direct the visual mining towards specific goals. We enable evolution correlations based on more sort criteria at the same time, by adding horizontal and vertical metric views. We propose a simple-to-use, yet powerful clustering technique that reduces the project visualization to a user-specified number of clusters with files having similar evolutions. This targets queries such as "show the whole project split into *n* similar components". We reduce the interference between the cluster rendering and file colors using a mixed cluster luminance and hue encoding. This combines the visual comfort of hue-based cluster segregation with the precision of the plateau cushions in the boundary regions.

We validate the proposed techniques by implementing them in CVSgrab, a visual tool for exploring the evolution of industry–size projects. The dense pixel visualization combined with interactively built layouts makes it possible to navigate and assess code projects beyond the size of what is possible by similar tools [8] or with better insight [5, 10]. For example, we can get a comprehensive overview of the complete evolution of the VTK project (2700 files, 40 developers, over 11 years, about 100 versions for active files) in five screens, with quite little interaction. True, CVSgrab does not allow visualizing code at line level. For this, other tools, such as [8] are best used. CVSgrab's main strength comes when one does not know where (and why) to zoom in, given a large software project of many versions. Secondly, the evolution-based similarity sorting and clustering proposed here can be effectively used to discover relations between files in a project that are not apparent, *without* needing to use more the complex, slower, language-specific parsing of the files' contents.

We plan to extend our approach with more sort criteria and different, more effective, similarity measures. The visual encoding of clusters should be improved to cope with the unbalanced cluster trees, e.g. by simultaneously displaying clusters with different similarity levels. Another challenge is to visualize and enable correlations across multiple version attributes at the same time. Our final aim is to create a fully featured code visualization and analysis toolset, and make it available to the software development and maintenance community.

References

[1] EVERITT E., LANDAU S., LEESE M.: *Cluster Analysis*. Arnold Publishers, 2001

[2] EICK S.G., STEFFEN J.L., SUMNER E.E.: SeeSoft - A Tool for Visualizing Line Oriented Software Statistics. *IEEE Trans. on Software Engineering*, 18(11), 1992, IEEE CS Press, 1992, pp. 957 – 968

[3] FISCHER M., PINZGER M., GALL H.: Populating a Release History Database from version control and bug tracking systems. *Proc. ICSM*, IEEE CS press, 2003, pp. 23 – 32

[4] GALL H., JAZAYERI M., KRAJEWSKI J.: CVS release history data for detecting logical couplings. *Proc. IWPSE*, IEEE CS Press, 2003, pp. 13–23

[5] LANZA M.: The evolution matix: Recovering software evolution using software visualization techniques. *Proc. Intl. Workshop on Principles of Software Evolution*, ACM Press, 2001, pp. 37–42

[6] LOMMERSE G., NOSSIN F., VOINEA S.L., TELEA A.: The Visual Code Navigator: An Interactive Toolset for Source Code Investigation. *Proc. IEEE InfoVis*, IEEE CS Press, 2005, pp. 24 – 31

[7] MOCKUS A., FIELDING R.T., HERBSLEB J.D., Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3), ACM Press, 2002, pp. 309–346

[8] VOINEA L., TELEA A., VAN WIJK J.J.: CVSscan: Visualization of Code Evolution, *Proc. ACM SoftVis*, ACM Press, 2005, pp. 47 – 56

[9] VAN WIJK J.J, VAN DE WETERING H.: Cushion Treemaps: Visualization of Hierarchical Information. *Proc. IEEE InfoVis,* IEEE CS Press, 1999, pp. 73-78

[10] WU J., SPITZER C.W., HASSAN A.E., HOLT R.C., Evolution Spectrographs: Visualizing Punctuated Change in Software Evolution. *Proc. IWPSE*, IEEE CS press, 2004, pp. 57-66

[11] ZIMMERMANN T., WEIßGERBER P.: Preprocessing CVS Data for Fine-grained Analysis. Presented at *MSR 2004*, available online at: http://www.st.cs.uni-sb.de/papers/msr2004/msr2004.pdf

[12] ZIMMERMANN T., WEIßGERBER P., DIEHL S., ZELLER A.: Mining version histories to guide software changes. *Proc. ICSE*, IEEE CS Press, 2004, pp. 429 – 445