Software Bugs and Evolution: A Visual Approach to Uncover Their Relationship

Marco D'Ambros and Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland

Abstract

Versioning systems such as CVS exhibit a large potential to investigate and understand the evolution of large software systems. Bug Reporting systems such as Bugzilla help to understand which parts of the system are affected by problems. In this article we present a novel visual approach to uncover the relationship between evolving software and the way it is affected by software bugs. By visually putting the two aspects close to each other, we can characterize the evolution of software artifacts. We validate our approach on 3 very large open source software systems.

1. Introduction

Studying the evolution and understanding the structure of large *legacy systems* are two key issues in software industry that are being tackled by academic research.

These problems are strongly coupled for various reasons: (i) examining the structure of subsystems allows us to gain a better understanding of the whole system evolution [7, 13]; (ii) the histories of software entities can reveal hidden relationships among them [6, 8] and (iii) analyzing several versions of a system improves our understanding of it [9–12].

Many studies were done and many approaches were proposed to tackle the problem of studying the evolution of software systems [7, 11, 21], the understanding of systems structures [16, 17] and both of them [4, 12]. However, only few approaches [18] simultaneously provide structural and historical information.

In this paper we introduce the *Discrete Time Figure*, a visualization technique in which both historical and structural data are embedded into one simple figure.

The evolution of a software system is not only about the histories of its software artifacts, but also includes other kinds of information concerning its development, such as documentation, use cases, test suites, *etc.*

We consider, in addition to the development information as retrieved from a versioning system, data regarding the *history of software problems* which are stored in bug reporting systems, such as Bugzilla (see http://www.bugzilla.org/).

Structure of the paper. In Section 2 we discuss goals and challenges in this research domain. We then present the principles behind our approach, which allow us to characterize the evolution of software artifacts using both evolutionary information and bug information. We then validate our approach on three large systems (Apache, gcc, Mozilla). Before concluding and looking at related work, we present the implementation details behind our approach.

2. Bugs and Evolution

The main problem in combining structural information obtained from the source code and versioning systems with other kinds of information (bug reports, documentation, *etc.*) is that the latter is largely unstructured. This prevents us from automating the process of combining them.

Specifically, in this paper we target information obtained from bug reports that we combine with structural and evolutionary information. The objective is to make sense of the integrated information to accomplish the following goals:

- Show the evolution of software entities at different granularity levels in the same way, *i.e.*, using the same visualization.
- Show both structural information (such as software metrics) and bug-related information.
- Merge all these kinds of information to obtain a clearer picture of the evolution of a system's entities. This is a key knowledge for a reengineering activity, since it allows us to detect the critical parts of the system which represent the starting point for a reengineering process.

There are technical challenges in achieving the above goals, such as:

1. *Data retrieval*: Retrieving the histories of files and bugs and combining them.

2. *Visualization*: Showing a large amount of information in a condensed, yet useful way. This also relates to scalability, *i.e.*, the visualization must work at all granularity levels of systems of millions of lines of code.

3. Visualizing Evolving software

Our approach is largely based on visualization, because it can help (if correctly used) to condense complex information into figures that humans can easily interpret.

Figure 1. A *TimeLine View* of revisions and bugs.

A possible way to visualize the evolution of software entities in terms of revisions and bugs is depicted in Figure 1: It shows a *TimeLine View* (time on the horizontal axis from left to right) where the visualized rectangles and crosses represent respectively revisions and bugs on CVS products (*i.e.*, files). In this figure we see the evolution of 2 specific files between 2001 and 2003: The exact horizontal position of rectangles is determined using the CVS commit time-stamp, while for the positions of the crosses the bug report time-stamp is used. The colors represent the different authors (*i.e.*, the person who committed the file revision to the CVS repository) for the revisions. For the bugs they represent the status [2].

Using this figure we detect the files having a lot of revisions (and/or bugs) and which are those characterized by small amount of them. Still, it is limited in the following ways: (1) It does not provide neither a qualitative nor a quantitative impression about the productions of bugs and revisions over time; (2) It is applicable only to files; and (3) It does not scale well, if the number of products, bugs or revisions is high.

To have a scalable view we devised a figure that encapsulates all the production-related (bugs and commits) information of a software entity: *The Discrete Time Figure*.

The Discrete Time Figure gives an immediate view of the history of a software entity. This history includes the growing / shrinking in terms of number of commits[1] and in terms of number of bugs. The history can be enriched with a software metric and structural information given by the view layout (*e.g.*, the directory hierarchy).

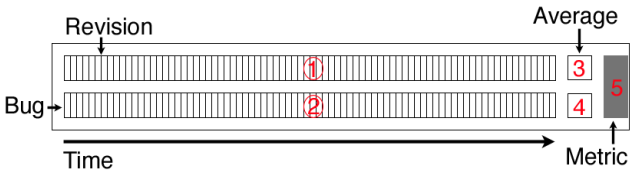

Figure 2. The Discrete Time Figure structure.

Figure 2 depicts the structure of a Discrete Time Figure. It is composed of five subfigures: The first and the second are composed by a sequence of rectangles which represent a *Discretization of time*. It means that each rectangle is associated to a precise and parametrizable interval of time, where rectangles having the same horizontal position (belonging to different subfigures) represent the same time period.

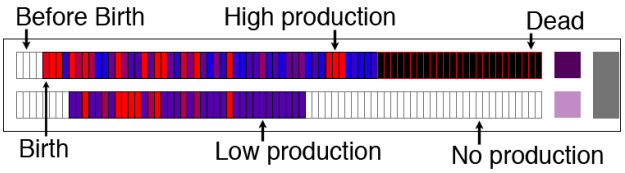

Figure 3. The Discrete Time Figure color mapping.

These sequences of rectangles are then colored according to a "temperature / production" policy (see Figure 3). Hot colors (in the red gamma) are used for time periods during which a lot of revision were commited (or a lot of bugs were reported), while cold colors (in the blue gamma) represent few commits (or few reported bugs).

In Table 1 we have summarized all colors we use in our figures.

The choice of the color is based on a set of thresholds, which can be either manually chosen or automatically computed. For the latter we distinguish two different scenarios:

1 If the software entity is a directory (or module) the number of commits is the sum of the number of commits of all the files included in the directory (or module).

Color	Inner area	Border	Meaning
Red	X		High production
Blue	X		Low production
White	X		No production
Black	X		Dead
Gray		X	No production
Red		X	Dead
Orange		X	High Stable
Green		X	Stable
Pink		X	Spike

Table 1. The colors used in the Discrete Time Figure and their meanings.

1. *Local threshold*. The threshold values are computed "locally" for each figure, taking into account only the bugs and the revisions belonging to the target entity. The local thresholds, as we will see in Section 4, are used to characterize software entities.
2. *Global threshold*. The threshold values are computed "globally", taking into account all the bugs and the revisions belonging to the visualized entities. These thresholds can be used for comparisons, since all the visualized figures refer to the same thresholds.

The third and the fourth subfigures are colored respectively with the "average" color of the first (revisions) and the second (bugs) subfigures. They provide an immediate and course-grained idea of the history of the artifacts: For the revisions box, a color in the red gamma implies high activity, as opposed to low activity (blue gamma). For the bugs box a color close to black signifies that the entity was "dead" (not present) most of the lifetime of the system; close to white implies that the entity has only recently been created in the system. This allows us to compare more easily the different entities.

The fifth subfigure is used to map a metric measurement on its color, using the grayscale values. The darker the color is, the highest the metric measurement is. We do not always make use of it, but it proved efficient for higher-level entities like directories, where this box can be used to represent the number of contained files, etc.

Scalability issues. The Discrete Time Figure provides a lot of evolution information in "one shot", but it still does not scale well (*i.e.*, the color of the inner rectangle becomes undecipherable if the whole figure is small).

A first solution to this problem consists in coloring not only the rectangle area but also the border. However in this way the figure is difficult to read, because of the high density of different colors in small areas.

The solution is to aggregate a sequence of rectangles into a bigger rectangle. We do this by introducing the concept of *Phases*. In Figure 4 we see a Discrete Time Figure with the phase layer.

We define the following phases:

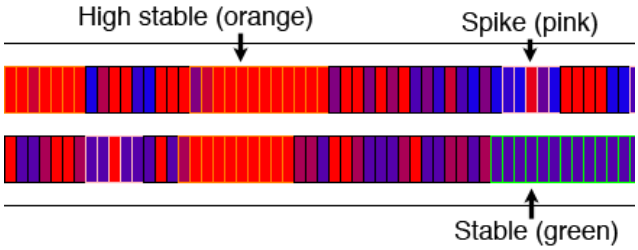

Figure 4. Introducing phases in a Discrete Time Figure.

- *Stable*. A Stable phase is defined as a sequence of (at least four) adjacent time intervals during which the number of revisions (or bugs) remains constant or has small variations around a low value (*i.e.*, a sequence of at least four blue rectangles).
- *High Stable Phase*. It is the same as a Stable phase with the only difference that the number of revisions (or bugs) must remain high (*i.e.*, a sequence of at least four red rectangles).
- *Spike*. This phase is characterized by the presence of: (i) an initial sequence of (at least two) time intervals during which the number of revisions (or bugs) remains low (*i.e.*, a sequence of at least two blue rectangles) (ii) a basic time interval during which this number is high (*i.e.*, a red rectangle) and (iii) a final sequence similar (with the same characteristics) to the initial one.

We also display the fact the entities are born and die within a system: We use different colors for both the inner part and the boundary of the rectangles representing periods of time preceding the birth or following the dead of the artifact. In Figure 3 we see that white rectangles with gray boundary are used for the former[2] while black rectangles with red boundary are used for the latter. Since this information is mapped on the border it is readable at any scale. In this case the problem of having a high density of different colors in small areas does not subsist since these time intervals are usually composed of several basic periods.

Figure 5 shows a tree of Discrete Time Figures with the phase layer applied: Only the phases and the born/death information is readable.

Coloring the rectangle boundary is useful in large scale views, but at the same time it makes the figure slightly confusing in small scale views, especially for inexperienced

2 White rectangles with gray boundary represent also no production (0 commit or 0 bug report) if they are after the birth.

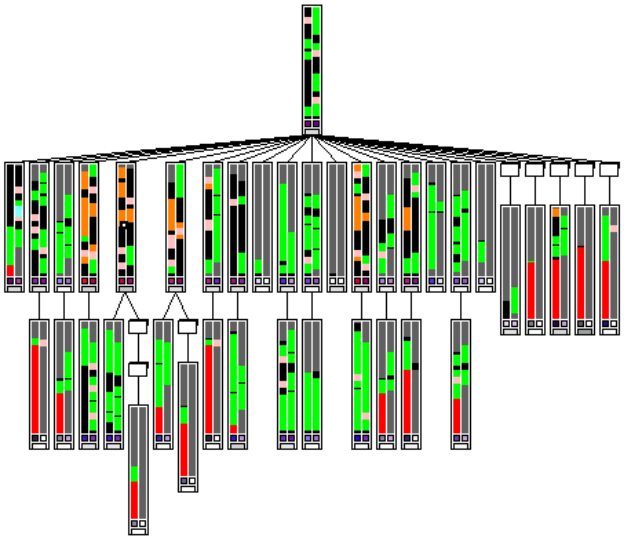

Figure 5. Looking at many figures at the same time does not affect the readability and the means of interpretation.

users. To address this problem in our tool implementation the user can dynamically set the boundary coloring according to how big is the part of the system he is analyzing.

4. Characterizing the Evolution of Software Entities

Using the previously described figure we can characterize the evolution of software entities by taking into account and combining both the development activity and the problems affecting them. We distinguish the following types:

Persistent. An artifact is defined as Persistent if (i) it is still alive (in the current version of the software) and (ii) its lifetime is greater than the 80% of the system lifetime. These kind of entities are likely to play important roles in the system because they survived most of the system changes.
We distinguish two types of persistent entities:
(1) Bug Persistent. The bug coverage is at least 80% of the revision coverage[3]. On one hand the presence of this pattern can be a good symptom because it indicates that the most of the development of the artifact was done together with testing. On the other hand the pattern can indicate that the entity was affected by problem for most of its lifetime, which is a symptom of bad design.
(2) High Production Persistent. At least 80% of the basic time intervals (represented by rectangles) are characterized by a large number of revisions (represented by red colors). The High Production Persistent entities have key roles in the system because the changes are concentrated on them. They are a good place to start reverse engineering activities.

Figure 6 shows examples of Bug Persistent and High Production persistent entities.

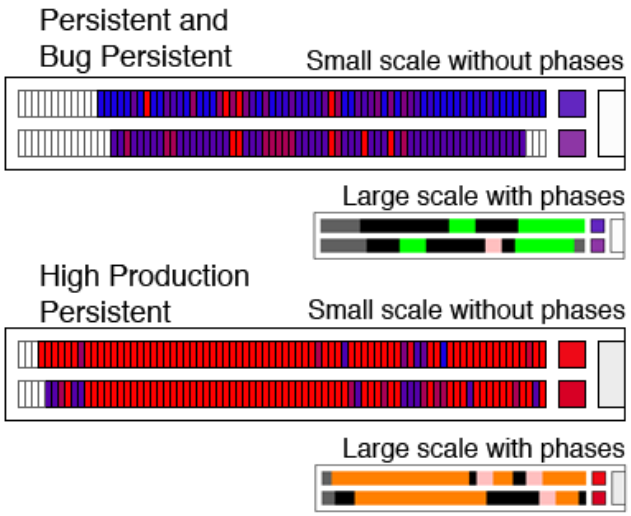

Figure 6. Examples of the various type of Persistent patterns.

Day-Fly. An entity is defined as Day-Fly if its revision coverage is no more than 20% of the system lifetime. We defined also a **Dead Day-Fly** as an artifact removed from the system which lasted in it at most 20% of the entire lifetime of the system.

A Dead Day-Fly can be a symptom of: (i) renaming[4], (ii) fast prototyping and (iii) a new implementation quickly removed because it violated the design.

A Day-Fly can indicate: (i) a spike solution no more under development or (ii) a forgotten part of the system. Figure 7 shows examples of Day-Fly and Dead Day-Fly entities.

3 Since the revision coverage for a Persistent entity is at least 80% of the system lifetime, the bug coverage is at least $80\% \times 80\% = 64\%$ of the system lifetime.

4 In CVS a renaming appears as a removal of an artifact (the old name) and an addition of another artifact (the new name).

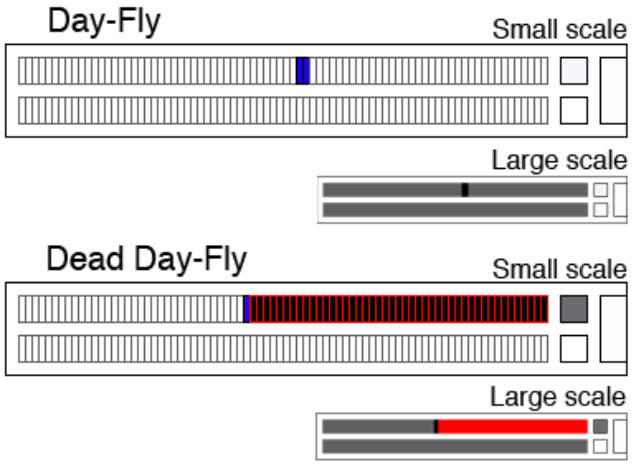

Figure 7. Examples of a Day-Fly and a Dead Day-Fly.

Introduced for Fixing. We define an entity as Introduced for Fixing if its first revision was introduced in the system after the first bug affecting it was reported[5], *i.e.*, the first colored rectangle in the bug subfigure precedes the first colored rectangle in the revision subfigure (as shown in Figure 8).

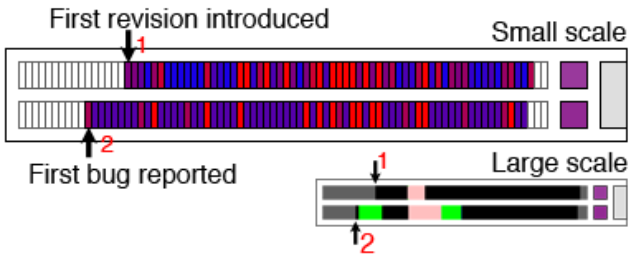

Figure 8. An example of the Introduced for Fixing pattern.

Stabilization. We define an entity as Stabilization if: (i) it is still living, (ii) it had an intense development (at least five basic intervals characterized by a high production and at least one High Stable or Spike phase) for both revisions and bugs and (iii) the end of its histories for both bugs and revisions are characterized by Stable phases. Figure 9 shows an example of the Stabilization pattern.

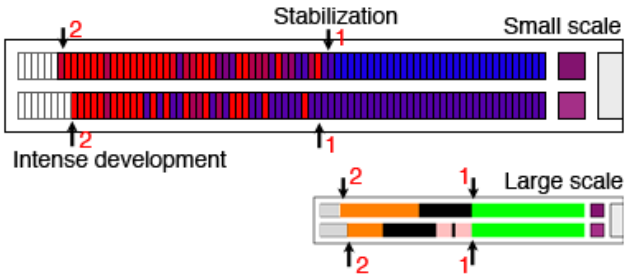

Figure 9. An example of the Stabilization pattern.

The following patterns, differently from the previous, are not related to the entire entity but only on parts of it. This means that more than one pattern can be found on the same figure, *i.e.*, the same artifact.

Addition of Features. An Addition of Features in the system consists in an increasing of the number of revisions (a lot of commits) together with an increasing of the number of bugs. Introducing new features is likely to introduce new bugs. Graphically this pattern can be detected by the presence of the pairs of phases listed below. We describe also the time relations between phases where $t1_{begin}$ and $t1_{end}$ are respectively the beginning and the end of the first phase, while $t2_{begin}$ and $t2_{end}$ have the same meaning for the second phase.

- *Spike* - *Spike* and *Spike* - *High Stable*.

$$t1_{begin} \leq t2_{begin} \leq t1_{end} + 2 \text{ basic time interval}$$

- *High Stable* - *High Stable* and *High Stable* - *Spike*

$$t1_{begin} \leq t2_{begin} \leq t1_{end} - 2 \text{ basic time interval}$$

Examples of this pattern are shown in Figure 10.

Bug Fixing. A Bug Fixing pattern is characterized by an increasing number of revisions (a lot of commits) together with a decreasing number of bugs. The effort revealed by the increasing number of revisions was spent to fix problems. The pairs of phases which indicate this pattern are:

- *Spike* - *Stable*.

$$t1_{begin} \leq t2_{begin} \leq t1_{end} + 2 \text{ basic time interval}$$

5 It is possible because a bug can affect many entities.

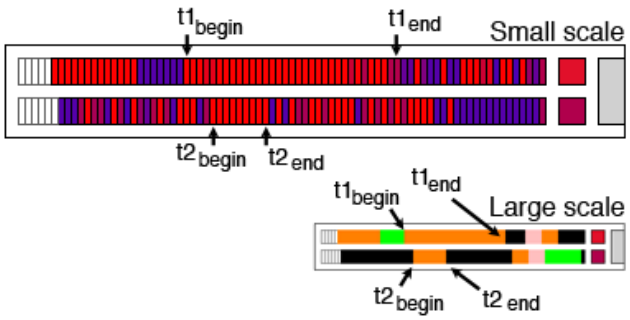

Figure 10. An example of the Addition of Features pattern with the High Stable - High Stable pair of phases.

- *High Stable - Stable*.

$$t1_{begin} \leq t2_{begin} \leq t1_{end} - 2 \text{ basic time interval}$$

Examples of this pattern are shown in Figure 11.

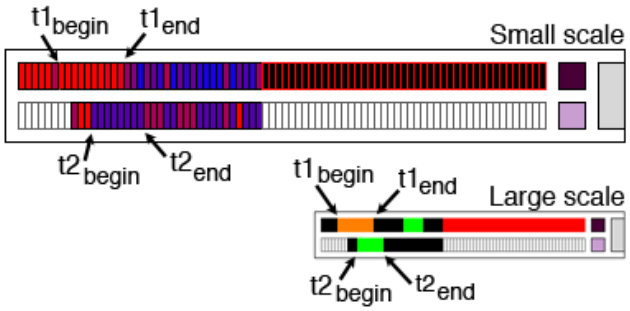

Figure 11. An example of the Bug Fixing pattern with the High Stable - Stable pair of phases.

Refactoring / Code Cleaning. A Refactoring or a Code Cleaning in the system consists in an increasing number of commits while the number of bugs remains fixed to a low value. In fact both Refactoring and Code Cleaning require a lot of effort in term of revisions, while they should not introduce new problems. Graphically this pattern is highlighted by the following pairs of phases:

- *High Stable - Unchanged Stable* and *Spike - Unchanged Stable*.

$$t1_{begin} > t2_{begin} \wedge t1_{end} < t2_{end}$$

Examples of this pattern are shown in Figure 12.

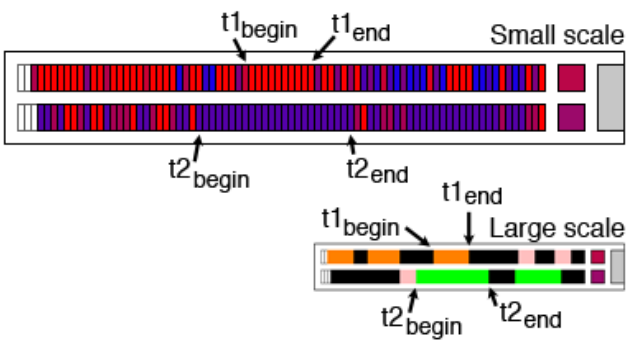

Figure 12. An example of the Refactoring / Code Cleaning pattern with the High Stable Unchanged Stable pair of phases.

This characterization allows an analyst to (i) argue about the entire system (in terms of the percentage of each category it includes) (ii) compare entities according to the categorization (iii) detect artifacts which need a further analysis.

5. Validation

To validate our approach we applied the following methodology on three case studies:

1. Build a view containing all the directories of the target system. Each directory is represented as a Discrete Time Figure if it includes at least one file, or it is represented as a white rectangle otherwise.

2. Apply a query engine (provided in our tool implementation) on the view to detect all the patterns defined in the previous Section. This allows us to characterize the system in terms of number of patterns it contains.

3. Analyze the view to understand how and where the various patterns are distributed within the system. We can also identify entities of particular interest which need further inspection (with our tool implementation the user can interactively navigate back and forth between different views).

Case studies. The three systems used as case studies are Apache, gcc, and Mozilla (see Table 2).

We analyzed only the .cpp and .c files. Hence the numbers depicted in Table 2 are referred to these parts of the systems.

System	Files	Revisions	Bugs	Bugs ref.
Apache	393	15524	377	717
gcc	18150	254785	3347	12936
Mozilla modules				
SeaMonkeyCore	4656	69391	4694	16889
RaptorDist	3446	50033	2753	10028
RaptorLayout	2925	99899	5797	22865
CalendarClient	1860	32468	2550	7001

Table 2. The dimensions of the case studies.

Pattern	Apache	gcc
Not empty directory	92	1145
Bug Persistent	0	0
High Prod. Persistent	0	5
Persistent	1	54
Dead Day-Fly	9	101
Day-Fly	12	465
Tot. Day-Fly	21	566
Intro. for Fixing	2	163
Stabilization	2	1
Spike - Spike phase	1	13
High Stable - High Stable	0	10
Spike - High Stable	0	0
High Stable - Spike	1	23
Tot. Add. of Feature	2	46
Spike - Stable	2	16
High Stable - Stable	9	52
Tot. Bug Fixing	11	68
High Stable - Unchanged Stable	0	8
Spike - Unchanged Stable	0	7
Tot. Refact. / Code Clean.	0	15

Table 3. The characterization of the Apache and gcc systems in terms of patterns found.

5.1. Characterization of systems

In Table 3 we summarize the number of detected evolutionary patterns in gcc and Apache.

Apache. The evolution of the Apache Web Server is characterized by:

1. A relatively large number of Day-Fly (21) and Bug Fixing (11).
2. A small number (between 0 and 2) of all the other patterns.

In this case we cannot formulate any conclusion on the system because the number of bugs we have are not enough to study the relationship between the evolution of the entities and the way they are affected by software bugs. This is the reason why the number of patterns detected is so small.

gcc. The most interesting data regarding the evolution of gcc is:

Patterns	Sea-Monkey-Core	Raptor-Dist	Raptor-Layout	Calendar-Client
Not empty directory	476	520	382	269
Bug Persistent	47	35	36	25
High Prod. Persistent	15	0	10	3
Persistent	59	60	46	29
Dead Day-Fly	68	98	68	37
Day-Fly	62	26	39	49
Tot. Day-Fly	130	124	107	86
Intro. for Fixing	77	50	57	50
Stabilization	16	25	14	12
Spike - Spike phase	10	8	6	2
High Stable - High Stable	56	15	41	12
Spike - High Stable	3	0	2	2
High Stable - Spike	46	11	30	13
Tot. Add. of Feature	115	34	79	29
Spike - Stable	7	10	7	5
High Stable - Stable	20	13	20	11
Tot. Bug Fixing	27	23	27	16
High Stable - Unchanged Stable	32	11	20	11
Spike - Unchanged Stable	21	13	10	5
Tot. Refact. / Code Clean.	53	24	30	16

Table 4. The characterization of the four biggest modules (with respect to the number of files included) of Mozilla in terms of patters found.

1. The number of Day-Fly is the 49% of the not empty directories (566 on 1145).
2. The number of Introduced for Fixing is the 14% of the not empty directories (163 on 1145).
3. The Stabilization pattern is only one over 1145 (0.0009%).
4. The percentage of all the other patterns (Persistent, Addition of Features, Bug Fixing and Refactoring / Code Cleaning) ranges from 0.01% to 0.06%.

We conclude that the gcc system was changed a lot and rapidly during its lifetime. This system is likely to contain a lot of spike solutions, while the number of entities survived to all its changes is small.

Mozilla. Table 4 shows the patterns detected in four modules of Mozilla. In this data facts of particular interest are:

1. The SeaMonkeyCore module is the one having the most intense development history. Even if it is not the biggest in terms of number of not empty directotories, it contains the maximum number of most of the patterns (all but Persistent and Stabilization).
2. The RaptorDist, which is the biggest module, is the most stable module since it has the maximum number of Persistent and Stabilization patterns.

3. The percentage of Persistent pattern varies from 11% (RaptorDist) to 12% (RaptorLayout) in all the modules; for the Stabilization pattern the values are 0.03% (SeaMonkeyCore) and 0.04% (CalendarClient) while for the Day-Fly pattern the percentage ranges from 23% (RaptorDist) to 31% (CalendarClient). We conclude that the Mozilla system includes a lot of spike solutions while the artifacts which survived for most of the system history are few.
4. The Mozilla system has gone through substantial changes during its lifetime, since the Addition of Features pattern are much more than the Bug Fixing and Refactoring / Code Cleaning patterns.
5. The SeaMonkeyCore module includes a lot of features (115 Addition of Features pattern) but most of them are likely to be spike solutions, taking into account the high number of Day-Fly it contains (130).

5.2. View analysis

For lack of space we cannot present neither a complete and in-depth analysis of one system, nor the views for all the case studies. Therefore we present a view for the CalendarClient module of Mozilla, shown in Figure 13, followed by a brief analysis.

The view in Figure 13 can be divided into the following parts:

- *Removed directories*, concentrated in the areas marked as 1, 3, 5, 6 and 9.
- *Day-Fly*, concentrated in the 2 and 7 areas.
- *Persistent*, concentrated in the 4 and 8 areas. This is the most interesting part of the module: In it we can find Addition of Features, Bug Fixing, Refactoring / Code Cleaning, Stabilization and Introduced for Fixing patterns.

6. BugCrawler: Implementation

We implemented the approach presented in this paper in a tool called BugCrawler, a major extension of the CodeCrawler tool [16, 17] with respect to two main components (see Figure 14), namely (1) the Release History Database and (2) a greatly extended visualization engine.

The Release History Database (RHDB). This component is completely new with respect to the original tool CodeCrawler. The RHDB, which is defined in [5] and extended in [4], contains the history of a system extracted from a CVS repository. To populate the database we use a set of perl and shell scripts. The scripts are responsible for the following tasks, run in completely autonomous batch mode:

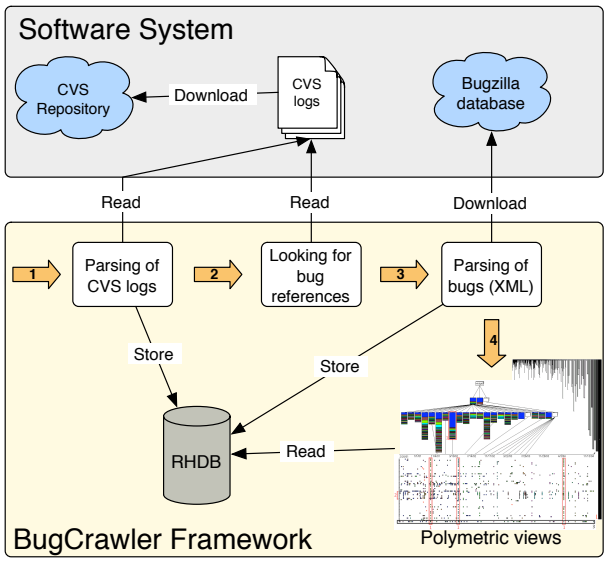

Figure 14. The BugCrawler framework structure and its interaction with a system.

1. Downloading and parsing the CVS log files. The processed information, concerning files, revisions of files, directories and modules, is stored in the database.
2. The comments inserted by the authors during the commits are analyzed looking for bug references by means of pattern matching techniques.
3. Each time a bug reference is found, the bug description is retrieved from the Bugzilla Database, parsed and stored in the RHDB.

7. Related Work

Visualization has long been adopted as a means to understand in a synthetic way large amounts of information, such as the one given by evolving software systems. All of the listed approaches do not take into account information given by bug reporting systems, this being one of the major contributions of this paper.

Ball and Eick [1] concentrated on the visualization of different aspects related to code-level such as code version history, difference between releases, static properties of code, code profiling and execution hot spots, and program slices. In [20], Tu and Godfrey tackle the issue of visualizing structural changes by studying the differences between releases of a system. In [15] Gall *et al.* presented an approach to use color and 3D to visualize the evolution history of large software systems. Jazayeri *et al.* analyzed the

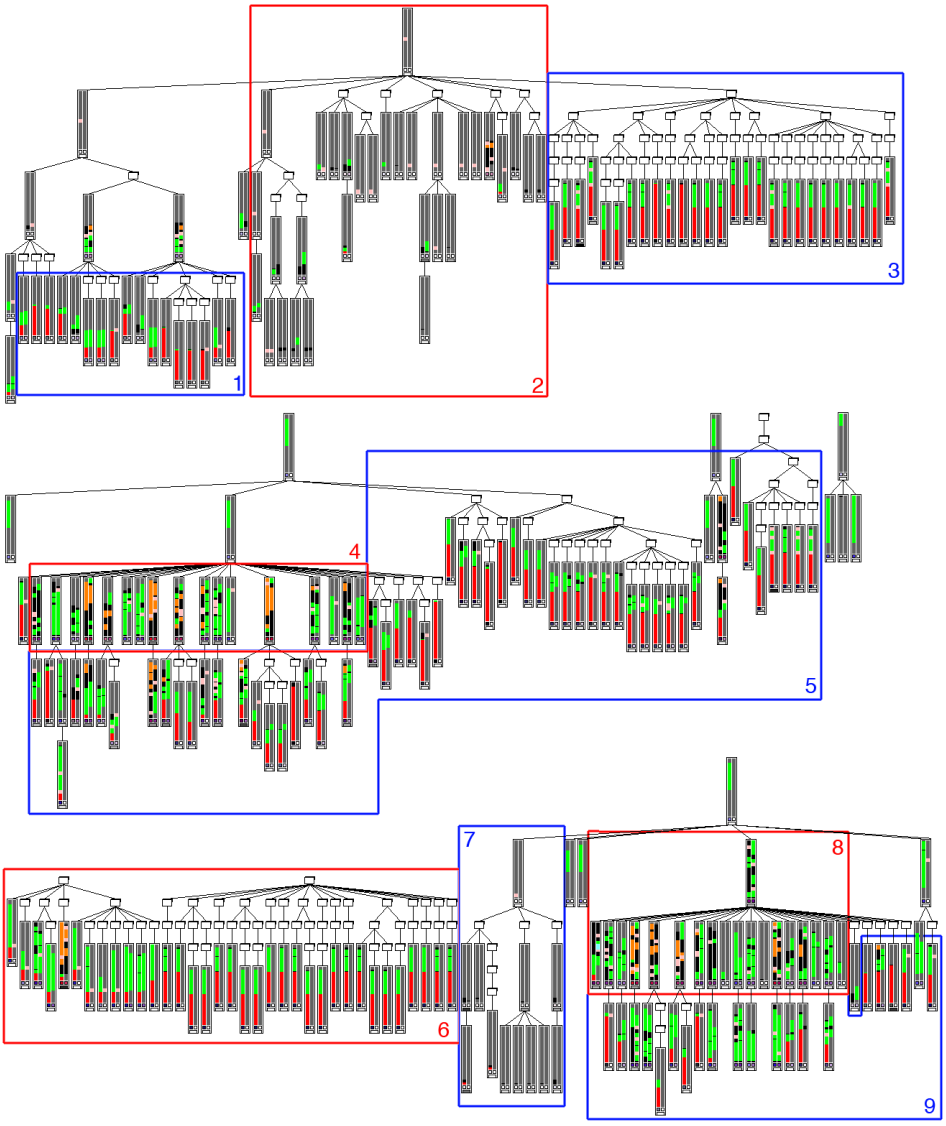

Figure 13. The Discrete Time Figure applied on the entire CalendarClient module of Mozilla.

stability of the architecture [14] by using colors to depict the changes. Wu *et al.* [22] used the spectograph metaphor to visualize how changes occur in software systems. Taylor and Munro [19] visualized CVS data with a technique called revision towers. Collberg *et al.* [3] focused on the visualization of the evolution of software using a temporal graph model. Rysselberghe and Demeyer [21] used a simple visualization based on information in version control systems (CVS) to provide an overview of the evolution of systems. In [18] Pinzger *et al.* proposed a visualization technique based on Kiviat diagrams to study the evolution of large software systems. The approach provides integrated condensed graphical views on source code and release history data of many releases. In [12] Girba *et al.* use visualization to characterize the evolution of class hierarchies. In [11] Girba *et al.* analyze how developers drive software evolution by visualizing code ownership based on information extracted from CVS.

8. Conclusion

In this paper we have presented a visual approach to study the relationship between the evolution of software artifacts and the way they are affected by problems. The approach is based on the application of *Discrete Time Figures* at any level of granularity. The scalability issue is tackled by abstracting the revisions and bugs trends by means of *Phases*.

The Discrete Time Figure indicates patterns of particular interest in the study of the evolution of software systems such as: *Persistent, Day-Fly, Introduced for Fixing, Stabilization, Addition of Features, Bug Fixing, Refactoring / Code Cleaning*.

These patterns include some of the possible relationships between the histories of revisions and bugs, providing them a precise and useful meaning. In our tool implementation is provided a small query engine which allows the user to automatically detect all the patterns presented in this paper.

This feature is quite valuable for a project manager (or an analyst) because it allows him to: (i) characterize the entire system in terms of patterns detected, (ii) identify areas of interest within the system (for example he/she can be interested in the entities which exhibit a Refactoring / Code Cleaning pattern) and (iii) compare different modules of a system in terms of patterns.

We validated our approach on Apache, gcc and Mozilla, thus proving the effectiveness of our approach.

Acknowledgments: We gratefully acknowledge the financial support of the projects "COSE - Controlling Software Evolution" (SNF Project No. 200021-107584/1), 'EvoSpaces - Multi- dimensional navigation spaces for software evolution" (Hasler Foundation Project No. MMI 1976), and "NOREX - Network of Reengineering Expertise" (SNF SCOPES Project No. IB7320-110997).

References

[1] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, 1996.

[2] A bug's life cycle. http://bugzilla.remotesensing.org/bug_status.html.

[3] 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.

[4] M. D'Ambros. Software archaeology - reconstructing the evolution of software systems. Master thesis, Politecnico di Milano, Apr. 2005.

[5] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, Sept. 2003.

[6] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[7] H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance 1997 (ICSM '97)*, pages 160–166, 1997.

[8] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, 2003.

[9] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM 2004 (International Conference on Software Maintenance)*, 2004.

[10] T. Gîrba, S. Ducasse, R. Marinescu, and D. Raţiu. Identifying entities that change together. In *Ninth IEEE Workshop on Empirical Studies of Software Maintenance*, 2004.

[11] 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)*. IEEE Computer Society Press, 2005.

[12] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance*, pages 2–11, 2005.

[13] M. W. Godfrey and Q. Tu. Evolution in Open Source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 131–142, San Jose, California, 2000.

[14] M. Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.

[15] M. Jazayeri, H. Gall, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM '99 Proceedings (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society, 1999.

[16] 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.

[17] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. Codecrawler — an information visualization tool for program comprehension. In *Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering)*, pages 672–673. ACM Press, 2005.

[18] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005, ACM Symposium on Software Visualization*, pages 67–76, St. Louis, Missouri, 2005.

[19] C. M. B. Taylor and M. Munro. Revision towers. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50. IEEE Computer Society, 2002.

[20] 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.

[21] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004. to appear.

[22] 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. IEEE Computer Society Press, Nov. 2004.