Available online at www.sciencedirect.com
ScienceDirect
Computers & Graphics 31 (2007) 410–428
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Visual Analytics

Visual data mining and analysis of software repositories

Lucian Voinea, Alexandru Telea*

Department of Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

In this article we describe an ongoing effort to integrate information visualization techniques into the process of configuration management for software systems. Our focus is to help software engineers manage the evolution of large and complex software systems by offering them effective and efficient ways to query and assess system properties using visual techniques. To this end, we combine several techniques from different domains, as follows. First, we construct an infrastructure that allows generic querying and data mining of different types of software repositories such as CVS and Subversion. Using this infrastructure, we construct several models of the software source code evolution at different levels of detail, ranging from project and package up to function and code line. Second, we describe a set of views that allow examining the code evolution models at different levels of detail and from different perspectives. We detail three views: the *file view* shows changes at line level across many versions of a single, or a few, files. The *project view* shows changes at file level across entire software projects. The *decomposition view* shows changes at subsystem level across entire projects. We illustrate how the proposed techniques, which we implemented in a fully operational toolset, have been used to answer non-trivial questions on several real-world, industry-size software projects. Our work is at the crossroads of applied software engineering (SE) and information visualization, as our toolset aims to tightly integrate the methods promoted by the InfoVis field into the SE practice.
© 2007 Elsevier Ltd. All rights reserved.

Keywords: Data mining; Software evolution; Software visualization; Software engineering; Maintenance

1. Introduction

Software configuration management (SCM) systems are an essential ingredient of effectively managing large-scale software development projects. Due to the growing complexity and size of industry projects, tools that automate, help and/or enforce a specific development, testing and deployment process, have become a “must have” [1].

An SCM system maintains a history of *changes* done in the structure and contents of the managed project. This serves primarily the very precise goal of navigating to and retrieving a specific version in the project evolution. However, SCM systems and the information they maintain enable also a wealth of possibilities that fall outside the above goal. The intrinsically maintained system evolution information is probably the best starting point for empirically understanding the software development process and structure. An important reason for this is that SCM systems are mainly used to store source code, which is widely recognized as the “main asset of the software engineering (SE) economy” [2]. Whereas documents and strategies easily become out-of-sync with the real system, source code is one of the best sources of information on the actual changes a system underwent during its evolution.

One of the main areas that can benefit from this information is the software maintenance of large projects. Industry surveys show that, in the last decade, maintenance and evolution exceeded 90% of the total software development costs [3], a problem referred to as the *legacy crisis* [4]. It is, therefore, of paramount importance to bring these costs down. This challenge is addressed on two fronts, as follows. The *preventive* approach tries to improve the overall quality of a system upfront, at design time. Many tools and techniques exist to assess and improve the design-time quality attributes [5,6]. However, the sheer dynamics of the software construction process, its high variability, and the quick change of requirements and

*Corresponding author. Tel.: +31 40 247 5008; fax: +31 40 246 8508.
E-mail addresses: l.voinea@tue.nl (L. Voinea), alext@win.tue.nl (A. Telea).

0097-8493/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2007.01.031

specifications make such an approach either cost-ineffective or even inapplicable in many cases. Increasingly popular software development methodologies, such as extreme programming and agile development [7], explicitly acknowledge the high dynamics of software and thus fit the preventive approach to a very limited extent only. The *corrective* approach aims to facilitate the maintenance phase itself, and is supported by program and process understanding and fault localization tools [8–10]. In most projects, however, appropriate documentation often lacks or it is "out of sync" with the implementation. In such cases, the code evolution information maintained in an SCM system (assuming such a system is used) is the one and only up-to-date, definitive reference material available. Exploiting this information in depth can greatly help the maintainers to understand and manage the evolving project.

In this paper, we propose an approach to support the corrective maintenance of software systems based on visual assessment of software evolution information contained in SCM systems. Central to our approach is the tight integration of software visualization in the traditional SE pipeline as a means to get insight of the system evolution and to guide both the analysis and the corrective maintenance tasks. In this paper we mainly concentrate on the visual analysis component of the SE pipeline and show how software evolution visualization can be used to perform non-trivial assessments of software systems that are relevant during the maintenance phase. We target quantitative, query-like questions such as "which are the files containing a given keyword?", data mining and reverse engineering-like questions such as "which is the decomposition of a given code base in strongly cohesive subsystems?", and also task-specific questions, such as "what is the migration effort for changing the middleware of a component-based system?" For all these question types, we advocate and propose a visual approach with three elements: the questions are posed visually, the answers are output in a visual form, and the visual metaphors used help formulating refined and new questions. We show in detail how we validated our approach by implementing it in a toolset that seamlessly and scalably combines data extraction with data mining and visualization. Our toolset integrates previous work [11–15] on visualizing software evolution and also extends it with a number of new directions which are discussed in this paper.

This paper is structured as follows. In Section 2, we present the role and place of visual analysis in the SE process and outline its relation with data mining. In Section 3 we overview existing efforts in analyzing the evolution information present in SCM systems. Section 4 gives a formal description of the software evolution data that we explore using visual means. Section 5 presents the visual techniques and methods we propose for the assessment of evolution. In Section 6 we illustrate the use of our toolset to perform a number of relevant assessments on several industry-size software projects. Section 7 reflects on the open issues and possible ways to address them.

2. Process overview

Fig. 1 illustrates the traditional SE pipeline. The figure is structured along two axes: phases of the SE process (y) and types of activities involved (x). The upper part shows the "traditional" SE development pipeline with its requirement gathering, design, and implementation phases. If the software and/or the SE process evolve with no problems, this is the usual process that takes place. The analysis phase (Fig. 1 middle) is typically triggered by the occurrence of such problems, e.g. software architectures that are too inflexible to accommodate requirement changes, repeated bugs, long time to next release, and high development costs. Analysis starts with gathering information from the SCM system and structuring it in a multi-scale model of the software evolution that ranges from code lines to functions, classes, files and system packages. Next, two types of activities take place, which attempt to answer several questions about the software at hand. *Data mining* activities target mostly quantitative questions, e.g. "how many bug reports are filed in a given period?" using various software analysis and reverse engineering techniques (see Section 3), and provide focused answers. *Software visualization* activities, the main focus of this paper, are able to target also qualitative questions, e.g. "is the software architecture clean?", by showing the correlations, distributions, and relationships present in complex data sets. The combination of concrete, usually numerical, answers from the data mining and insight provided by the visualization activities have two types of effects. First, decisions are taken on which actions to perform to solve the original problems. In this paper, we focus on corrective maintenance actions such as refactoring, redesign, bug-fixing and iterative development. Second, the analysis results can trigger asking new questions (more specific but also totally different ones). The *visual analysis* loop repeats until a decision is taken on which action to execute.

The above model implies no hard border, but a natural overlap, between data mining and visualization, the quantitative versus qualitative nature of the targeted questions, and the precise demarcation between answers and insight. Yet, data mining is far more often used in practice in SE than software visualization. We believe that this is not due to fundamental limitations of the software visualization usefulness, but rather to weaknesses in visualization (tool and technique) scalability, simplicity of use, explicit addressing of focused questions, and integration in an accepted process and tool chain. In this paper we mainly concentrate on the visual analysis loop and address these claims by showing how visualization can be used to perform nontrivial assessments of software systems that are relevant during the maintenance phase, if the above

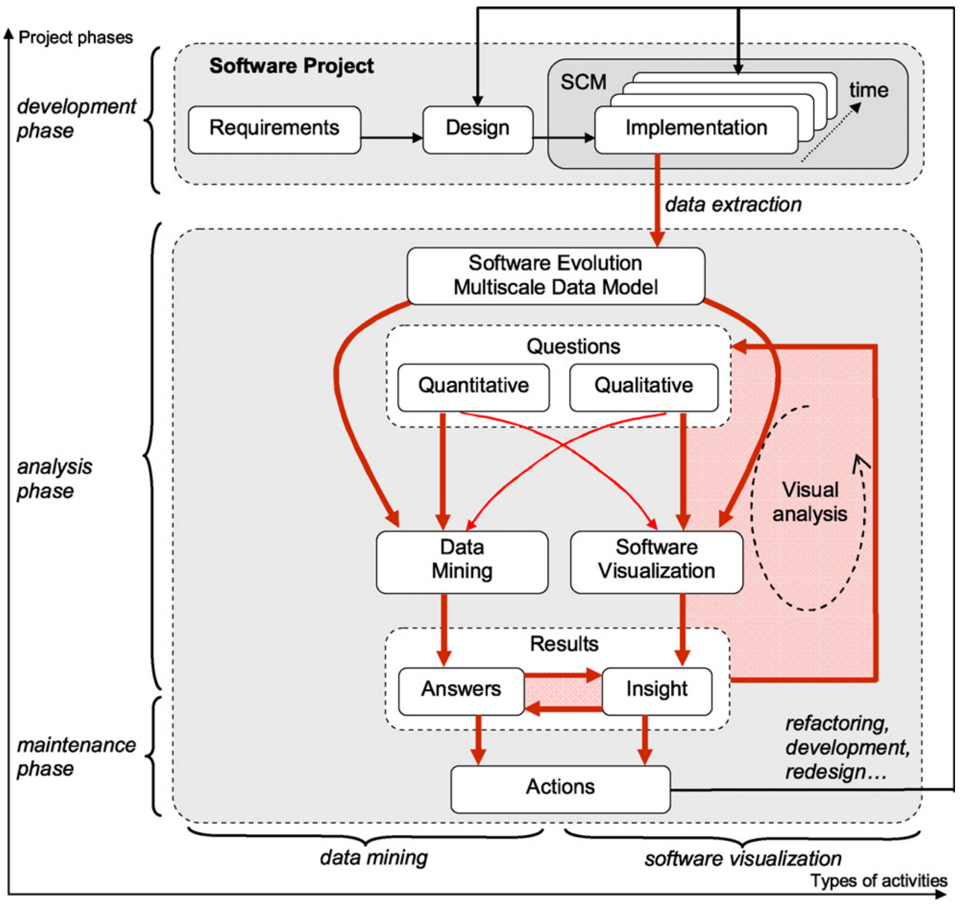


Fig. 1. Software engineering process for corrective system maintenance.

limitations are overcome. Examples of non-trivial and specific questions we target with our approach are:

- What are the structure and the development context of a specific file in a project?
- What is the migration effort for changing the middleware of a component-based system?
- What is the importance of a major change in the source code ?
- How do debugging induced changes propagate to other parts of a system?

In the remainder of this paper, we show how we validated our approach by implementing it in a toolset that seamlessly and scalably combines data extraction with data mining and visualization.

3. Previous work

As explained in Section 2, the analysis phase of the SE process involves data mining and software visualization tools. For this to work in practice, analysis must be coupled with concrete SCM tools such as CVS [16] and Subversion [17], which provide basic management functions, e.g. software check-in, check-out, and branching, and advanced management functions, e.g. bug management, regression testing, and release scheduling. Data mining tools provide data analysis functions, e.g. computation of search queries, software metrics, pattern detection, and system decomposition, all familiar to reverse engineers (e.g. [18–21]). Visualization tools provide various views that let users gain insight in a system or answer targeted questions. These activities can (and should) take place at different *scales* of software representation, e.g. lines of code, functions, classes, files and packages. Software engineers must often quickly and easily change the level of detail at which they work. For example, a developer who edits a function (i.e. works at line level) needs to check what other functions or files are affected (i.e. work at function/file level) or verify if the system architecture is violated or not (i.e. work at component/package level).

All in all, an ideal tool that supports the analysis process in Fig. 1 should address several requirements:

- *management*: check-in, check-out, bug, branch, and release management functions;
- *multiscale*: able to query/visualize software at multiple levels of detail (lines, functions, packages);
- *scalability*: handle repositories of thousands of files, hundreds of versions, millions of lines of code;

- *data mining and analysis*: offer data mining and analysis functions, e.g. queries and pattern detection;
- *visualization*: offer visualizations that effectively target several specific questions;
- *integration*: the offered services should be tightly integrated in a coherent, easy-to-use tool.

Many tools exist that target software repositories. Table 1 shows several popular SCM and software visualization tools and outlines their capabilities in terms of the requirements mentioned above.

Data mining tools focus on extracting relevant facts from the evolution data stored in SCM systems. As SCM systems such as CVS or Subversion focus on basic, "raw" data management, higher-level information is usually inferred by the mining tools from the raw information. In this direction, Fischer et al. propose a method to extend the raw evolution data from SCM tools with information about file merge points [19], Gall [25] and German [20] propose transaction recovery methods based on fixed time windows. Zimmermann and Weigerber extend this work with sliding time windows and information acquired from commit e-mails [34]. Ball [18] proposes a new metric for class cohesion based on the SCM extracted probability of classes being modified together. Relations between classes based on change similarities have been extracted also by Bieman et al. [35] and Gall et al. [25]. Relations between finer grained blocks, e.g. functions, are extracted by Zimmermann et al. [21,24] and Ying et al. [36]. Lopez-Fernandez et al. [26] apply general social network analysis methods on SCM data to characterize the development process of large projects and find inter-project similarities.

Data visualization tools take a different path, by making fewer assumptions about the data than mining tools. The idea is to let the user discover patterns and trends rather than coding pattern models to be searched for in the mining process. SeeSoft [8] is a line based code visualization tool that uses color to show the code fragments corresponding to a given modification request. Augur [28] combines in a single image information about artifacts and activities of a software project at a given moment. Xia [32] uses treemap layouts to show software structure, colored by evolution metrics, e.g. time and author of last commit and number of changes. Such tools are successful in revealing the structure of software systems and uncovering change dependencies at single moments in time. However, they do not show code attribute and structural changes made during an *entire* project. Global overviews allow discovering that problems in a specific part of the code appear after another part was changed. Global overviews also help finding files having tightly coupled implementations. Such files can be easily spotted in a global context as they most likely have a similar evolution. In contrast, lengthy manual cross-file analyses are needed to achieve the same result without an evolution overview. As a first step towards global evolution views, UNIX's `gdiff` and its Windows version WinDiff show code differences (insertions, deletions, and modifications) between two versions of a file. More recent tools try to generalize this to evolution overviews of real-life projects that have thousands of files, each with hundreds of versions. Collberg

Table 1
CVS tools activities and approach overview

Tool	Management activities			Analysis activities		
Name	Basic management	Data analysis	Advanced management	Visualization	Data mining	Multiscale
libcvs	×					File
WinCVS	×					File
javacvs	×					File
Bonsai [22]	×					File
Eclipse CVS plugin	×					File
NetBeans.javacvs [23]	×					File
Release history database [19]	×	×				File
Diff		×				Line
WinDiff		×		×		Line
eRose [24]	×	×			×	Line, function, file
QCR [25]		×			×	File
Social network analysis [26]		×			×	File
MOOSE [27]	×	×				File, class
SeeSoft [10]		×	×	×		Fine, file
Augur [28]	×	×		×		File
Gevol [29]		×		×		Class
CodeCrawler [30]		×		×		File,class
Evolution spectograph [31]		×		×		File
Xia [32]		×		×		File, class, package
SoftChange [33]	×	×	×	×	×	File
CVSscan [11]		×		×	×	Line
CVSgrab [13]	×	×	×	×	×	File, directory, subsystem

et al. [29] visualize software structure and mechanism evolution as a sequence of graphs. Yet, their approach does not seem to scale well on large systems. Lanza [30] visualizes object-oriented software evolution at class level. Closely related, Wu et al. [31] visualize the evolution of entire projects at file level and visually emphasize the evolution moments. One of the farthest-reaching attempts to unify all SCM activities in one coherent environment was proposed by German with SoftChange [33]. Their initial goal was to create a framework to compare Open Source projects. Not only CVS was considered as data source, but also project mailing lists and bug report databases. SoftChange concentrates mainly on basic management and data analysis and provides only simple chart-like visualizations. We have also previously proposed methods for software evolution visualization at different granularity levels: CVSscan [11] for assessing the evolution of a small number of source code files at line level and CVSgrab [13] for project-wide evolution investigations at file level.

A less detailed aspect of SCM data mining and visualization is the data extraction itself. Many researches target CVS repositories, e.g. [11,13,19,24–26,33,36]. Yet, there exists no standard application interface (API) for CVS data extraction. Many CVS repositories are available over the Internet, so such an API should support remote repository querying and retrieval. A second problem is that CVS output is meant for human, not machine reading. Many actual repositories generate ambiguous or non-standard formatted output. Several libraries provide an API to CVS, e.g. the Java package `javacvs` and the Perl module `libcvs`. However, `javacvs` is undocumented, hence of limited use, whereas `libcvs` is incomplete, i.e. does not support remote repositories. The Eclipse environment implements a CVS client, but does not expose its API. The Bonsai project [22] offers a toolset to populate a database with data from CVS repositories. However, these tools are more a web access package than an API and are little documented. The NetBeans.javacvs package [23] offers one of the most mature APIs to CVS. It allegedly offers a full CVS client functionality and comes with good documentation.

Concluding our review, it appears that basic management and data analysis activities seem to be supported by two different groups of tools (Table 1). Also, the data mining and visualization activities (the left and right halves of the pipeline in Fig. 1) have little or no overlap in the same tool. All in all, there is still no tool for SCM repository visual analysis that complies to a sufficient extent with *all* requirements listed at the beginning of this section. We believe this is one of the main reasons for which software evolution visualization tools have not yet been widely accepted by the SE community.

In the remainder of this paper, we shall describe our approach towards an integrated framework, or toolset, for visual analysis and data mining of SCM repositories. We believe that our proposal, which combines and extends our previous CVSscan [11] and CVSgrab [13] tools and techniques, scores better than most existing tools in this area. We describe our approach next (Sections 4 and 5), detail its extensions as compared to previous work [11,13] and present the validation done with several scenarios (Section 6).

4. Evolution data model

In this section, we detail the data model that describes our software evolution data. This model is created from actual SCM repositories using repository query APIs and data mining tools (Section 3).

The central element of a SCM system is a *repository* R which stores the evolution of a set of NF files:

$$R = \{F_i | i = 1, \dots, NF\}.$$

In a repository, each file F_i is stored as a set of NV_i versions:

$$F_i = \{V_{ij} | j = 1, \dots, NV_i\}.$$

Each version is a tuple with several *attributes*. The most typical ones are: the unique version id, the author who committed it, the commit time, a log message, and its contents (e.g. source code or binary content):

$$V_{ij} = \langle id, author, time, message, content \rangle.$$

To simplify notation, we shall drop the file index i in the following when we refer to a single file. The id, author, time and message are unstructured attributes. The content is modeled as a set of *entities*:

$$content = \{e_i | i = 1, \dots, NE\}.$$

Most SCM repositories model content (and its change) as a set of text lines, given that they mostly store source code files. However, the entities e_i can have granularity levels above text lines, e.g. scopes, functions, classes, namespaces, files or even entire directories. We make no assumptions whatsoever on how the versions are internally stored in the SCM repositories. Concretely, we have instantiated the above data model in our toolset on CVS [16] and Subversion [17] repositories as well as memory management profiling log files [37]. Other applications are easy to envisage.

To visualize evolution, we need a way to measure *change*. We say two versions V_i and V_j of the same file differ if any element of their tuples differs from the corresponding element. Finding differences in the id, author, time, and message attributes is trivial. For the content, we must compare the $content(V_i)$ and $content(V_j)$ of two versions V_i and V_j. We make two important decisions when comparing content:

- we compare only consecutive versions, i.e. $|i - j| = 1$;
- we compare content at the same granularity level, or scale.

The first choice can seem restrictive. However, in practice changes in the source code stored in repositories are easiest to follow and understand incrementally, i.e. when we compare V_i with V_{i+1}. Moreover, repositories store such incremental changes explicitly and exactly, so we can have direct access to them. Comparing two arbitrary files is more complex and prone to errors. In CVS, for example, changes are seen from the perspective of a `diff`-like tool that reports the inserted and deleted lines in V_{i+1} with respect to V_i. All entities not deleted or inserted in V_{i+1} are defined as constant (not modified). Entities reported as both deleted *and* inserted in a version are defined as modified (edited). Let us denote by e_{ij} the jth entity of a version V_i, e.g. the jth line in the file V_i. Using `diff`, we can find which entities e_{i+1j} in V_{i+1} match constant (or modified) entities e_{ij} in V_i. Given such an entity e_{ij}, we call the complete set of matching occurrences in all versions, i.e. the transitive closure of the `diff`-based match relation, the *evolution* $E(e_{ij})$ of the entity e_{ij}. This concept can be applied at any scale, as long as we have a `diff` operator for entity types on that scale. In Section 6, we shall illustrate the above concepts at the line, component, and file granularity levels. We next detail the techniques used to map the data model described in this section on visual elements.

5. Visualization model

We now describe the visualization model we use to present the evolution data model described in the previous section. By a *visualization model*, we mean the set of invariants of the mapping from abstract data to visual objects. Our visualization model (Fig. 2) is quite similar with the classical "visualization pipeline" [38]. Its three main elements are the *layout*, *mapping*, and *user interaction*. It is well known in scientific and information visualization that the effectiveness of a visualization application is strongly influenced by decisions taken in the design of this mapping [38,39]. We detail here the design decisions, invariants, and implementation of these elements and explain them in the light of the requirement set presented in Section 2.

5.1. Layout

Layout assigns a geometric position, dimension and shape to every entity to be visualized. We choose upfront for a 2D layout. Our need to display many attributes together may advocate a 3D layout. Yet, we had problems in the past with getting 3D visualizations accepted by software engineers [10]. A 2D layout delivers a simple and fast user interface, no occlusion and viewpoint choice problems, and a result perceived as simple by software engineers. In particular, we opted for a simple 2D orthogonal layout that maps time or version number to the x-axis and entities (lines, files, etc) to the y-axis (Fig. 2). Finally, entries are shaped as rectangles colored by the mapping operation (see Section 5.2). Within this model, several choices exist:

- *selection*: which entities from the complete repository should we visualize?
- *x-sampling*: how to sample the horizontal (time) axis?
- *y-layout*: how to order entities e_{ij} (for the same i, different j) on the vertical axis?
- *sizes*: how to size the "rows" and "columns" of the layout?

Selection allows us to control both what subset of the entire repository we see, and also at which scale. We have designed several so-called *views*, each using a different selection and serving a different purpose: the code view (Section 5.4), the file view (Section 5.3), the project view (Section 5.5) and the decomposition view (Section 5.6).

The horizontal axis can be *time* or *version* sampled. Time sampling yields vertical version stripes (V_i in Fig. 2) with different widths depending on their exact commit times. This layout is good for project-wide overviews as it

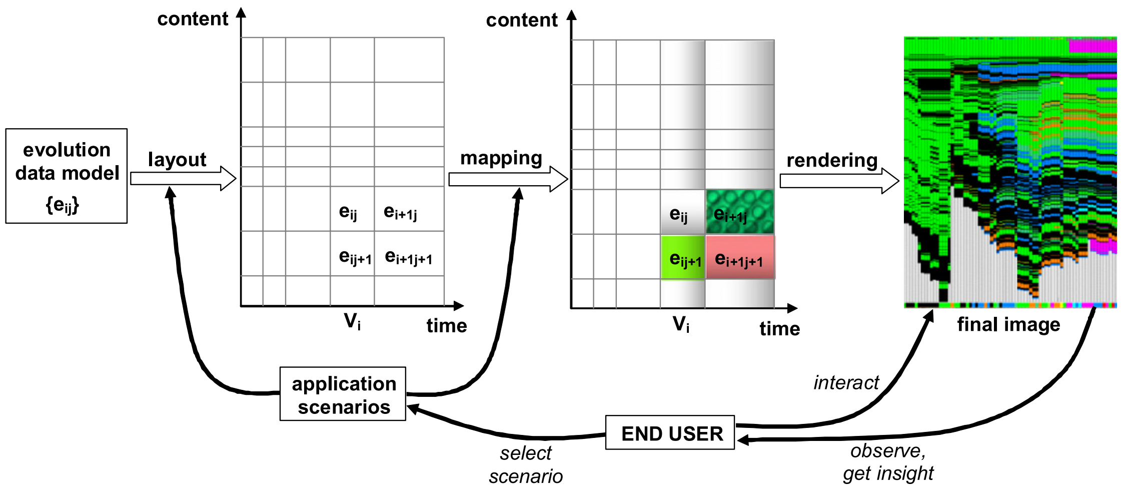


Fig. 2. Generic visualization model for software evolution.

separates frequent-change periods (high activity) from stable ones (low activity). However, too quick changes may result in subpixel resolutions. The project view (Section 5.5) can be set to use this layout. Version sampling uses equal widths for all version stripes. This is more effective for entities that have many common change moments, e.g. lines belonging to the same file [11]. The file view (Section 5.3), uses this strategy by default.

The vertical axis shows entities e_{ij} in the same version V_i. Two degrees of freedom exist here: first, we can choose in which order to lay out the entities e_{ij} for a version. Second, we can stack the entities one above each other or use vertical empty space between entities. Both choices are detailed in Sections 5.3 and 5.6.

5.2. Mapping

Mapping specifies how entity attributes (e.g. author, date, type) map to an entity's color, shading, and texture. As for the layouts, concrete mappings are highly task-dependent and are discussed in Section 6. Yet, we have found several design decisions which were generally applicable to all our visualizations, as follows.

- *Categorical* attributes, e.g. authors, file types, or search keywords are best shown using a fixed set of around 20 perceptually different colors. If more exist (e.g. in a project with 40 authors), colors are cycled. Using different color sets for different attributes performed best even when only a single attribute was shown at a time. Categorical sets with less than $4\ldots6$ values can also be effectively mapped to carefully chosen simple texture patterns if the zoom level is above 20 pixels per entity in both dimensions [15]. Texture and color allow showing two independent attributes simultaneously.
- *Ordinal* attributes, e.g. file size or age, bug criticality, or change amount, are best shown using continuous colormaps. We tried several colormaps: rainbow, saturation (gray-to-some-color), and three-color (e.g. blue–white–red). Interestingly, the rainbow colormap was the quickest to learn and accept by most software engineers and also by non-expert (e.g. student) users.
- *Shading* is not used to show attributes but structure. We use shaded parabolic [40] and plateau cushions [41] to show entities on different scales: files in project views (horizontal cushions in Figs. 5, 8, 13, and 15), file versions in file views (vertical stripe-like cushions in Figs. 3 and 10), and even whole subsystems in the decomposition view (Fig. 14).
- *Antialiasing* is essential for overview visualizations. These can easily contain thousands of entities (e.g. files in a project or lines in a file), so more than one entity per pixel must be shown on the vertical axis. For memory allocation logs [37], the horizontal (time) axis also can have thousands of entries. We address this by rendering several entries per pixel line or column with an opacity controlled by the amount of fractional pixel coverage of every entry. An example of antialiasing is given in Section 6.5.

We next present the several types of views used by our multiscale software evolution visualizations.

5.3. File view

In the *file view*, the entities are lines of code of the same file. For the vertical layout, we tried two approaches. The first one, called *file-based layout*, simply stacks code lines atop of each other as they come in the file (Fig. 3 top). This layout offers a "classical" view on file structure and size evolution similar to [8]. The second approach, called *entity-based layout* (Fig. 3 bottom), works as follows. First, we identify all evolution sets $E(e_{ij})$ using the transitive closure of the line `diff` operator. These are the sets of lines e_{ij} in all file versions V_i where all lines in a set are found identical by the `diff` operator. Next, we lay out these line sets atop of each other so that the order of lines in every file version V_i is preserved. For a version V_i, this layout inserts empty spaces where entities have been deleted in a previous version V_j $(j<i)$ or will be inserted in a future version V_k $(k>i)$. As its name says, the entity-based layout assigns the same vertical position to all entities found identical by the `diff` operator, so it emphasizes where in time *and* in file major code deletions and insertions have taken place.

Fig. 3 visualizes a file evolution through 65 versions. Color shows line status: green is constant, yellow modified, red modified by deletion, and light blue modified by insertion, respectively. In the line-based layout (bottom), gray shows inserted and deleted lines. The file-based layout (top) clearly shows the file size evolution. We note the stabilization phase occurring in the last third of the project. Here, the file size decreases slightly due to code cleanup, followed by a relatively stable evolution due to testing and debugging. Yellow fragments show edited code during the debugging phase. Different color schemes are possible, as described later by the use case in Section 6.1.

5.4. Code view

The code view offers the finest level of detail or scale in our toolset, i.e. a detailed text look at the actual source code corresponding to the mouse position in the file view. Vertical brushing over a version in the file view scrolls through the program code at a specific moment. Horizontal brushing in the entity-based layout (Fig. 3 bottom) goes through a given line's evolution in time. The code view is similar to a text editor with two enhancements. First, it indicates the author of each line by colored bars along the vertical borders (Fig. 4a). The second enhancement regards what to display when the user brushes over an empty space in the entity-based layout (light gray areas in Fig. 3 bottom). This space corresponds to code that was deleted in a previous version or will be inserted in a future version. Freezing the code view would create a sensation of

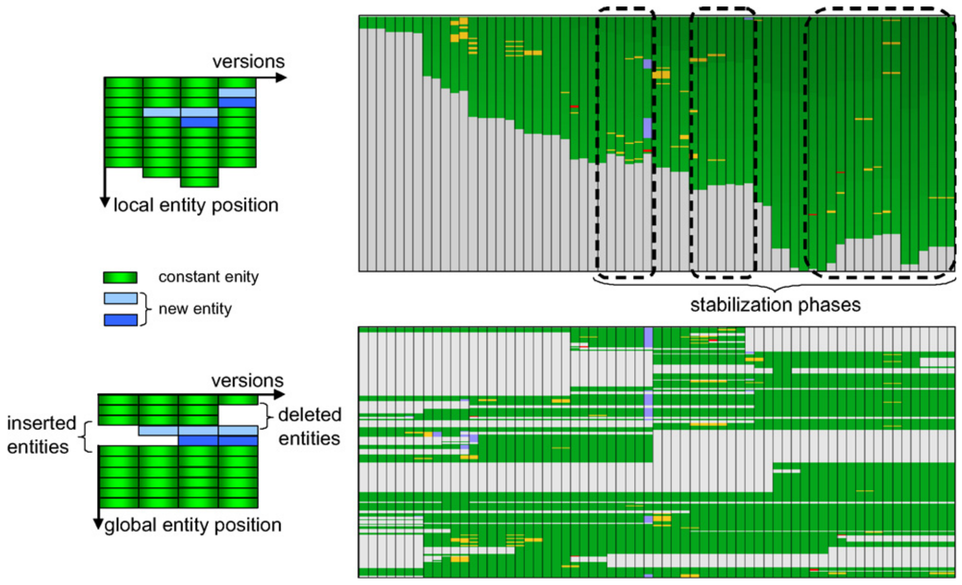
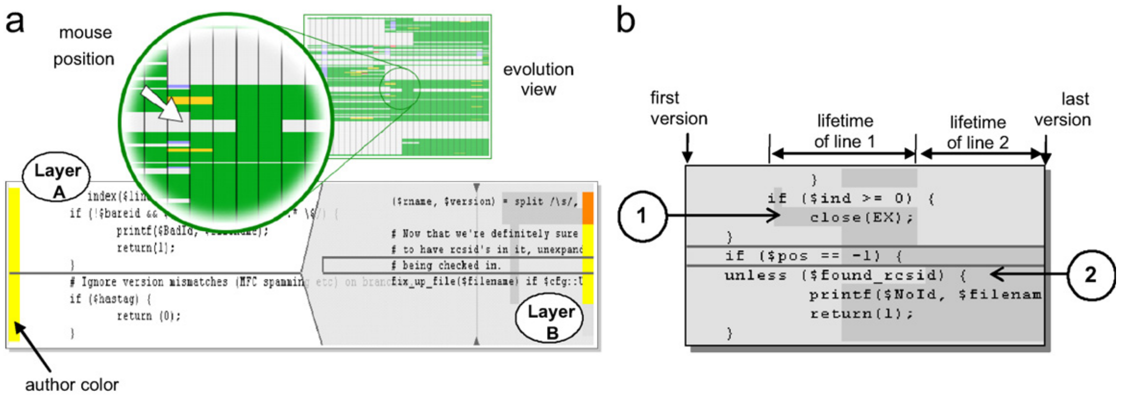


Fig. 3. File view with file-based (top) and entity-based layouts (bottom).



```
    index($lin
  if (!$bareid &&                          .* \$/) {
        printf($BadId,          );
        return(1);
  }
  # Ignore version mismatches (MFC spamming etc) on branc
  if ($hastag) {
        return (0);
  }
```

```
($rname, $version) = split /\s/,

# Now that we're definitely sure
# to have rcsid's in it, unexpanc
# being checked in.
fix_up_file($filename) if $cfg::U
```

```
            }
        if ($ind >= 0) {
            close(EX);
    }
    if ($pos == -1) {
    unless ($found_rcsid) {
            printf($NoId, $filenam
            return(1);
    }
```

Fig. 4. (a) Two-layered code view correlated with a version-uniform sampling entity layout, (b) code view, layer B. Line 1 is deleted before line 2 appears, i.e. they do not coexist.

scrolling disruption, as the mouse moves but the text does not change.

We solve this problem by the following enhancement. We use two text layers to display the code around the brushed entity position both from the version under the mouse *and* from versions in which this position refers to a non-empty space (Fig. 4a). While first layer (A) freezes when the user brushes over an empty region in the file view, the second layer (B) pops-up and scrolls through the code that has been deleted, or will be later inserted, at the mouse location. This creates a smooth feeling of scrolling continuity during browsing. This preserves the context of the selected version (layer A) and gives also a detailed, text-level peak, at the code evolution (layer B). The three motions (mouse, layer A scroll, layer B scroll) are shown by the captions 1, 2, and 3 in Fig. 4b.

We must now consider how to assess the code evolution shown in layer B. The problem is that, as the user scrolls through empty space in the file view, layer B consecutively displays code lines (deleted in past or inserted in future) that may not belong to a *single* (past or future) version. To correlate this code with the file view, we display the entities' lifetimes as dark background areas in layer B (Fig. 4b).

5.5. Project view

The project view shows a higher level perspective on the evolution of an entire system. The entities are file versions. The project view uses an entity-based layout—the evolution of each file is a distinct horizontal strip in this view, rendered with a cylindrical shaded cushion. Fig. 5 shows this for a small project. Sorting the files on the *y*-axis provides different types of insight. For example, the files in Fig. 5 are sorted on creation time and colored by author id. We quickly see a so-called "punctuated evolution"

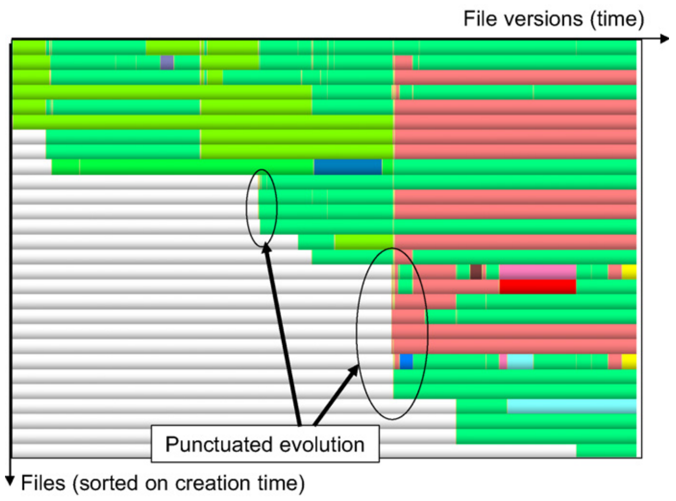


Fig. 5. Project view: Files are sorted on creation time and colored by author IDs.

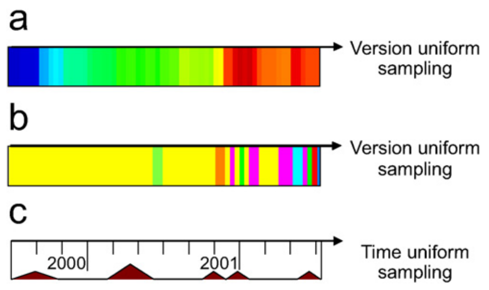


Fig. 6. Horizontal metric bars: (a) version size; (b) version author; (c) activity density.

moment, when several files have been introduced at the same time in the project. Virtually in all cases, such files contain related functionality. We can also sort files by *evolutionary coupling* with a given target file. Evolutionary coupling measures the similarity of two files' commit moments, as detailed in [14]. Similar files change together, so most probably contain highly related code or signal code drift and refactoring events. In terms of rendering, we can explicitly emphasize the individual entities (i.e. file versions) by drawing them as separate shaded cushions (Fig. 8). The project view is illustrated in use cases in Sections 6.3 and 6.4.

5.6. Decomposition view

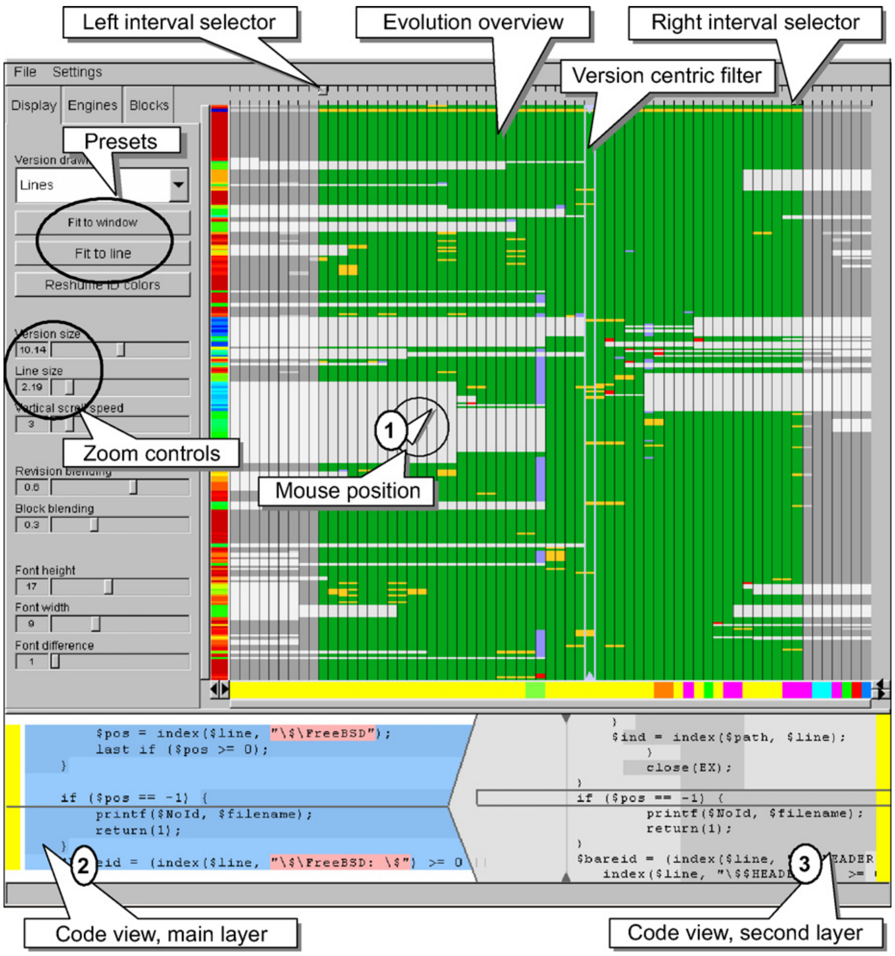
The decomposition view offers an even more simplified, compact, view than the project view. The role of this view is to let users visualize the strongly cohesive, loosely coupled components of a software system evolution. Since this view is easier to explain with a concrete use scenario, we postpone its description until Section 6.4.

5.7. User interaction

User interaction is essential to our toolset. We sketch here the set of interaction techniques we provided using the perspective proposed by Shneiderman [42]. Real tool snapshots illustrating these techniques are shown in Figs. 7 and 8.

The file, project and decomposition views offer *overviews* of software evolution, all as 2D images. To get detailed insight, *zoom* and *pan* facilities are provided. Zooming brings *details-on-demand*—text annotations are shown only below a specific zoom level, whereas above another level antialiasing is enabled (see e.g. Fig. 13 later in this paper). We offer preset zoom levels: global overview (fit all code to window size) and one entity-per-pixel-line level. To support the file evolution analysis from the perspective of a given version, we offer a *filtering* mechanism that removes all lines that are inserted after, or lines that are deleted before that version. Filtering enables assessing a version, selected by clicking on it, by showing its lines that are not useful and will be eventually deleted and the lines that have been inserted into it since the project start. This is demonstrated by the use case in Section 6.2. Hence, filtering provides a version-centric visualization of code evolution. Our tool gives the possibility to *extract* and select only a desired time interval by using two sliders (Fig. 7 top) similar to the page margin selectors in word processors. This mechanism proved to be useful in projects with a long lifetime (e.g. over 50 versions) which have distinct evolution phases that should be analyzed separately. The distinct phases were identified using a project view (Fig. 8), after which detailed file views were opened and the period of interest was selected using the version sliders described above. Note the resemblance in design the file and project view (Figs. 7 and 8). This is not by chance but a conscious design decision which tries to minimize the cognitive change the user has to undergo when changing views in our visualization toolkit. Interestingly enough, we noticed that this change occurs even when the differences of the two views are functionally minimal, i.e. they "work the same way" but happen to use different GUI toolkits in their implementation. Consequently, to minimize this difference which was experienced by our users as a serious hinder in using the toolkit, we had to re-implement the file view using the same type of toolkit as the project view—a laborious but highly necessary endeavor.

All views enable *correlating* information about the software evolution with overall statistic information, by means of *metric bars* (Fig. 6). These show statistical information about all entities sharing the same x or y coordinate, e.g. the lifetime of a code line, amount of project-wide changes at a moment, author of a commit, etc. The bi-level code view (Fig. 7, captions 2 and 3) gives *details-on-demand* on the fragments of interest by simply brushing the file evolution area. Moreover, the project view shows detailed information about the brushed file version in the form of user commit comments (Fig. 8, caption 2).



```
        $pos = index($line, "\$\FreeBSD");
        last if ($pos >= 0);
    }

    if ($pos == -1) {
        printf($NoId, $filename);
        return(1);
    }
    eid = (index($line, "\$\FreeBSD: \$") >= 0
```

```
    }
    $ind = index($path, $line);
        }
        close(EX);
}
if ($pos == -1) {
        printf($NoId, $filename);
        return(1);
}
$bareid = (index($line,  EADER
    index($line, "\$$HEAD  >=
```

Fig. 7. File view.

6. Use cases and validation

The main audience of our software evolution visualizations is the software maintenance community. Maintainers work outside the primary development context of a project, usually long after the end of the initial development. In order to validate our proposed techniques, we organized several informal user studies and experiments based on the methodology proposed in [43]. We assessed the visualization insight by analyzing the experiences of

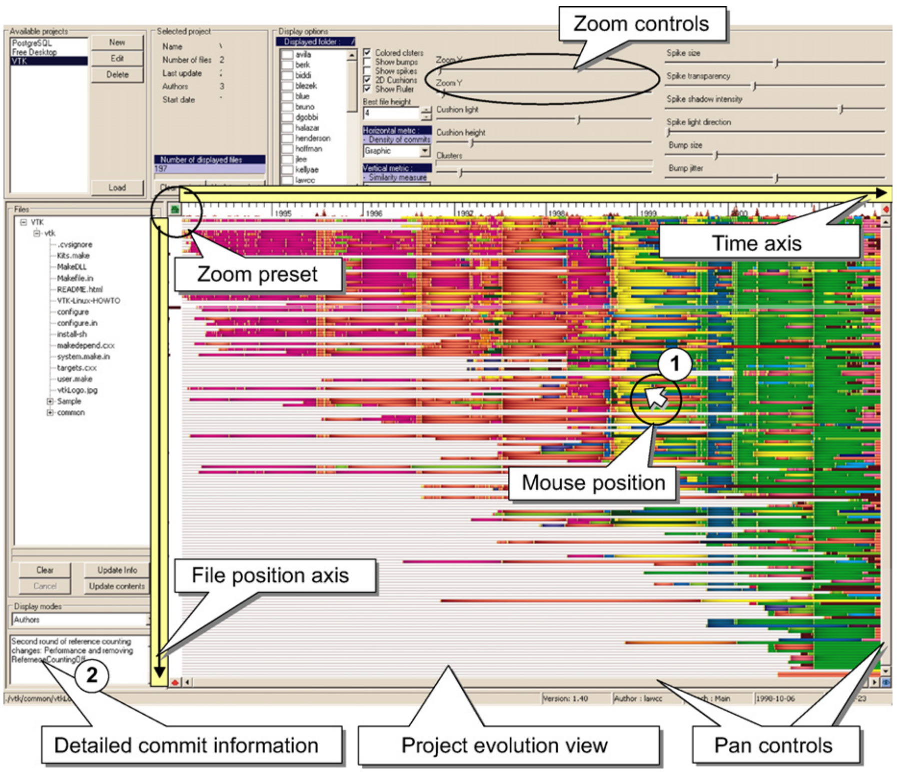
- *developers* and *architects* familiar with (i.e. involved in the production of) a given system;
- *developers* who investigate completely new code, but are familiar with similar systems;
- *developers* who investigate a completely new code and are unfamiliar with similar systems.

In all cases, only our visualization toolset (plus a typical text editor) were used. No documentation and/or expert coaching on the examined system were provided. We present below the outcome of several such experiments, selected from a larger set of studies that we have performed in the past two years. Each experiment illustrates a different type of scenario and uses different features of our toolset.

6.1. Use case: assessment of file structure and development context

An experienced C developer was asked to analyze a file containing the socket implementation of the X Transport Service Layer in the Linux FreeBSD distribution. The file had approximately 2900 lines and spanned across 60 versions. The user was not familiar with the software, nor was he told what the software was. We provided a file view (Section 5.3) and a code view (Section 5.4) able to highlight C grammar and preprocessor constructs, e.g. `#define`, `#ifndef`, etc. The user received around 30 min of training with our toolset. A domain expert acted as a silent observer and recorded both user actions and findings (marked in italics in the text below). At the end, the domain expert subjectively graded the acquired insight on five categories: *Complexity*, *Depth*, *Quality*, *Creativity*, and *Relevance*. Each category was graded from 1 (i.e. minimum/weak) to 5 (i.e. maximum/strong).

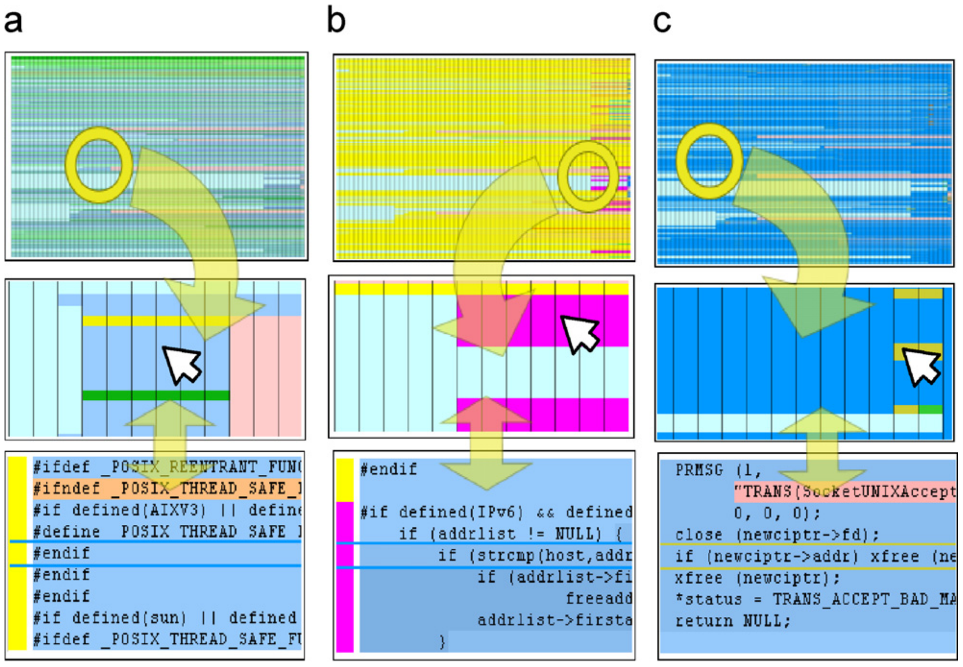
The user started his analysis in the line-based layout (e.g. Fig. 3 bottom) and searched first for comments: *This is the copyright header, pretty standard. It says this is the*



```
Second round of reference counting
changes: Performance and removing
ReferenceCounting0
```

```
Version: 1.40    Author : lawcc    1998-10-06
```

Fig. 8. File view.



```
#ifdef _POSIX_REENTRANT_FUN(
#ifndef _POSIX_THREAD_SAFE_I
#if defined(AIXV3) || define
#define  POSIX THREAD SAFE I
#endif
#endif
#endif
#if defined(sun) || defined
#ifdef _POSIX_THREAD_SAFE_FU
```

```
#endif

#if defined(IPv6) && defined
    if (addrlist != NULL) {
        if (strcmp(host,addr
            if (addrlist->fi
                freeadd
            addrlist->firsta
        }
```

```
PRMSG (1,
      "TRANS(SocketUNIXAccept
      0, 0, 0);
close (newciptr->fd);
if (newciptr->addr) xfree (ne
xfree (newciptr);
*status = TRANS_ACCEPT_BAD_MA
return NULL;
```

Fig. 9. Case study—analysis of a C code file.

implementation of the X Transport protocol... It seems they explain in this comments the implementation procedure... Next, he switched his attention to the compiler directives: *A lot of compiler directives. Complex code, supposed to be portable on many platforms. Oh, even Windows.* Next, he started to evaluate the inserted and deleted code blocks: *This file was clearly not written from scratch, most of its contents has been in there since the first version. Must be legacy code... I see major additions done in the beginning of the project that have been removed soon after that... They tried to alter some function calls for Posix thread safe functions...* (see Fig. 9a top bottom) *I see big additions also* *towards the end of the project... A high nesting level, could be something complex... It looks like IPv6 support code. I wonder who did that?*

The user switched then to the *author* color encoding: *It seems the purple user, Tsi, did that* (Fig. 9b top bottom). *But a large part of his code was replaced in the final version by... Daniel, who committed a lot in the final version... And everything seems to be IPv6 support. The green user, Eich... well, he mainly prints error messages.* Finally, our user switched on line status color encoding and zoomed in: *Indeed, most work was done at the end Still, I see some major changes in the beginning throughout the file... Ah, they changed the memory manager. They stepped to one specific to the X environment. All memory management calls are now preceded by x* (Fig. 9c top bottom)*... And they threw away the TRANS macro.*

The user spent the rest of the study assessing the changes and the authors that committed them. After 15 min, the user did not have a very clear image of the file's evolution, but he concluded easily that the file represented a piece of legacy code adapted by mainly two users to support the IPv6 network protocol. He also pointed out a major modification: the change of the memory manager. The subjective grading estimating the visualization insight is given in Table 2.

Although informal, this study shows that the line-based file and code views support a quick assessment of the important activities and line-level artifacts produced during development, even for users that had not taken part in any way in developing the examined code. The file view scored very well in the categories *Complexity*, *Quality* and *Relevance*. The *Depth* and *Creativity* categories scored only medium. An explanation for this could be the relatively short examination time (30 min) that did not allow the user to consolidate the discovered knowledge and make more advanced correlations. The study subject valued most the compact overview (the file view) coupled with easy access to source code (the code view). These enabled the user to easily spot issues at a high level and then get detailed line-level information. Concluding, the file and code views can be useful to new developers in a team who need to understand a given development context, thereby reducing the time (and costs) required for knowledge transfer.

6.2. Use case: assessment of framework migration effort in component-based systems

Component-based SE is regarded as a promising approach towards reducing the software development time and costs. However, as the number of component models increases, a new challenge arises: how to discriminate among models that satisfy the same set of requirements so that the best suited one is selected as development base for a given system? Using the evaluation methodology proposed in [44], one can reach the conclusion that e.g. the Koala [45], and PECOS [46] component models offer similar benefits regarding testability, resource utilization, and availability. In such a case, the selection of the best suited model can be further refined e.g. with information on which model fits better with the software development strategy that will be used during the project's lifecycle.

When component frameworks are not yet mature, new framework versions are often incompatible with previous ones. In such cases, existing components need to be re-architected in order to be supported by the new framework. The effort in this step may be so high that migrating to a totally different, more mature, component framework or staying with the old framework may be better alternatives. A good estimation of the transition cost of framework change is therefore of great importance.

We show here how the file view can be used to make such estimations, based on history recordings for components that have been already re-architected to comply with new framework versions.

Fig. 10 shows two file views for the evolution of a ROBOCOP [47] component along 17 versions. The transition from version 16 to 17 corresponds to the component migration from ROBOCOP 1.0 to ROBOCOP 2.0. In Fig. 10, a file-based layout is used together with a version filter (see [11,12]) to depict the amount of code from one version that can be found in other versions. Only code that can be tracked to the selected version is displayed for each version. Hence, the selected version appears to have always the largest line count, as lines that have been previously deleted or inserted afterwards are not displayed. Color shows change: light gray are unchanged lines and black (dark) shows changed lines. From this image, one can infer that a lot of code had to be changed when passing from component version 16 to version 17, as many lines are black. Also, only about 70% of the component code from version 16 is found in version 17, as the vertical length of version 17 is less than three quarters the length of version 16 in Fig. 10 left. Similarly, Fig. 10 right shows that about 40% new code had to be written for version 17 over what was preserved from version 16. Overall, about 50% of the component code in version 17 differs from the one in version 16. This signals a quite high effort to adapt components to cope with changes in the Robocop framework. These findings were validated by the Robocop development team after this experiment was completed.

Concluding, the effort required to migrate a component based system from ROBOCOP 1.0 to ROBOCOP 2.0 is quite large. If a migration step has to be taken anyway, one should review alternative component frameworks and consider migration to one of them provided they offer higher benefits for a comparable effort. This type of

Table 2
Insight grading for analysis of a C code file

Category	Grade
Complexity	5
Depth	3
Quality	5
Creativity	3
Relevance	5

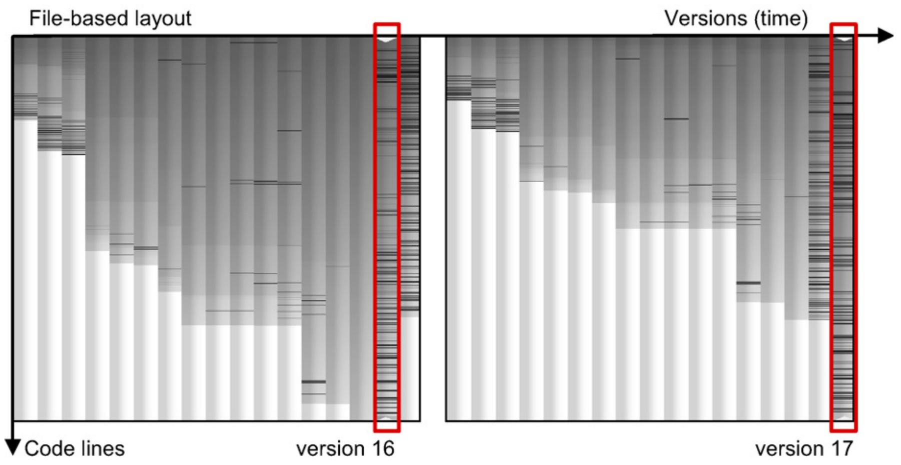


Fig. 10. Component migration from Robocop 1.0 to Robocop 2.0. Left: changes from the perspective of version 16. Right: changes from the perspective of version 17. Code that cannot be tracked to the selected version is not displayed.

assessment can be used by project managers to quickly assess the transition efforts for a component framework, provided that previous transition examples exist, whether from the same or another project.

6.3. *Use case: assessment of major changes in a project*

During the lifetime of a project, major changes may occur. These involve changing a large amount of code and files due to specific circumstances. The occurrence patterns of such changes can disclose the circumstances that led to their appearance and their relevance on the system architecture and/or quality.

We used the project view (Section 5.5) to asses the major changes in the VTK project [48]. VTK is a complex C++ graphics library of hundreds of classes in over 2743 files, including the contribution of more then 40 authors over a 12 year period. In the project view, every file is shown as a horizontal strip, and every version as a vertical one. On the y-axis, files are sorted alphabetically based on their full path and thus are implicitly grouped on folders. A rainbow colormap encodes for each file version the normalized amount of change. Blue shows no change and red shows the maximal change throughout the project (Fig. 11). Antialiasing is used to improve the visual appearance.

Looking for red (maximal change) patterns in the result, we find three interesting evolution patterns. Pattern A, an elongated horizontal segment, denotes a major size change (hundreds of lines) affecting a small number of files in the same directory for every version over a very long period. Zooming in, we discovered that this anomaly is caused by binary files which have been automatically checked in the CVS repository. CVS can only handle text line changes, so binary files are seen to be completely new every time they change. In general, configuration managers consider as good practice not including binary code in a repository. Pattern B denotes a major size change affecting a large number of files *in the same directory* during about 15% of

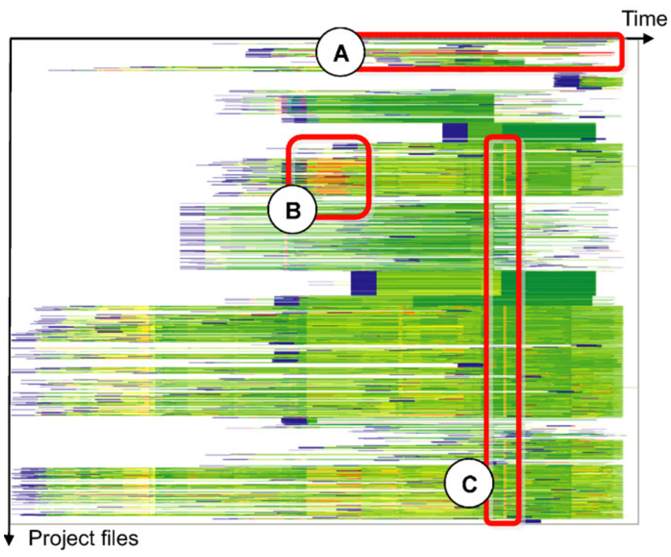


Fig. 11. Major change patterns in the VTK toolkit.

the project lifetime. This type of pattern indicates typically an architecture change localized to a given subsystem. For the VTK project, this pattern matches the period when a new API was released for the imaging subsystem. Pattern B indicates thus critical development events for a system's architecture or quality. Finally, pattern C, shaped as a thin ling vertical strip, shows a major size change affecting 75% of all project files, but only at a specific time moment. This type of pattern usually signals cosmetic activities (e.g. indentation) that do not change the system architecture or functionality in any way. These patterns often correspond to official releases of a project. For VTK, pattern C marks the change of the copyright notice that is included in most source code files. Indeed, its log comment signals the official `release-3-2-branch-point`. The findings have been checked and validated by an expert developer with over eight years VTK experience.

We have found the major change patterns shown in Fig. 11 in all large software projects. Finding them is important for several types of users. By identifying type A patterns of type A, configuration managers can spot archive bloaters, e.g. automatically generated and accidentally committed binary files, and remove them from the make process. Type B patterns are highly relevant for architects and project managers. They denote critical periods in the development of the project. This insight can be used by architects during reverse engineering to understand the design decisions of a project when documentation is not available. They are also important for managers who must ensure that full regression tests are successfully ran after each such moment. Also, project managers can use these moments as starting point for estimating change propagation costs and calculating the effort needed to complete a specific development or maintenance task. Finally, type C patterns can be used to identify the number of policy or copyright-related-changing releases of a project.

6.4. *Use case: assessment of propagation of debugging-induced changes*

Large software systems change often e.g. because of adding new functionality or due to debugging. Change propagation is very important when assessing the effort needed to modify a specific part of a system. It gives an indication of the total change integration costs, including changes that might be needed in other parts of the system, in order to preserve consistency. To reduce this collateral change cost, software architects try to organize systems as loosely coupled entities, minimizing the risk of changes to propagate across entities. Hence, the patterns of change propagation in a system can help assessing its architectural quality.

We used our project (Section 5.5) and decomposition (Section 5.6) views to assess the propagation of changes induced by debugging activities in the Firefox project, part of the Open Source project Mozilla. Firefox has 659 files contributed by 108 authors over more than 4 years. It contains fixes for 4497 bugs from the total bug count reported.

We used our toolset to load the Firefox evolution data from the Mozilla CVS server. Separately, we used the Bugzilla web interface of the Mozilla project to load the list of fixed bugs. We started from the assumption that changes induced by bug fixes propagate to files that have been reportedly modified at the same time with the files which were debugged. Hence, we started our inquiry by identifying files versions containing bug fixes. Fig. 12 shows a project view containing the 659 files of the Firefox browser sorted vertically in alphabetical order. The locations of debugging activities are marked by fixed size red icons. Due to the window size, it is possible that such icons overlap. To convey the actual icon density, we render semitransparent disks centered at the debugging event locations. The blended overlap of these disks yields areas of higher color intensity in regions of high debugging density. This technique is similar to the graph splatting promoted by Van Liere et al. [49] for visualizing complex graphs.

After identifying the files containing bug fixes, we pursued our inquiry by filtering these candidates to a smaller, more interesting set. We looked for a subsystem with a high debugging activity in the recent history, as this could be a change-prone subsystem also in the near future. Fig. 12 highlights such an area. As files are implicitly grouped on folders, the highlighted area shows a (group of) folder(s) with recent intense debugging activity. We identified the specific files by zooming in until file names became visible (Fig. 13) and discovered that all files in the high debugging activity area are in the `/component/places` folder.

We interactively marked the files in this folder with yellow (Fig. 13b). Next, we continued our analysis by

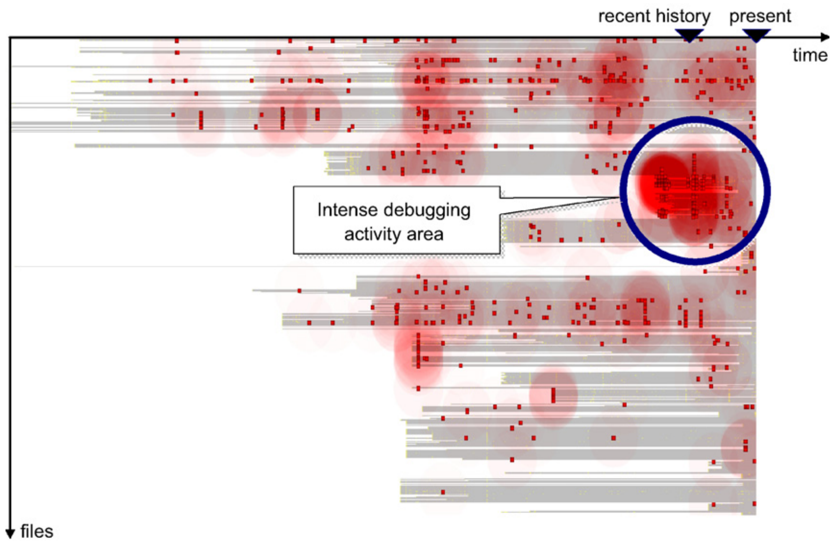


Fig. 12. Bug fix locations in the Firefox project.

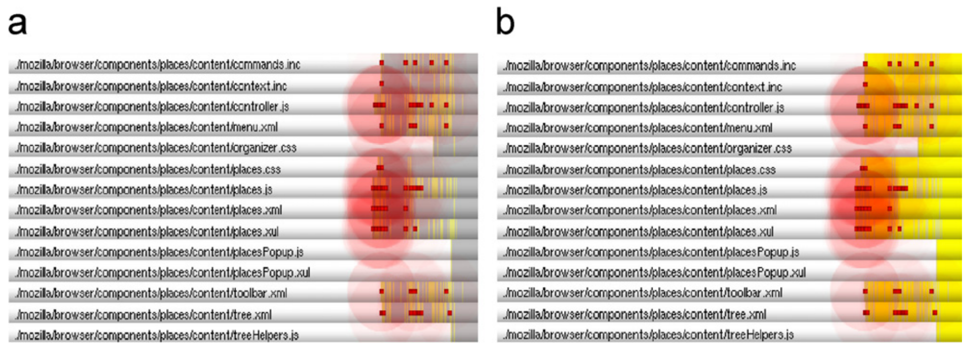

a
b
./mozilla/browser/components/places/content/commands.inc
./mozilla/browser/components/places/content/context.inc
./mozilla/browser/components/places/content/controller.js
./mozilla/browser/components/places/content/menu.xml
./mozilla/browser/components/places/content/organizer.css
./mozilla/browser/components/places/content/places.css
./mozilla/browser/components/places/content/places.js
./mozilla/browser/components/places/content/places.xml
./mozilla/browser/components/places/content/places.xul
./mozilla/browser/components/places/content/placesPopup.js
./mozilla/browser/components/places/content/placesPopup.xul
./mozilla/browser/components/places/content/toolbar.xml
./mozilla/browser/components/places/content/tree.xml
./mozilla/browser/components/places/content/treeHelpers.js

Fig. 13. Zoom-in in a high-debugging activity area in the Firefox project.

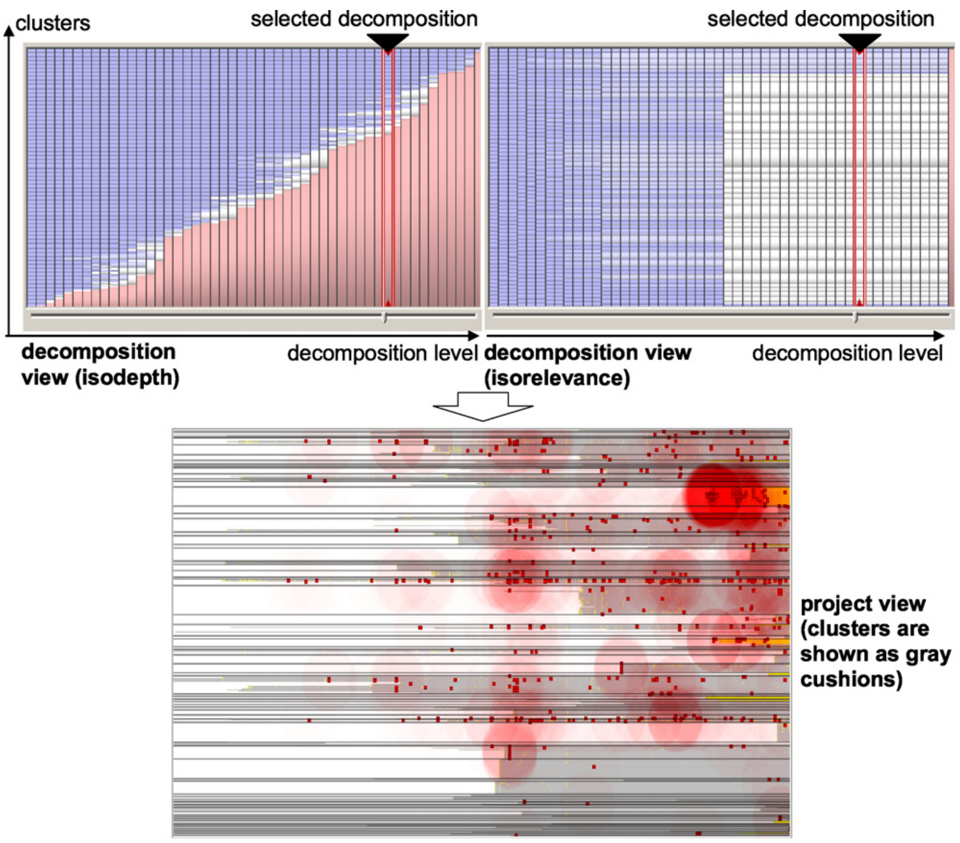

clusters
selected decomposition
selected decomposition
decomposition view (isodepth)
decomposition level
decomposition view (isorelevance)
decomposition level
project view (clusters are shown as gray cushions)

Fig. 14. Firefox system decomposition: isodepth partition (top); isorelevance partition (bottom).

identifying how changes in these files propagate to other files in the project. For this, we clustered all files in Firefox based on the so-called *evolutionary coupling*. As explained previously, two files evolve similarly if they have similar commit moments. This technique is described in detail in [14]. The clustering produces a tree of increasingly larger file clusters. Leaf clusters contain files which evolve very similarly and top clusters contain clusters of less similarly evolving files. The cluster tree is visualized by the *decomposition view* shown in Fig. 14 (top). The entities in this view are the clusters. The layout of this view is as follows. The x-axis maps the decomposition level. This is the only view of our toolset where the x-axis does not map the time. The y-axis maps the decomposition itself by drawing all clusters (groups of files) for the current decomposition level (x-axis) as stacked rectangle entities, scaled vertically to show the cluster size, i.e. number of files in a cluster. The clusters are drawn as shaded cushions and colored based on their cohesion, or coupling strength using a blue–white–red colormap (blue = strong cohesion, red-weak cohesion). Once a decomposition level is chosen (by clicking on a column in the decomposition view), its file clusters are drawn over the files in the project view as luminance plateau cushions. These cushions are visible in the project view in Fig. 14 as horizontal gray bands, the area between two dark gray bands being a cluster.

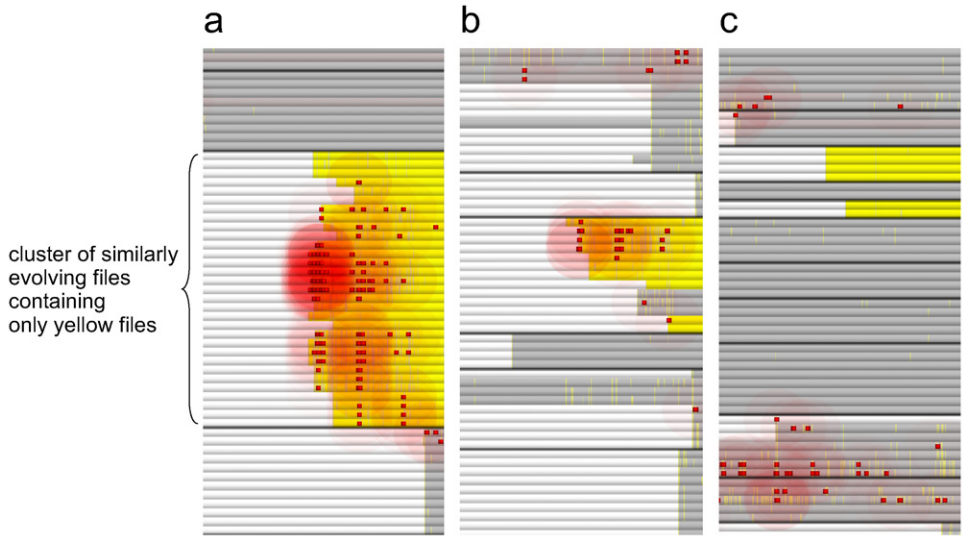


Fig. 15. Occurrences of files from /component/places in the clustered project view.

We used the decomposition view to choose an appropriate system decomposition level to look at as a compromise between the number of clusters, cluster size and cluster relevance. We considered two clustering methods: isodepth and isorelevance [15]. In the isodepth method, a decomposition level contains clusters with the same depth in the cluster tree (Fig. 14 upper left). However, this tends to produce a few large clusters and many tiny clusters on the same level. In the isorelevance method, a level contains clusters having relatively equal cohesion (Fig. 14 upper right). In line with previous findings [15], the iso-relevance method proved the best (easiest to understand) decomposition: At every level, this decomposition provides file clusters that are equally likely to be modified together (Fig. 14 bottom).

The last step of our investigation was to find clusters containing files from the high debugging activity folder `/component/places` (i.e. yellow files in Fig. 13b) and to discover what *other* files these clusters contain, i.e. what other files have a similar evolution. For this, we zoomed in the project view and we looked at each cluster individually. Clusters containing notable occurrences of `/component/places` (i.e. yellow) files are shown in Fig. 15. The largest cluster (Fig. 15a) contains only files in the /component/places folder (yellow files). Consequently, debugging activities in this group of files seem to be contained in the folder. The second largest cluster (Fig. 15b) contains mainly yellow files and only three files belonging to other system parts (gray files). This means it is possible that *changes induced by debug activities in the yellow files could propagate to these three files*. Fig. 15c shows an example of the remaining notable occurrences of yellow files in the project view. The clusters contain just a few yellow files, without marks of debugging activity, and no files from other folders (gray files).

We concluded that the debugging-induced changes in the `/component/places` folder are *mainly contained in the folder* and do not propagate to other system parts (other folders). Although the folder is still subject to intense debugging activity in recent history (Fig. 12 right), it is likely the effort will be confined to changes inside the folder. This insight can help project managers to make a more precise estimation of the resource planning and is an indication of a weakly coupled (i.e. good quality) architecture of the Firefox system. In general, this type of assessment is mainly useful for project and product managers. Project managers can use it to predict hidden costs that are not directly associated with specific system functionality but result from integration and synchronization activities. Product managers can use this to assess the quality of third-party systems before using them in a specific product.

6.5. *Use case: assessment of the behavior of a dynamic memory allocator*

We conclude our use cases series with a different kind of example. We visualize the dynamic behavior of a memory allocator. Entities, saved in a log file by an allocator profiler [37], are now (de)allocated heap blocks instead of code as in the previous examples. Entity attributes are the ID of the process which (de)allocated it, its memory start and end address, allocation and deallocation time, and *bin number*. The allocator slices the heap into 10 memory portions or bins. Each bin b_i holds only blocks within a given size range $[r^i_{min}, r^i_{max}]$ to limit fragmentation. Our visualization targets software engineers interested in optimizing a given memory allocator, e.g. reduce fragmentation or decrease allocation time.

For each bin, we visualize the memory data using a view similar to the project view (Fig. 16). The x axis maps time,

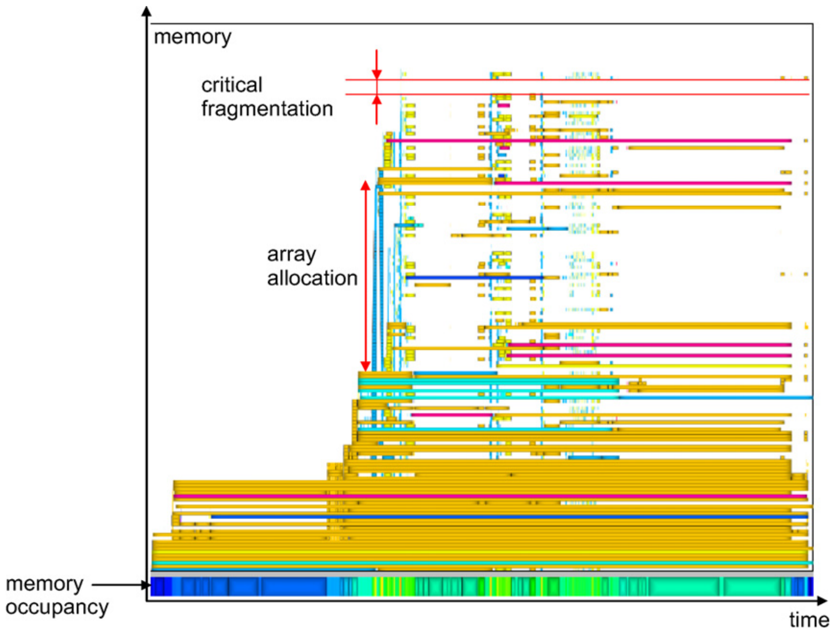


Fig. 16. Visualization of the evolution of dynamic memory allocations.

the y axis maps the memory. Blocks are drawn using shaded plateau cushions, in this case colored by process ID. Memory fragmentation maps to the coverage of display space by memory blocks. We quickly see that there is much higher fragmentation in the upper than in the lower memory range. This points to a suboptimal allocator behavior. Also, we find horizontal "gaps" in the visualization (see Fig. 16 top). These denote critical fragmentation events which should be intercepted by the allocator. Thin vertical contiguous strips denote typical array allocations—many memory entities of the same size and consecutive locations allocated at the same time. As we can see in Fig. 16, such array allocations can dramatically increase fragmentation as they block large portions of the memory. We also see an expected phenomenon, namely that the lifetime of a block is totally uncorrelated with the moment when the block was allocated. Early allocated blocks can last very long, such as the ones in the lower part of Fig. 16, but so can lately allocated blocks too. A horizontal metric bar displays the total memory occupancy using a blue-to-red colormap. We notice that critical fill-in levels (warmer colors in the bar) correspond to points when arrays get allocated. Also, we notice that the profiled scenario ends up with about the same (low) level of memory occupancy as it started—the occupancy metric bar shows the same blue color at beginning at end. However, the memory is clearly more fragmented at the process end than at the beginning—blocks on the vertical axis are much less compact at the end moment than at the start moment of the monitored time interval.

A final point to notice is antialiasing. Providing this feature was absolutely essential for this application. In Fig. 16, there are 7770 (de) allocations drawn for a period of a few seconds, so the needed time (x)-axis resolution is high above the screen pixel resolution. Antialiasing, as sketched in Section 5.2, is essential here for correctly rendering the high-frequency (de)allocation events which take place in a short time interval.

Overall, the visualization presented here uses exactly the same techniques (and visualization software) as the software code evolution cases described previously, but targets another problem and data mode. This proves that our visualization evolution framework is generic enough to handle and provide insight in a large set of application areas having different data models and target questions.

7. Discussion

We have presented an integrated set of techniques and tools for visually assessing the evolution of source code in large software projects. Reflecting back on the requirements stated at the beginning of Section 2, we note the following. Our toolset provides the standard basic *management* tools of SCM systems via its integrated CVS client. We let users query and visualize software at several *scales*: source code in a single file (code view), source code in all versions of a file (file view), files in a whole project (project view), and a hierarchy of similarly-evolving subsystems over a whole project evolution (decomposition view). Our tool was easily *scalable* to handle huge projects of thousands of files, hundreds of releases, and tens of developers (e.g. VTK, ArgoUML, PostgresSQL, X Windows, and Mozilla). So far, we did not provide classical *data-mining* type of analysis tools (except the evolution-based clustering), but rather focused on the visual analysis itself. We offered a rich set of fully customizable *views* for different tasks. All views share a few basic design principles: 2D dense pixel orthogonal layouts for organizing software entities, colors and textures for attributes, and shaded cushions for structure. We usually do not parse the files' contents so our approach can handle any types of files in a repository. Finally, our toolset is *integrated* with a full-fledged CVS and a Subversion client, so it enables software engineers directly bring the power of visualization into their typical engineering activities, without having to incur the burden of cross-system switching.

Designing highly polished, but simple, user interfaces and responsive visualizations which react quickly even when large data amounts are loaded was absolutely essential for the tool to get accepted by various users, including engineers from small and large IT companies. Finally, we mention that we designed several less usual views, not presented here, e.g. the version-centric view with interpolated layout [12] or the isometric decomposition view [15]. However, as these views turned out to be significantly less understood by our users, we considered removing them from the actual tool distribution. This is in line with our strive to provide a toolkit based on a *minimal* set of features which is easy to learn and use, thus highly probable to adopt on a larger scale, outside the research environment itself and into the industrial engineering practice.

8. Conclusion

We have presented an integrated set of tools and techniques for the visual assessment of the evolution of large-scale software systems. The main characteristic of our system is, probably, its simplicity. Even though we can address a number of complex use cases, which few other (visual) analysis tools for software systems can handle, we do this by combining a few techniques: 2D layouts of software entities at several scales extracted from SCM repositories whose axes can be sorted to reflect various decompositions, dense pixel renderings encoding data attributes via customizable colormaps and texture patterns, shaded cushions to show up to three levels of system structuring, and ubiquitous user interaction and visual feedback such as brushing, cursors, and correlated views. To produce all visualizations shown here (or similar ones) one needs only to start the toolset, type in the location of a repository, and wait for the screen to be populated with visual information about the downloaded data. Most subsequent manipulations, such as sorting entities, changing color attributes, getting details on demand and so on, are reachable via just a few mouse clicks.

The main contribution of this paper is the presentation of a cohesive framework that is able to target visualizations of the evolution of a wide range of software artifacts (code, project structure, behavior) via a simple set of elements and design rules: 2D orthogonal layouts, dense pixel displays, color-mapped attributes, and shaded cushions. We generalize here our previous work and findings on software evolution visualization [11,13–15] to novel application areas and data types how our 'minimal' framework can effectively target a wide set of applications and questions. We illustrate our findings with several case studies of a wider experimental set performed over a period of over two years with our toolset. Given these results, we believe that our visualization framework can also target more, different types of evolutionary datasets in software engineering and even beyond the borders of this application domain.

All work presented here was implemented with our toolset which is available for download at: http://www.win.tue.nl/~lvoinea/VCN.html.

We are currently working to extend and refine our set of methods and techniques for visual code evolution investigation in two main directions. First, we work to refine the data model to incorporate several higher-level abstractions (e.g. classes, methods, namespaces). Second, we are actively researching novel ways to display the existing information in more compact, more suggestive ways. We plan to actively conduct more user tests to assess the concrete value of such visualizations, the ultimate proof of our proposed techniques.

References

[1] Burrows C, Wesley I. Ovum evaluates: configuration management. Burlington, MA, USA: Ovum Inc.; 1999.

[2] Stroustrup B. The C++ Programming Language. 3rd ed. Reading, MA: Addison-Wesley Professional; 2004.

[3] Erlikh L. Leveraging legacy system dollars for e-business. In: (IEEE) IT Pro; May–June, 2000. p. 17–23.

[4] Seacord RC, Plakosh D, Lewis GA. Modernizing legacy systems: software technologies, engineering process, and business practices. SEI Series in Software Engineering. Reading, MA: Addison-Wesley; 2003.

[5] Eiglsperger M, Kaufmann M, Siebenhaller MA. Topology-shape-metrics approach for the automatic layout of UML class diagrams. In: Proceedings of the ACM SoftViz '03. NY, USA: ACM Press; 2003. p. 189–98.

[6] Gutwenger C, Junger M, Klein K, Kupke J, Leipert S, Mutzel P. A new approach for visualizing UML class diagrams. In: Proceedings of ACM SoftViz '03. NY, USA: ACM Press; 2003. p. 179–88.

[7] Beck K, Andres C. Extreme programming explained: embrace change. 2nd ed. Reading, MA: Addison-Wesley; 2000.

[8] Eick SG, Steffen JL, Sumner EE. Seesoft—A tool for visualizing line oriented software statistics. In: IEEE Transactions on Software Engineering, Vol. 18, No. 11, Washington, DC, USA: IEEE Press; 1992. p. 957–68.

[9] Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. In: Proceedings of ICSE '02. NY, USA: ACM Press; 2002. p. 467–77.

[10] Telea A, Maccari A, Riva C. An Open toolkit for prototyping reverse engineering visualization. In: Proceedings of IEEE VisSym '02, The Eurographics Association, Aire-la-Ville, Switzerland, 2002. p. 241–51.

[11] Voinea L, Telea A, van Wijk JJ. CVSscan: Visualization of code evolution. In: Proceedings of the ACM Symposium on software Visualization (SoftVis'05). NY, USA: ACM Press; 2005. p. 47–56.

[12] Voinea L, Telea A, Chaudron M. Version centric visualization of code evolution. In: Proceedings of the IEEE Eurographics Symposium on Visualization (EuroVis'05). Washington, DC: IEEE Computer Society Press; 2005. p. 223–30.

[13] Voinea L, Telea A. CVSgrab: Mining the history of large software projects. In: Proceedings of the IEEE Eurographics Symposium on

Visualization (EuroVis'06). Washington, DC: IEEE Computer Society Press; 2006. p. 187–94.

[14] Voinea L, Telea A. An open framework for CVS repository querying, analysis and visualization. In: Proceedings of Intl Workshop on Mining Software Repositories (MSR'06). New York: ACM Press; 2006. p. 33–9.

[15] Voinea L, Telea A. Multiscale and multivariate visualizations of software Evolution. In: Proceedings of ACM Symposium on Software Visualization (SoftVis'06). New York: ACM Press; 2006. p. 115–24.

[16] CVS online: ⟨http://www.nongnu.org/cvs/⟩.

[17] Subversion online: ⟨http://subversion.tigris.org/⟩.

[18] Ball T, Kim J-M, Porter AA, Siy HP. If your version control system could talk .. ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering, May 1997, available online at: ⟨http://research.microsoft.com/~tball/papers/icse97-decay.pdf⟩.

[19] Fischer M, Pinzger M, Gall H. Populating a release history database from version control and bug tracking systems. In: Proceedings of ICSM'03. Silver Spring, MD: IEEE Press; 2003. p. 23–32.

[20] German D, Mockus A. Automating the measurement of open source projects. Presented at ICSE '03 Workshop on Open Source Software Engineering (OOSE'03), Portland, Oregon, USA, 2003. available online at: ⟨http://www.research.avayalabs.com/user/audris/papers/oose03.pdf⟩.

[21] Zimmermann T, Diehl S, Zeller A. How history justifies system architecture (or not). In: Proceedings of IWPSE'03. Washington DC, USA: IEEE Computer Society press; 2003. p. 73–83.

[22] Bonsai online: ⟨http://www.mozilla.org/projects/bonsai/⟩.

[23] NetBeans.javacvs online: ⟨http://javacvs.netbeans.org/⟩.

[24] Zimmermann T, Weigerber P, Diehl S, Zeller A. Mining version histories to guide software changes. In: Proceedings of ICSE'04. Silver Spring, MD: IEEE Press; 2004. p. 429–45.

[25] Gall H, Jazayeri M, Krajewski J. CVS release history data for detecting logical couplings. In: Proceedings of IWPSE 2003. Washington DC, USA: IEEE Computer Society Press; 2003. p. 13–23.

[26] Lopez-Fernandez L, Robles G, Gonzalez-Barahona JM. Applying Social Network Analysis to the Information in CVS Repositories, International Workshop on Mining Software Repositories (MSR'04), Edinburgh, Scotland, UK, 2004, online at: ⟨http://opensource.mit.edu/papers/llopez-sna-short.pdf⟩.

[27] Ducasse S, Lanza M, Tichelaar S. Moose: an extensible language-independent environment for reengineering object-oriented systems, Proceedings of the second International Symposium on Constructing Software Engineering Tools (CoSET '00), June 2000, online.

[28] Froehlich J, Dourish P. Unifying artifacts and activities in a visual tool for distributed software development teams. In: Proceedings of ICSE '04. Washington DC, USA: IEEE Computer Society Press; 2004. p. 387–96.

[29] Collberg C, Kobourov S, Nagra J, Pitts J, Wampler K. A system for graph-based visualization of the evolution of Software. In: Proceedings of ACM SoftVis '03. NY, USA: ACM Press; 2003. p. 77–86.

[30] Lanza M. The evolution matix: Recovering software evolution using software visualization techniques. In: Proceedings of the International workshop on principles of software evolution, 2001. NY, USA: ACM Press; 2001. p. 37–42.

[31] Wu J, Spitzer CW, Hassan AE, Holt RC. Evolution spectrographs: visualizing punctuated change in software evolution. In: Proceedings of the seventh International Workshop on Principles of Software Evolution (IWPSE'04). Silver Spring, MD: IEEE Press; 2004. p. 57–66.

[32] Wu X. Visualization of version control information. Master's thesis, University of Victoria, Canada, 2003.

[33] German D, Hindle A, Jordan N. Visualizing the evolution of software using SoftChange, In: Proceedings of the 16th Internation Conference on Software Engineering and Knowledge Engineering (SEKE 2004). p. 336–41.

[34] Zimmermann T, Weigerber P. Preprocessing CVS data for fine-grained analysis, International workshop on mining software repositories (MSR), Edinburgh, May 2004. ⟨http://www.st.cs.unisb.de/papers/msr2004/msr2004.pdf⟩.

[35] Bieman JM, Andrews AA, Yang HJ. Understanding change-proneness in OO software through visualization. In: Proceedings of the International Workshop on Program Comprehension (IWPC'03). Silver Spring, MD: IEEE Press; 2003. p. 44–53.

[36] Ying ATT, Murphy GC, Ng R, Chu-Carroll MC. Predicting source code changes by mining revision history. In: IEEE Transactions on Software Engineering, vol. 30(9), Washington, DC, USA: IEEE Computer Society Press; 2004. p. 574–86.

[37] Del Rosso C. Dynamic memory management for software product family architectures in embedded real-time systems. In: Proceedings WICSA'05. Silver Spring, MD: IEEE Press; 2005.

[38] Card SK, Mackinlay JD, Shneiderman B. Readings in information visualization: using vision to think. San Francisco: Morgan Kaufmann; 1999.

[39] Spence R. Information visualization. New York: ACM Press; 2001.

[40] van Wijk JJ, van de Wetering H. Cushion Treemaps: visualization of hierarchical information. In: Proceedings of IEEE InfoVis. Washington; DC: IEEE Computer Society Press; 1999. p. 73–8.

[41] Lommerse G, Nossin F, Voinea SL, Telea A. The visual code navigator: an interactive toolset for source code investigation. In: Proceedings of IEEE InfoVis'05. Washington DC, USA: IEEE Computer Society Press; 2005. p. 24–31.

[42] Shneidermann B. The eyes have it: A task by data type taxonomy for information visualization. In: Proceedings of IEEE Symp on Visual Languages (VL '96). Washington DC, USA: IEEE Computer Society Press; 1996. p. 336–43.

[43] North C. Toward measuring visualization insight. Computer Graphics and Applications, vol. 3(26), Silver Spring, MD: IEEE Press; 2006, p. 6–9.

[44] Möller A, Åkerholm M, Fredriksson J, Nolin M. Evaluation of component technologies with respect to industrial requirements. In: Proceedings of EUROMICRO'04. Washington DC, USA: IEEE Computer Society Press; 2004. p. 56–63.

[45] van Ommering R, van der Linden F, Kramer J, Magee J. The koala component model for consumer electronics, In: IEEE Transactions on Computers, vol. 33(3). Washington, DC, USA: IEEE Computer Society Press; 2000. p. 78–85.

[46] Winter M, Genssler T, Christoph A, Nierstrasz O, Ducasse S, Wuyts R, Arvalo G, Mller P, Stich C, Schnhage B. Components for Embedded Software—The Pecos Approach, Second International Workshop on Composition Languages, ECOOP'02, 2002. ⟨http://www.iam.unibe.ch/~scg/Archive/pecos/public_documents/Wint02a.pdf⟩.

[47] ITEA, ROBOCOP: Robust Open Component Based Software Architecture for Configurable Devices Project—Framework concepts. Public Document V1.0, May 2002, ⟨http://www.hitech-projects.com/euprojects/robocop/⟩.

[48] VTK online: ⟨http://www.kitware.com/⟩.

[49] van Liere R, de Leeuw W. GraphSplatting: visualizing graphs as continuous fields. In: IEEE transactions on visualization and computer graphics, vol. 2(9), IEEE Educational Activities Department, 2003. p. 206–12.