Research

Visual Software Evolution Reconstruction

Marco D'Ambros[1], Michele Lanza[1]

[1]*REVEAL @ Faculty of Informatics - University of Lugano, Switzerland*

SUMMARY

The analysis of the evolution of large software systems is challenging for many reasons, such as the retrieval and processing of historical information and the large quantity of data that must be dealt with. While recent research advances have led to solutions to these problems, a central question remains: How do we deal with this information in a methodical way and where do we start with our analysis?

We present a methodology based on interactive visualizations which support the reconstruction of the evolution of software systems. We propose several visualizations which help us to perform software evolution analysis of a system "in the large" and "in the small", and apply them to 2 large systems.

KEY WORDS: Software Evolution Analysis, Software Visualization

Introduction

Real world software systems require continuous change to satisfy new user requirements, adapt to new technologies and repair errors [22]. As time goes by, software increases in size and complexity, and the original design gradually decays unless maintenance work is done. Indeed, the problem of understanding the evolution of software has become a vital matter in today's software industry. Starting in the early seventies, software evolution has in the meantime become a recognized research field. Its goal is to use the history of a software system to analyse and understand its present state and to predict its future development [4, 14, 15, 23, 28].

Apart from the technical challenges with respect to recovering and modeling the data, the main challenge is how to deal with historical information in a useful and methodical way to understand and reconstruct the phenomenon of evolution itself. Many people regard the history of a system as being the information contained in a versioning system. But the evolution of a software system is not only the collection of all the versions of its components: Developing software is a human activity, and the evolution of a software system therefore also includes all the activities performed by developers, testers and users during the entire history of the system. This additional information comes from various sources such as comments committed by developers during the implementation, problem reports delivered by users and stored in bug tracking systems, mailing list archives, *etc.*

Acquiring a comprehensive understanding of a system's evolution implies two major challenges:

1. *Retrieving and handling the data*. Once the data sources have been defined, the information has to be retrieved, processed and stored for the analysis. While some data sources provide information in a structured way (Bugzilla, a widely used bug tracking system, for example provides problem reports in XML), others need to be treated (for example the log files of CVS). Moreover, it is not trivial to link the different sources. A problem report, for example, refers to one or more software artifacts developed with a versioning system. Since there is no explicit and formal link between them, it must be established with data mining techniques.
2. *Understanding the data*. Once the information has been retrieved and stored, techniques are needed to support its analysis and understanding. They must be able to deal with huge amounts of complex data.

We propose a technique called *software archeology* [3] which, by means of various visualizations, helps us to reconstruct the evolution of a software system in a methodical way. We omit the details on the way we recover the data, but concentrate on the way we use the retrieved data. We perform software archeology in two ways: (1) "in the large" to understand the overall structure and evolution of a system in terms of its high-level components such as modules, and (2) "in the small" to understand the internal structure of the modules, going from the directories down to the level of file versions.

Software Archeology in a Nutshell. To reconstruct the evolution of a system, we need to retrieve information about its history. We use as data sources the CVS and SubVersion versioning systems and the Bugzilla and Issuezilla bug tracking systems. The first step of our approach consists in retrieving the information from these data sources, parsing and storing it in a Release History Database [3, 12]. Then we use interactive visualizations with our BugCrawler tool [6], a major extension of CodeCrawler [20]. BugCrawler uses polymetric views [21] to represent artifacts (*e.g.,* modules, directories, files, bugs) and relationships. We provide a set of views to support archeology in the large (to get an overview of the whole system and the relationships between system modules) and in the small (to see the details of any single system fragment). The visualizations are interactive, providing facilities like searching, zooming and panning and our tool also provides navigation support to make it possible for the user to quickly jump back and forth between the views and easily reach the source code representation. The main idea of our approach is to provide visualizations concerning several aspects of software evolution in order to answer questions that the software "archeologist" may have, such as:

- Commit information: Which are the parts of the system with the most intense development? Which are the stable/dead parts of the system? Which parts have grown/shrunk?
- Author information: How many developers worked on the entity? How was the effort distributed among them? Is there an "owner" of the entity?
- Bugs: Which components are affected by many bugs? Which bugs affect many components?
- Logical coupling: Which artifacts are most coupled?
- Conceptual entities: How has an entity evolved over time? When was it introduced in the system? When did it generate many bugs? When did it have intense development?

In the following sections we present example visualizations to support for large-scale and small-scale archeology. We apply the views on two large case studies, namely Mozilla (http://www.mozilla.org) and Gimp (http://www.gimp.org), both well-known in the open source community.

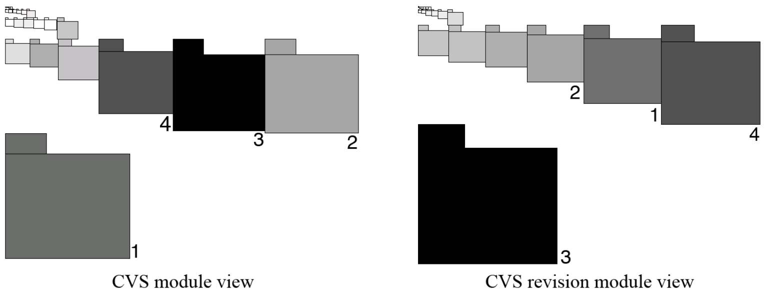


Figure 1. Two visualizations of the Mozilla modules. The color of the figures represents the number of bugs and the size represents the number of files (view on the left) or the development effort (view on the right).

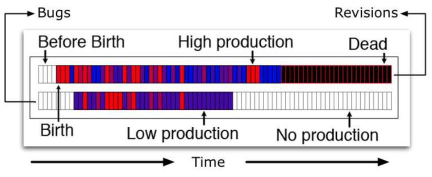
Software Archeology in the Large

Obtaining an Overview. The first goal is to get one's bearings in a system as large as Mozilla. On the left part of Figure 1 we see a simple visualization of figures representing all Mozilla modules, called *CVS module view*. We consider a module as a collection of directories and files. This does not always represent the internal organization of a system, but it allows us to get the module decomposition from CVS (with the command "cvs co -c") and thus to analyze all the systems developed using CVS in the same way. The size of each figure represents the number of files contained in the module and the color represents the number of bugs affecting the module (the darker the figure, the greater the number of bugs). This view helps to answer questions such as: Which are the key modules in the system (big and dark figures), how big are the modules with respect to each other, and where are the most bugs located?

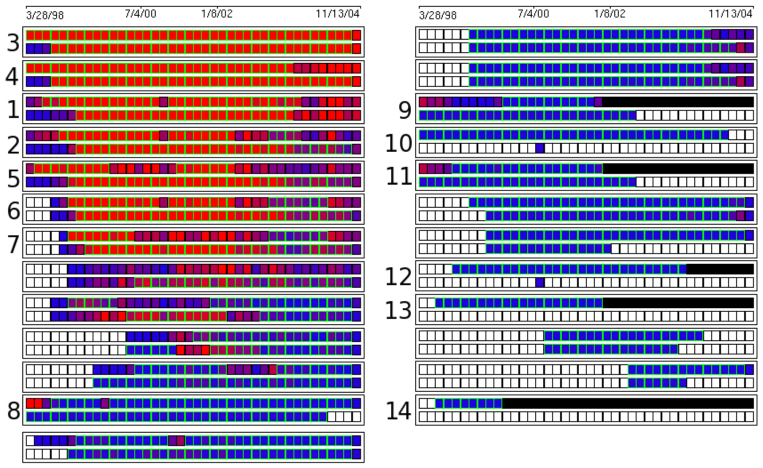
From the left part of Figure 1 we see that there are two types of modules: The big modules affected by many bugs (marked as 1, 2, 3, 4) and the small modules (all the others). The module SeaMonkeyCore (marked as 1) is the biggest in terms of number of files (3266), while SeaMonkeyLayout (marked as 3) is the most affected by bugs (29'412 bug references). To see how the development effort was distributed among the modules, we can use a variation of this view, called *CVS revision module*, where we map the number of commits of each module to the corresponding figure size. The number of commits of a module is equal to the sum of the number of commits of all the files contained in the module. This new view applied on Mozilla (right part of Figure 1) gives us a result similar to the previous one. The four biggest modules in terms of number of files are also the ones with the most intense development.

Taking Time into Account. So far we have obtained a mental picture of the current state of the system and its history. In a sense we have looked at the system as a sum of its previous states. The next thing we want to do is to obtain a picture of how the system has traversed time. We can do that by using a visualization called *Discrete Time Figure* [5], which renders the history of an entity with respect to its development intensity (the number of commits) and its problems (the number of bugs).

The principles of this visualization are shown in Figure 2(a): It has 2 subfigures, each of which is composed of a sequence of rectangles, representing a discretization of time of the revisions and the



(a) Principles of a Discrete Time Figure.



(b) The *Discrete time module view* applied to the Mozilla modules.

Figure 2. Visualizing the evolution of code and bugs with Discrete Time Figures.

bugs, respectively. Each rectangle is associated to a precise and parametrizable interval of time, where 2 vertically aligned rectangles having the same horizontal position represent the same time period. The rectangles are colored using a heat map, *i.e.*, hot colors (in the red hue range) represent time periods with many revisions (or many reported bugs), cold colors (in the blue hue range) represent few commits (or few reported bugs). White rectangles represent time periods without development activity or without reported bugs. Black rectangles represent the time intervals after the removal of the entity from the system (the entity is "dead").

Figure 2(b) shows the *Discrete time module view*, displaying all Mozilla modules with this new perspective. Since the time scale is the same for all the figures, we can understand which parts of the system have changed more frequently, when these changes introduced many bugs and when modules were added to / removed from the system. The development effort was mainly concentrated on the modules SeaMonkeyLayout (3), RaptorLayout (4) and SeaMonkeyCore (1). During the observed time many revisions were committed in these modules and many bugs related to them were reported, indicated by the red rectangles in respectively the first and the second row of each figure.

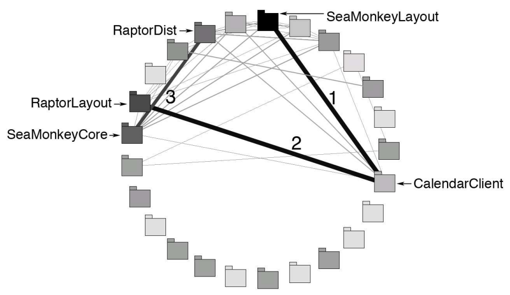
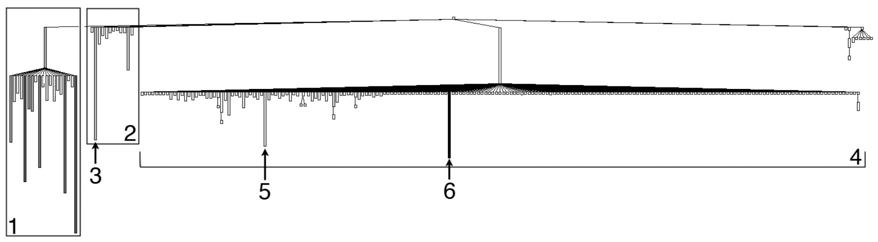


Figure 3. The *Module bug-sharing correlation view* applied to the Mozilla modules.

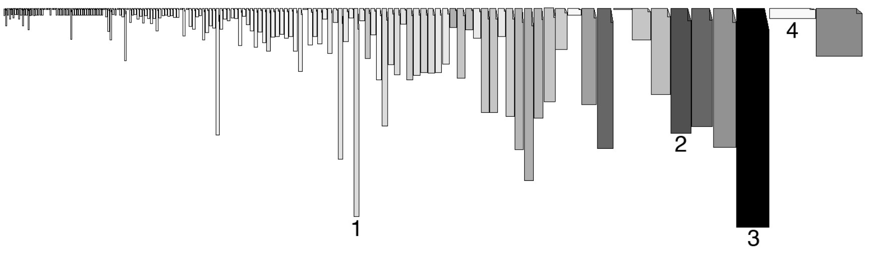
Modules marked as 2, 5, 6 and 7 have had an intense development with many bugs, but in the last part of the system history, their development tended to be less vigorous (blue rectangles on the right part of the figures). For system restructuring purposes, the modules to focus on are the first three (3, 4 and 1) since they generated many bugs and they were heavily changed (many commits) during the entire system lifetime and especially in the recent past. The underlying assumption is that components which changed the most in the recent past are also likely to suffer important changes in the near future [15]. Other observable facts are: The modules marked as 9, 11, 12, 13 and 14 were removed from the system (9, 11 and 13 at the same moment, implying a big change in the system). Moreover, only the modules 1, 2, 3, 4, 5, 8, and 10 were part of the system from the beginning and "survived" until the present.

Discovering Hidden Dependencies Now that we have an overview of the system and its history in terms of the modules that compose it, we want to analyze the relationships between them. We use the dependency of two entities sharing the same bug: A bug is shared by two entities when it affects both of them, *i.e.,* when there is a link between the bug and each entity. The higher the number of shared bugs is, the stronger the dependency between the two entities is. A strong dependency of this type between modules could point to misplaced entities, very much in the spirit of logical coupling [13] (the implicit dependency between software artifacts that frequently changed together during a system's evolution).

In the visualization shown in Figure 3, called *Module bug-sharing correlation view*, we see all Mozilla modules as icons and the bug sharing dependencies as edges. The color of the modules is proportional to the total number of commits of the files they contain, while the width and the color of the edges is related to the number of shared bugs between the two connected modules. The thicker and darker the edge is, the stronger the correlation is. In our tool it is possible to filter out the dependencies characterized by a number of shared bugs below a given threshold: We have done so in the figure eliding edges representing less than 30 shared bugs. The 3 strongest dependencies are between SeaMonkeyLayout and CalendarClient (1), between CalendarClient and RaptorLayout (2) and between SeaMonkeyCore and RaptorDist (3). The first two have more than 500 bugs in common while the third one is characterized by more than 100 shared bugs. CalendarClient has dependencies with 6 other modules, where two of these dependencies are very strong. SeaMonkeyCore plays a central role in the system: It is connected with 8 other modules, pointing to potential signs of architectural decay.



(a) The *Directory tree view*: a visualization of all the directories.



(b) The *File size-effort-bug view* visualizing the files contained in the `app/actions` directory.

Figure 4. Two visualizations of Gimp.

Software Archeology in the Small

Software archeology "in the small" is aimed at understanding the internal structure and evolution of a given module, a directory tree, even a single file or bug. In the following we present 4 visualizations designed for archeology in the small activities.

Detecting Key Directories. In Figure 4(a) we see a visualization of all the directory hierarchies of Gimp called *Directory tree view*. The height of the figures is proportional to the number of files the corresponding directory directly contains (without considering the files contained in children directories), the width is fixed and the color is proportional to the number of bugs. The view is aimed at providing a first insight into the system (or a module) structure: Understanding which are the largest sub-hierarchies, which ones contain many files and/or many recorded bugs. Of particular interest in this visualization are (1) outliers, *i.e.*, figures different from the other figures in the same sub-hierarchy, and (2) tall and dark figures representing directories which contain a lot of files affected by many bugs.

In the *Directory tree view* shown in Figure 4(a) the hierarchy `app` (marked as 1) is the biggest in terms of number of files (2589) and number of bugs (2164). It contains the main application and it is composed of directories characterized by an high number of files and bugs. To choose on which directory or set of directories to continue the analysis, we can apply the *Discrete time directory tree*, which uses Discrete Time Figures to represent directories in a tree layout, to study their evolution.

Another view which can be applied, to study the developers effort, is the *Fractal directory tree*, which represents directory in a tree layout as Fractal Figures, explained in a following section.

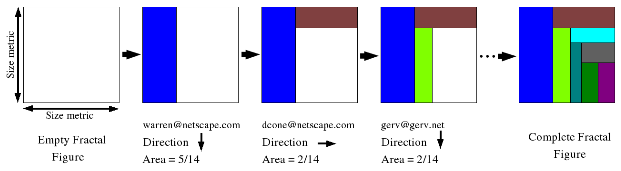
The directories marked as 2 are the libraries, where the directory `libgimp` (marked as 3) is an outlier with 275 files and 104 bugs. It should be further analyzed taking time into account (using the Discrete Time Figure) and at the file granularity. The hierarchy `plug-ins` (marked as 4) is the biggest in term of number of directories. Most of these directories contain one file only (the plug-in) and 0 bugs, with some exceptions. Of particular interest are the directories `plug-ins/imagemap` (marked as 5) with 130 files and 48 bugs and `plug-ins/common` (marked as 6) with 159 files and 423 bugs. Imagemap is a complex plug-in, which is also used as underlying engine by other plug-ins. The following analysis at the file level should certainly focus on the `plug-ins/common` directory, where the common behaviors and interfaces are implemented and where most of the problems related to the plug-ins are located. This directory plays a crucial role for the `plug-ins` hierarchy.

Opening the Lid. After obtaining an idea of the structure and evolution of the directories, we may want to have a look "under the hood" and examine their contents. In Figure 4(b) we see the *File size-effort-bug view* visualizing all the files belonging to the `app/actions` directory of Gimp. Files are represented as rectangle figures, mapping the number of commits on the figure width, the number of lines of code (in the last version of the system) on the figure height and the number of bugs on the figure color. The figures are sorted according to the width metric, *i.e.*, the number of commits the corresponding file has, because this facilitates the identification of outliers.

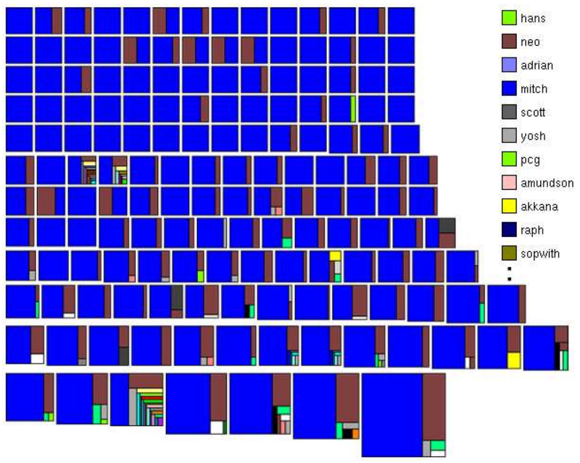
While the LOC metric refers to the last version of the system, the number of commits and bugs are computed according to the entire history of the files, summarizing their evolution in a simple figure. The view is helpful for detecting the biggest files (the tallest figures) that need to be refactored because they generated many bugs (dark figures) and files which had an intense development (wide figures). Tall and narrow figures represent big files with few commits. Such a pattern is due to copy-pasted code or a late insertion of the file in the repository. However, to know which is the case we need to look at the history and/or the source code of the file. The file `context-actions.c` (marked as 1 in Figure 4(b)) is characterized by this pattern (more than 1 KLOC and 18 commits). Looking at its history we found out that it was inserted late in the repository (in the last 1.5 years). Reading the source code we discovered that the file contains only constant definitions, another reason which explains the small number of commits. In Figure 4(b) we also have the opposite pattern: Wide and short figures, representing files with few lines of code in the last version of the system, which passed through an intense development. The file `help-commands.c` (marked as 4) has 20 LOC and 218 commits. The inspection of the history of its source code revealed that it was present from the first version of the system and from that point on its size (in terms of LOC) has constantly decreased. At the beginning it contained the implementation of all the actions of the help menu and then, over the history, more and more menu item actions were moved to other files to which `help-commands.c` delegates the calls.

The files most affected by bugs are `layers-commands.c` (1 KLOC, 35 bugs) and `image-commands.c` (600 LOC, 24 bugs), marked as 3 and 2 respectively. `layers-commands.c` implements the actions present in the layer menu of Gimp, while `image-commands.c` implements the image menu actions. The high number of bugs calls for a detailed inspection.

What about the Human Factor? The previous view provided an understanding of the evolution of files from the point of view of commits and bugs. Now we also want to see how many developers worked



(a) The structural principles of a Fractal Figure.



(b) The *Fractal file view*: Fractal Figures applied to Gimp files.

Figure 5. Visualizing development effort with Fractal Figures.

on the files and how the development effort was distributed among them. We can do that by using a specific visualization called *Fractal Figure* [8]. A Fractal Figure is composed of a set of rectangles having different sizes and colors. Each rectangle, and each color, is mapped to an author who worked on the artifact. The area of the rectangle is proportional to the percentage of commits performed by the author over the whole set of commits. Fractal Figures can be enriched by rendering a software metric measurement on their size. Looking at a Fractal Figure (see Figure 5(a)) we can easily figure out whether the development was done mainly by one author or many people contributed to it and in which terms. Fractal Figures are similar to slice-and-dice treemaps [25] which recursively partition the planar display area along both dimensions, alternatively vertically and horizontally. Fractal Figures allow the definition of four different patterns [8] (one developer, few balanced developers, one major developer and many balanced developers), according to the *gestalt* principle, with which the user can immediately understand how the development effort was distributed among the authors.

Figure 5(b) shows the *Fractal file view*, a visualization of all the files belonging to the `app/actions` directory of Gimp. The view shows the same entities of the visualization shown in Figure 4(b), but from a different perspective: The developer contributions. Files are represented as

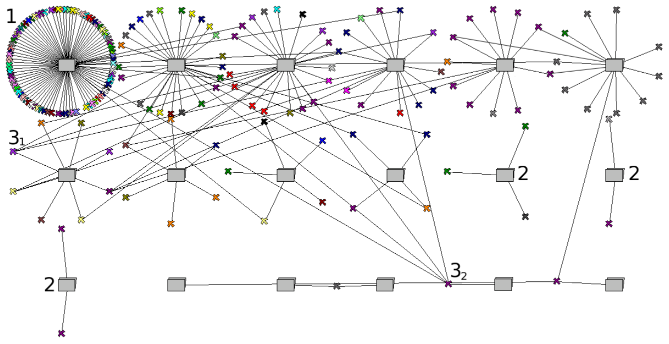


Figure 6. Detailing dependencies in Mozilla with the *Directory bug dependency view*.

Fractal Figures, where the size of each figure is proportional to the number of bugs the corresponding file is affected by. The view is aimed at understanding how the development effort is distributed among the different authors, according to the code ownership principle. Is there an author mainly responsible for the development of the files? Or is the development effort equally distributed on many authors? Do most of the files have the same development pattern?

In the directory `app/actions` of Gimp (see Figure 5(b)) most of the files have the same pattern. The author related to the blue color (*mitch*, as shown in the legend in the right part) is mainly responsible for their development. This information should be carefully interpreted. There is no one-to-one mapping between developers and CVS accounts: A developer can have multiple CVS accounts and a CVS account can "hide" several developers behind it. In the situation shown in Figure 5(b) it can either be that *mitch* developed most of the `app/actions` code or *mitch* is a "*proxy*", *i.e.*, an author who is responsible to collect patches and commit them to the repository. This is a common practice for open source projects, where the write permission to the repository is given to few people and patches are sent to them via e-mail. To verify whether *mitch* was a proxy, we contacted him and discovered that he is actually Gimp's main developer, responsible for most of the `app` code.

Detailing Hidden Dependencies. The last activity in the large we have presented was discovering hidden dependencies among modules, based on bug-sharing. Now we want to zoom-in onto these dependencies and see the details about bug-sharing: Which low-level entities share bugs and which bugs have the biggest impact? We do that by means of the *Directory bug dependency view*, shown in Figure 6. A set of directories are placed within a grid. For each directory, all its bugs are positioned around it in a circle. The directories are represented as folders, the bugs as crosses. The color of the bugs maps the bug owner information: The same color represents the same owner. Bugs with multiple edges are shared bugs: The greater the number of edges, the more directories are sharing the bug.

The visualization shows two interesting facts: "Self-contained" directories and bugs linked with many directories. Self-contained directories are characterized by not sharing bugs or by having a small amount of shared bugs (with respect to the entire set of bugs). They are likely to encapsulate specific

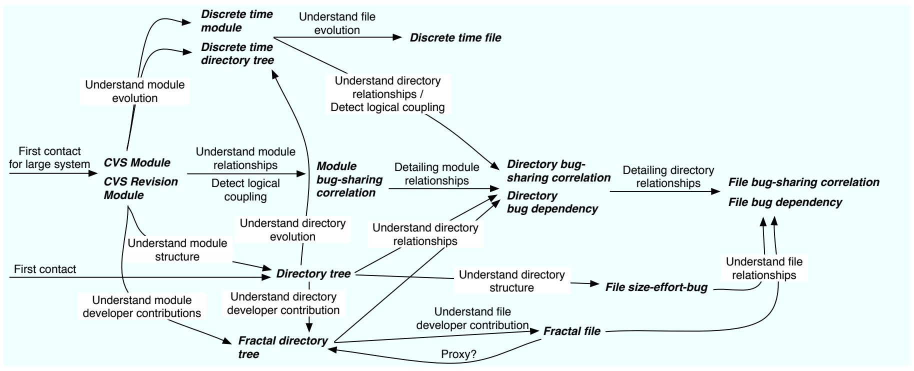
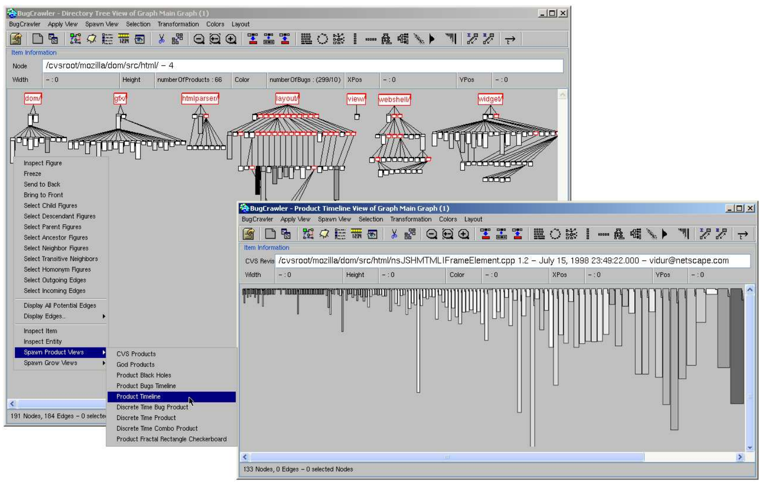

Discrete time module
Discrete time directory tree
Understand file evolution
Discrete time file
Understand directory relationships / Detect logical coupling
Understand module evolution
First contact for large system
CVS Module
CVS Revision Module
Understand module relationships
Detect logical coupling
Module bug-sharing correlation
Detailing module relationships
Directory bug-sharing correlation
Directory bug dependency
Detailing directory relationships
File bug-sharing correlation
File bug dependency
Understand directory evolution
Understand directory relationships
Understand module structure
First contact
Directory tree
Understand directory structure
File size-effort-bug
Understand file relationships
Understand module developer contributions
Understand directory developer contribution
Understand file developer contribution
Fractal directory tree
Fractal file
Proxy?

Figure 7. A general schema of our visual software evolution reconstruction approach. The schema suggests which view should be applied according to the goal we want to achieve.

data or responsibilities, especially if the number of different bug owners is small. Examples of this pattern are the directories marked as 2 in Figure 6 (not sharing bugs) and the directory marked as 1, which has 6 shared bugs on a total of 90. The directory marked as 1 needs to be further analyzed at the file level, given the high number of bugs, while the directories marked as 2 do not need further inspection, since they are affected by two or three bugs only. Bugs linked with many directories point to hidden dependencies between the directories and to potentially misplaced files. Looking at their description and comment fields is useful to understand the directories' responsibilities and the reasons why they are shared. The two bugs marked as 3_1 and 3_2 in Figure 6 are respectively shared by four and five directories. Reading the bug comments we discovered that the problems are related to: “A virtual function that should not be virtual” for the bug marked as 3_1 and “A general purpose stack which creates a lot of confusion” for the bug marked as 3_2. Since the bug status field is “fixed” for both the bugs, we found also the patches for these problems (patches are part of our bug metamodel, and they can be reached from the context menu displayed when clicking on a bug figure). The same visualization can be applied at the file level with the same principles, substituting directories with files.

Methodology and Tool Implementation

Methodology. Figure 7 shows an overview of our approach to reconstruct the evolution of a software system. The schema is a graph in which each node is a particular visualization (or set of visualizations) and an edge going from the node A to the node B with label L indicates that from view A, to achieve the goal described in L we should apply the view B. The schema is not strict: For example, the fact that from a node C to a node D there is no edge, does not mean that it is not possible to apply the view D from C. Figure 7 shows the most common “path” according to our experience in using the tool. Such a schema is useful for new users, who are learning how to use and interpret the different visualizations to reconstruct the evolution of a system.


BugCrawler - Directory Tree View of Graph Main Graph (1)
BugCrawler Apply View Spawn View Selection Transformation Colors Layout
Item Information
dom/
gfx/
htmlparser/
layout/
view/
webshell/
widget/
Inspect Figure
Freeze
Send to Back
Bring to Front
Select Child Figures
Select Descendant Figures
Select Parent Figures
Select Ancestor Figures
Select Neighbor Figures
Select Transitive Neighbors
Select Homonym Figures
Select Outgoing Edges
Select Incoming Edges
Display All Potential Edges
Display Edges...
Inspect Item
Inspect Entity
Spawn Product Views
Spawn Grow Views
CVS Products
God Products
Product Black Holes
Product Bugs Timeline
Product Timeline
Discrete Time Bug Product
Discrete Time Product
Discrete Time Combo Product
Product Fractal Rectangle Checkerboard
191 Nodes, 184 Edges – 0 selected
BugCrawler - Product Timeline View of Graph Main Graph (1)
BugCrawler Apply View Spawn View Selection Transformation Colors Layout
Item Information
133 Nodes, 0 Edges – 0 selected Nodes

Node /cvsroot/mozilla/dom/src/html/ – 4
Width – : 0 Height numberOfProducts : 66 Color numberOfBugs : (299/10) XPos – : 0 YPos – : 0

CVS Revis /cvsroot/mozilla/dom/src/html/nsJSHMTMLIFrameElement.cpp 1.2 – July 15, 1998 23:49:22.000 – vidur@netscape.com
Width – : 0 Height – : 0 Color – : 0 XPos – : 0 YPos – : 0

Figure 8. BugCrawler in action: The window on the top left is showing a *Directory Tree* visualization and from a selected directory a *File size-effort-bug* view is spawned, visualizing all the files contained in the directory.

Tool Implementation. Our environment for visual software evolution reconstruction is composed of the importers, the Release History Database (RHDB) and BugCrawler. Both the importers and BugCrawler are written in Smalltalk. The importers retrieve the data from CVS/SubVersion repositories and Bugzilla/Issuezilla bug tracking systems, parse and process it (*e.g.*, link CVS artifacts with Bugzilla problem reports) and store it on the RHDB. The importers are accessible from a web interface, part of the Churrasco framework [7], where all the pieces of information needed to run the importers are the url of the repository and/or bug database. Once a repository is imported, it is automatically and periodically updated. The time required to import and process a CVS repository or a Bugzilla database mostly depends on the time needed to checkout the repository or download the bug reports. Checking out the entire Mozilla CVS repository took about 2 hours, while downloading more that 160'000 Mozilla bug reports took more than 40 hours. Parsing and processing the data (CVS log files and bug reports) took less than 1 hour.

Figure 8 shows BugCrawler "in action": Each window renders a visualization: The one on the top left is a *Directory tree*, the second one is a *File size-effort-bug* (a visualization of all the files contained in a directory selected in the first view). Navigating between views is possible through context menus: The menu items depend on the entity represented by the selected figures. For example the menu of a directory allows the rendering of views for the files it contains.

BugCrawler provides flexibility to the user, by allowing him/her to customize the views parameters (*e.g.*, figures, layouts, metrics mapping) and to design new visualizations on-the-fly. However, for non expert users, an extensive set of predefined visualizations is available, and these can be applied by just selecting a menu item. Scalability is provided by using visualizations in the large to see the entire system, and then by focusing on the interesting parts with views in the small. The visualizations in the tool are interactive: It is possible to inspect any entity in the visualization and to reach its original data source (source file, problem report) using context menus.

Related Work

Visualization has long been used in software evolution research to break down the data quantity and complexity [2, 17, 19, 24, 26, 28]. Many approaches consider different releases of a software system (snapshots) and visualize their history and their differences. Jazayeri *et al.* in [19] used a three-dimensional visual representation for analyzing a software system's release history. In [18] Jazayeri proposed a retrospective analysis technique to evaluate architectural stability, based on the use of colors to depict changes in different releases. In [27], Tu and Godfrey proposed an approach which integrates the use of metrics, software visualization and origin analysis for studying software evolution. Girba *et al.* used the notion of history to analyze how changes appear in the software systems [15] and succeeded in visualizing the histories of evolving class hierarchies [17]. The main difference between these approaches and ours is that we consider the fine-grained history of a software system, *i.e.,* all the versions of all the software artifacts, while the listed techniques consider snapshots of the system.

Another approach to software evolution visualization consists in retrieving the history of a software system from versioning system log files. Ball and Eick [1] focused on the visualization of different source code evolution statistics such as code version history, difference between releases, static properties of code, code profiling and execution hot spots, and program slices. Taylor and Munro [26] used visualization together with animation to study the evolution of a CVS repository. The technique, called revision towers, allows the user to find out where the active areas of the project are and how work is shared out across the project. Rysselberghe and Demeyer [28] used a simple visualization of CVS data to recognize relevant changes in the software system such as: (1) unstable components, (2) coherent entities, (3) design and architectural evolution, and (4) fluctuations in team productivity. In [32] Wu *et al.* used the spectograph metaphor to visualize how changes occur in software systems. Girba *et al.* [16] analyze how developers drive software evolution by visualizing code ownership based on information extracted from CVS log files.

A number of approaches use information from both different releases of a software system and versioning system log files. The EvoGraph visualization [11] combines release history data and source code changes to assess structural stability and recurring modifications. Pinzger *et al.* [24] proposed a visualization technique based on Kiviat diagrams which provides integrated views on source code metrics in different releases together with coupling information computed from CVS log files. Collberg *et al.* proposed a graph drawing technique for visualization of large graphs with temporal component, with the aim of understanding the evolution of legacy software [2]. The main difference between the mentioned approaches and ours is that these visualizations do not provide bug related information, while our visualizations integrate CVS log file and bug report data. Voinea and Telea [30] proposed the CVSgrab tool which supports querying, analysis and visualization of CVS based software repositories, integrating also Bugzilla information. Their tool allows the user to produce views, to interact with them, to do querying and filtering and to customize the view through a rich set of metrics computed from the CVS data. They applied CVSgrab to assess change propagation of buggy files [29]. The same authors in [31] proposed several visualization techniques (and the corresponding tools), with the aim of supporting software engineers manage the evolution of large and complex software systems. A point in favour of their toolset, with respect to ours, is that it covers a wide granularity spectrum, from the evolution of the entire system / subsystem level, down to the evolution of single lines of code, while our finest granularity is at the level of files. A difference to BugCrawler is that it supports different evolutionary perspectives using different figures (*e.g.*, Fractal Figures for the author perspective) and layouts, while in Voinea and Telea toolset this is supported by changing the color scheme, superimposing another visualization layer and using clustering.

Ducasse *et al.* proposed a general visualization technique, called Distribution map [9], to analyze how properties are distributed in a software system. A benefit of this technique over BugCrawler is its generality, since it is applicable to any property. In [10] Ducasse *et al.* introduced the Package surface blueprint, a visualization approach to study the relationships among the packages of a system. The main difference with our approach is that they address specifically the problem of understanding package relationships, rendering packages and classes, while we study various perspectives of software evolution, visualizing modules, directories, files and bugs.

Conclusion

We have presented a visual approach for reconstructing the evolution of a software system which relies on information residing in versioning systems and bug tracking systems. We discussed several visualizations aimed at understanding the various aspects of the evolution of a system. The technique supports the analysis of a system's history "in the large" and "in the small". Our approach features many additional visualizations that we did not list because of space reasons, which can be found in [3].

The main contribution of this paper, in particular with respect to previously published work presenting some of the visualization techniques used here [5, 8], is that it addresses the problem of understanding software evolution "in the large", *i.e.*, starting from just a CVS (or SubVersion) repository and a Bugzilla (or Issuezilla) database, considering various aspects of the system's evolution, and going down to the single file history. In this paper we also proposed a methodology which provides a systematical way to address the problem of understanding software evolution. It allows the software archeologist to get a complete picture of the evolution of a software system, by combining several aspects of the evolution of its components.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation (SNF Project No. 118063, 'DiCoSA - Distributed Collaborative Software Analysis").

REFERENCES

1. T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.
2. C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86. ACM Press, 2003.
3. M. D'Ambros. Software archaeology - reconstructing the evolution of software systems. Master thesis, Politecnico di Milano, Apr. 2005.
4. M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pages 189–198. IEEE CS Press, 2006.
5. M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of CSMR 2006 (10th IEEE European Conference on Software Maintenance and Reengineering)*, pages 227–236. IEEE CS Press, 2006.
6. M. D'Ambros and M. Lanza. Bugcrawler: Visualizing evolving software systems. In *Proceedings of CSMR 2007 (11th IEEE European Conference on Software Maintenance and Reengineering)*, pages 333–334. IEEE CS Press, 2007.

7. M. D'Ambros and M. Lanza. A flexible framework to support collaborative software evolution analysis. In *Proceedings of CSMR 2008 (12th IEEE European Conference on Software Maintenance and Reengineering)*, pages 3–12. IEEE CS Press, 2008.
8. M. D'Ambros, M. Lanza, and H. Gall. Fractal figures: Visualizing development effort for cvs entities. In *Proceedings of Vissoft 2005 (3th IEEE International Workshop on Visualizing Software for Understanding)*, pages 46–51. IEEE CS Press, 2005.
9. S. Ducasse, T. Gîrba, and A. Kuhn. Distribution map. In *Proceedings International Conference on Software Maintainance (ICSM 2006)*, 2006.
10. S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *Proceedings IEEE International Conference on Software Maintainance (ICSM 2007)*, pages 94–103, Los Alamitos CA, Oct. 2007. IEEE CS Press.
11. M. Fischer and H. C. Gall. Evograph: A lightweight approach to evolutionary and structural analysis of large software systems. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, pages 179–188, Benevento, Italy, October 2006. IEEE Computer Society.
12. M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, Los Alamitos CA, Sept. 2003. IEEE Computer Society Press.
13. H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE CS Press.
14. H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM'97)*, pages 160–166. IEEE CS Press, 1997.
15. T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 40–49, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.
16. T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE CS Press, 2005.
17. T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 2–11. IEEE CS Press, 2005.
18. M. Jazayeri. On architectural stability and evolution. In *Reliable Software Technologies-Ada-Europe 2002*, pages 13–23, Berlin, 2002. Springer Verlag.
19. M. Jazayeri, H. Gall, and C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 99–108. IEEE CS Press, 1999.
20. M. Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.
21. M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.
22. M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.
23. T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings IWPSE2001 (4th International Workshop on Principles of Software Evolution)*, pages 83–86, 2001.
24. M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.
25. B. Shneiderman. Tree visualization with tree-maps: A 2-D space-filling approach. *ACM Transactions on Graphics*, 11(1):92–99, 1992.
26. C. Taylor and M. Munro. Revision towers. In *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50, Los Alamitos CA, 2002. IEEE Computer Society.
27. Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *10th International Workshop on Program Comprehension (IWPC'02)*, pages 127–136. IEEE Computer Society Press, June 2002.
28. F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.
29. L. Voinea and A. Telea. How do changes in buggy mozilla files propagate? In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 147–148, New York, NY, USA, 2006. ACM.
30. L. Voinea and A. Telea. An open framework for cvs repository querying, analysis and visualization. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 33–39. ACM, 2006.
31. L. Voinea and A. Telea. Visual data mining and analysis of software repositories. *Computers & Graphics*, 31(3):410–428, 2007.
32. J. Wu, R. Holt, and A. Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89, Los Alamitos CA, Nov. 2004. IEEE CS Press.