"A Bug's Life" Visualizing a Bug Database

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

Martin Pinzger
s.e.a.l. - software evolution and architecture lab
University of Zurich, Switzerland

Abstract

Visualization has long been accepted as a viable means to comprehend large amounts of information. Especially in the context of software evolution a well-designed visualization is crucial to be able to cope with the sheer data that needs to be analyzed. Many approaches have been investigated to visualize evolving systems, but most of them focus on structural data and are useful to answer questions about the structural evolution of a system.

In this paper we consider an often neglected type of information, namely the one provided by bug tracking systems, which store data about the problems that various people, from developers to end users, detected and reported. We first briefly introduce the context by reporting on the particularities of the present data, and then propose two visualizations to render bugs as first-level *entities.*

1 Introduction

Bug tracking systems play an important role in software development [9, 12]. They are used by developers, quality assurance people, testers, and end users to provide feedback on the system. This feedback can be reported as an incorrect or anomalous situation or as a request for enhancements.

Bug tracking systems are also used in software evolution research to perform retrospective system analysis [1, 4, 5, 10]. In this context the analysis goal is to understand which are the most problematic parts of the system. Bug tracking systems provide useful information for each bug, and include properties such as the description, the severity, the person assigned to fix the bug, etc.

Visualization of software evolution information is not new, and there are a number of techniques, which however mainly focus on the structural evolution of systems. Bugs, often considered as an unwanted "side dish" of the evolution phenomenon, in fact represent a valuable source of information that can lead to interesting insights about a system, that would be hard or impossible to obtain relying exclusively on structural information.

In this paper we consider bugs as *first-level* entities which can change and evolve over time. In particular we focus on the bug life cycle, *i.e.,* the history of a bug and the various states it traverses. Based on the information we recovered from a well-known bug tracking system, namely Bugzilla[1], we present two visualization techniques aimed at understanding bugs at two different levels of granularity:

1. *System Radiography.* This visualization renders bug information at the system level and provides indications about which parts of the system are affected by what kind of bugs at which point in time. It is a high-level indicator of the system health and serves as a basis for reverse engineering activities.

2. *Bug Watch.* This visualization provides information about a specific bug and is helpful to understand the various phases that it traversed. The view supports the characterization of bugs and the identification of the most critical ones.

In this paper we present and discuss both visualizations, which are complementary to established structural visualizations and other reverse engineering techniques.

Structure of the paper. In Section 2 we introduce the bug model, with a special focus on the bug life cycle. We explain the challenges and the constraints in the analysis of the Mozilla bug database in Section 3. Our two techniques for visualizing and analyzing a bug database are presented in Section 4 and Section 5. In Section 6 we discuss their benefits and shortcomings. In Section 7 we look at related work and conclude by summarizing our contributions in Section 8.

[1] See http://www.bugzilla.org for more information.

2 Modeling Bugs

Our reference model of a bug is an abstraction of the Bugzilla implementation. We chose Bugzilla because it is the most used bug tracking system in the open source community and because the models of other systems, such as Trac and Scarab[2], are simplifications of the Bugzilla model.

Bugzilla describes the following properties of a bug:

- *The problem.* Includes the unique identifier (*id*), the description (*short description*) of the problem and its location in the system. The location is identified by the pair *product::component*, where a product contains several components. Each bug has a list of comments (*long description*) describing possible solutions.

- *The criticality* of the bug, indicated by the fixing *priority* (from 1 to 5) and by its *severity*. The possible severities are, in order: Blocker (application unusable), critical, major, normal, minor, trivial (minor cosmetic issue), enhancement (request of enhancement).

- *The involved people*, which includes the *reporter* of the bug, the developer in charge to fix it (*assignedTo*), the quality assurance (*qa*) person who will test the solution and a list of people who are interested in being notified of the bug fixing progress (*CC*).

- *The condition* in which the bug was detected, such as *operating system* and *platform*.

- *The state* of the bug, composed of *status* and *resolution*. The status identifies at which stage of the life cycle the bug is. The possible values are: Unconfirmed, new, assigned, resolved, verified, closed and reopened. The resolution indicates whether and how the problem was solved, once the bug reaches the resolved status. Possible values here are: Fixed, invalid, won't fix, not yet, remind, duplicate, and works for me.

Our model takes time into account. Every field of a bug can be modified over time thus generating a *bug activity*. The activity records which field is changed, when, by whom and the pair of old and new values. Activities are important because they allow us to keep track of a bug's life cycle, *i.e.,* the sequence of statuses the bug went through.

Figure 1 shows all possible bug statuses and transitions, based on the Bugzilla model[3]. Each possible status is associated with a different color. Green colors represent statuses in which the bug is considered fixed (*resolved*, *verified*, *closed*), while red colors represent statuses in which the bug has to be fixed (*new*, *assigned*, *reopened*). We consider *reopened* as the most critical status, since a first attempt did not fix the bug. *Unconfirmed* is associated with cyan, since it is not known yet if the reported bug is real.

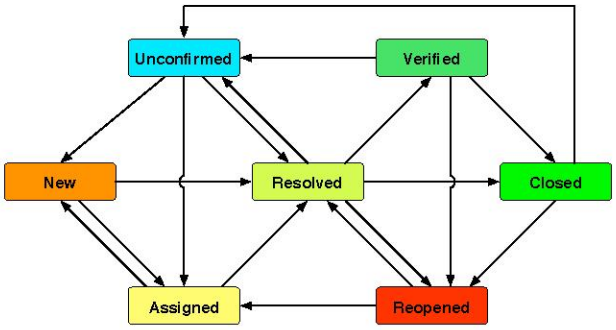

Figure 1. The Bug Status Transition Graph.

The typical life cycle of a bug is the following: It is reported (either *new* or *unconfirmed* according to the privileges of the reporter), it is assigned to a developer for fixing (*assigned*) and then he/she proposes a solution (*resolved*) or decides that a solution is not needed (for example when the resolution is set to duplicate or invalid or won't fix). When the bug is *resolved*, the quality assurance tests the proposed solution and sets the bug status to one of the following: Verified, closed, reopened, unconfirmed. The bug status transition graph does not have a final state, because a bug can always be reopened.

3 Visualizing the Mozilla Bugs

We visualize the bug data set of Mozilla[4] between September 1998 (when Bugzilla was introduced) and April 2003. Due to its sheer size and complexity this leads to a number of constraints on the visualization techniques. In particular, three properties of the Mozilla bug database have to be considered:

1. *Bug number*: The database contains 255'302 bug reports, therefore the visualizations have to scale.
2. *Bug liveliness*: On average a Mozilla bug lasts 523.2 days (time between the reporting and the last registered activity) with an average of 10.6 activities. The visualizations should not only display individual bugs, but also complementary information such as their activities and status histories.
3. *Bug importance*: Expressed with severity and priority, bugs have different impact and importance, and the visualization technique must help to convey this distinction.

[2] See http://trac.edgewall.org and http://scarab.tigris.org
[3] See http://www.bugzilla.org/docs/2.18/html/lifecycle.html
[4] See http://www.mozilla.org

4 The System Radiography View

The goal of the *System Radiography* view is to support the analysis of the bug database as a whole. We want to study how the open bugs (not fixed yet) are distributed in the system and over time.

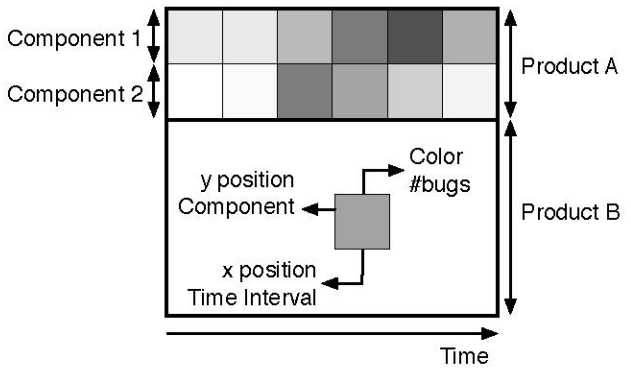

Figure 2. The principles of the System Radiography visualization.

Visualization Principles. Figure 2 shows the principles of the System Radiography visualization. The evolving system is displayed using a matrix-based representation. Each row of the matrix represents a system component, and each group of rows, *i.e.,* components, represents a system product. This hierarchical decomposition of the system in Product::Component is obtained by the bug database itself, since each bug affects a particular component of a particular product. The columns represent time from left to right. Each column corresponds to a parametrizable interval of time.

The y position of each cell represents a Product::Component pair, while the x position represents an interval of time. The color of the cells maps the number of bugs affecting the *y* component during the *x* time interval, where x and y are the position of the cell. We use a gray scale: The darker the color, the larger the number of bugs.

A bug, at any point in time (of its life), is characterized by a status. Therefore, it does not make sense to count the number of bugs during a time interval without considering their statuses. For this reason, the color of the cells represents the number of bugs with a given status (or set of statuses) during the considered time interval. This allows us to see the distribution of the bugs in the system and over time with respect to their status: For example "open" bugs, *i.e.,* bugs with *new*, *assigned* or *reopened* status, "solved" bugs (*resolved*, *verified* or *closed*), or only *new* or *reopened* bugs, *etc.* Other filters can be used, one per each field of the bug. For example, we can use the severity filter to count the *blocker* and *critical* bugs only.

Once the matrix is created, we apply a sorting algorithm to its rows before displaying it. The goal is, for each product, to sort its components according to the similarity of their histories, *i.e.,* the numbers of bugs for every time interval. Given a product p and two components c_1, c_2, corresponding to the matrix rows r_1, r_2 of size n (number of columns), the similarity between the components is defined as the Euclidian distance of the points r_1, r_2 in a n-dimensional space, as defined by Equation 1.

$$d(r_1, r_2) = \sqrt{(r_1(1) - r_2(1))^2 + \cdots + (r_1(n) - r_2(n))^2} \quad (1)$$

The value of $r_i(j)$ is the number of bugs with a given status (and after the filtering) of the ith component within the jth time interval.

After sorting, we obtain a matrix in which the products are alphabetically sorted and where the components within each product are sorted according to the defined similarity. This sorting allows us to detect groups of components which were affected by a large number of bugs in the same time window.

Figure 3 shows our tool visualizing a System Radiography of Mozilla, from June 1999 to April 2003, where open bugs only are considered. The time interval used is three days, meaning that each column of the matrix represents three days of time. The main window of the tool is divided in a visualization part on the left, containing the actual System Radiography, and an information part on the right. This part displays the information relative to the matrix cell under the pointer: The time interval, the product::component pair and the list of bugs (id and short description) affecting that component during that time interval. Only the bugs with the considered statuses and severities are listed.

Visualization Interpretation. Our goal with this first visualization is to understand where and when the open bugs are concentrated. In Figure 3 we see that *Browser* is the biggest product, in terms of number of components, and the most affected by open bugs. It has a large number of gray rows, *i.e.,* components affected by many bugs. We identified and annotated five system areas with the highest density of open bugs, described in Table 1.

These areas contain system components which were affected by a large number of open bugs for a long period of time. The average number of bugs per basic time interval (three days) varies from 145 to 408, while the total amount of different bugs varies from 874 to 24407. The shortest time window is 22 months, the longest 46 months. Such amounts and densities of bugs with such a persistency over time are bad symptoms, which indicate that the components

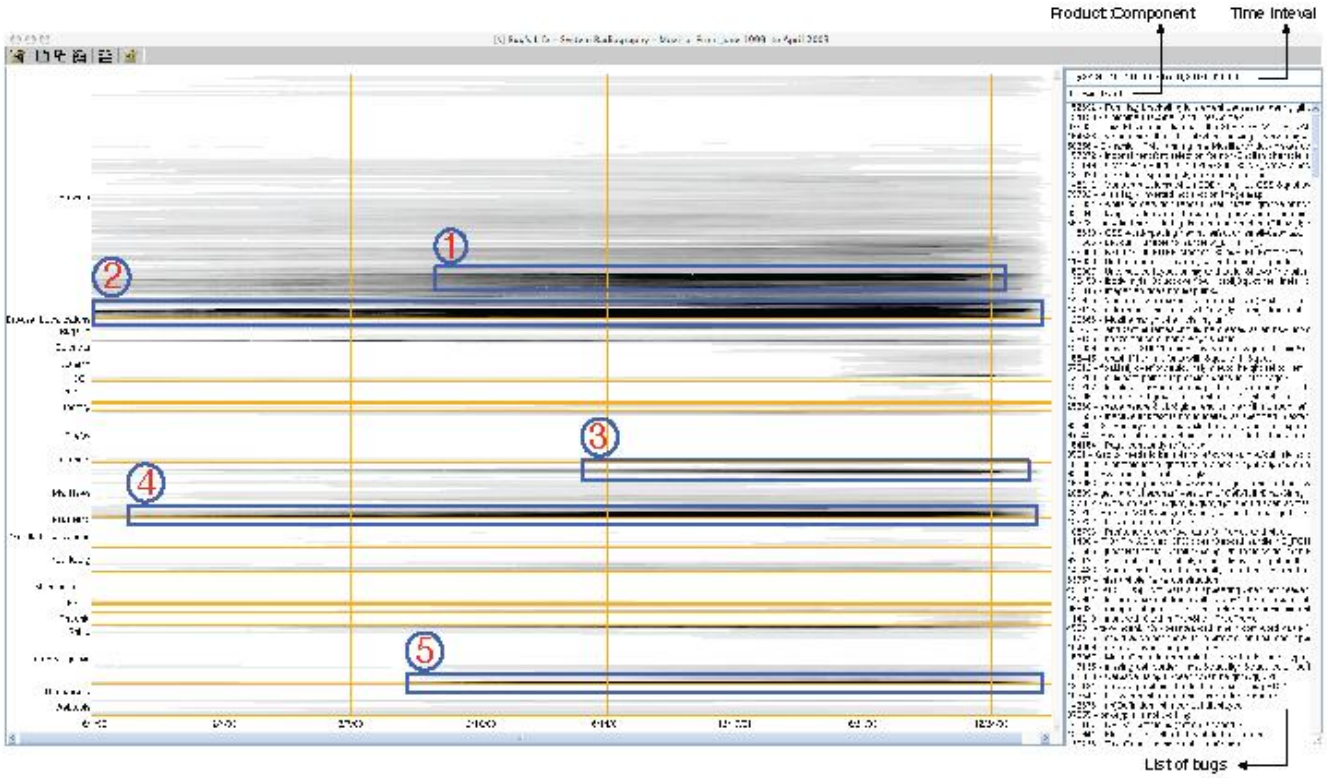

Figure 3. The System Radiography of Mozilla from June 1999 to April 2003. Only bugs with new, assigned or reopened statuses are considered.

Label	Product	Components	Time interval	Avg no. of bugs / 3 days	# of different bugs
1	Browser	Bookmarks, Layout: Form Controls, Layout: Tables, Plug-ins, XP-Apps: GUI Features, Event Handling	May '01-Jan '03	215	5570
2	Browser	Browser-general, Layout, XP Toolkit/Widget, Editor Core, Networking, XP Apps, OJI	Jun '99-Apr '03	291	24407
3	MailNews	Networking IMAP, Account Manager	May '01-Mar '03	145	874
4	MailNews	Address Book, Composition, Mail Back End, Mail Window Front End	Aug '99-Apr '03	408	9421
5	Tech Evangelism	Europe West, US General	Oct '00-Apr '03	250	1871

Table 1. The properties of the five highlighted areas in Figure 3.

are badly designed or implemented and thus they should be reengineered to decrease the number of introduced bugs.

The same bug can be present in different cells of the same row. For example, if a bug was in the assigned status for 9 days, and the time interval is three days, then the bug is present in three cells. On the other hand, since the product and component fields of a bug can have just one value, the same bug cannot be present in different cells of the same column.

Our second goal of the analysis in the large is to understand where and when the open and most severe bugs are located. To do so, we generate a second System Radiography by selecting the bugs with the *blocker* or *critical* severity. In the obtained visualization, not shown for lack of space, we detect an area similar to the ones highlighted in Figure 3. That area contains the components Browser-General and OJI from March 2002 to April 2003. The average number of bugs per three days is 54 with a total of 256 different bugs. The two components are present also in area 2 in Figure 3. They are the most critical components in terms of being affected by bugs, and thus should be analyzed in detail.

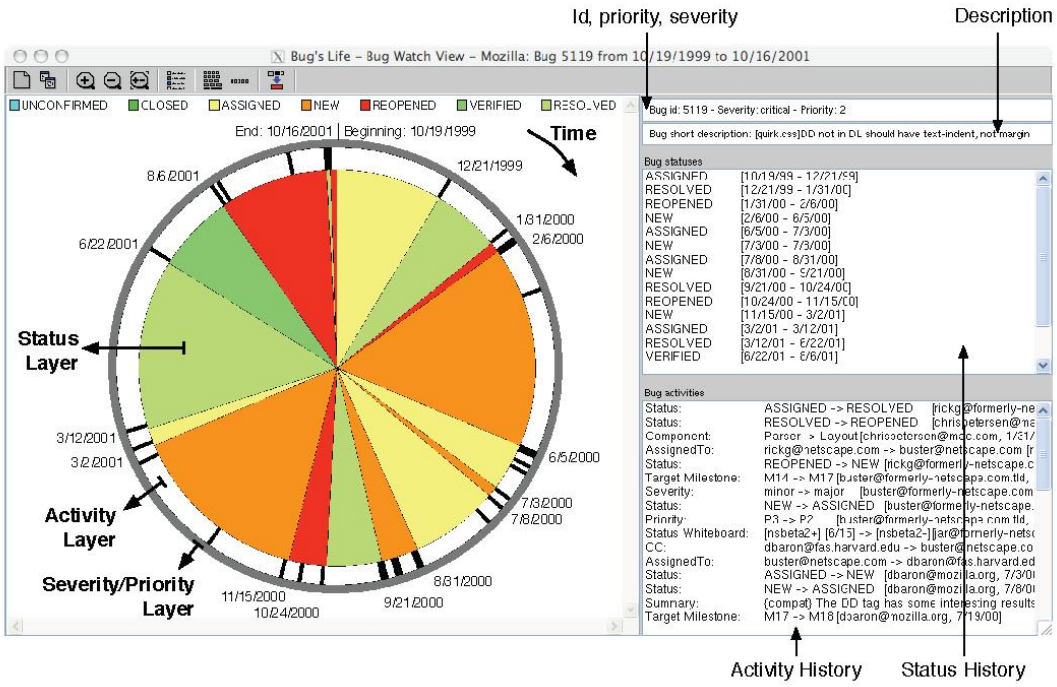

Figure 4. A Bug Watch Figure visualizing Bug 5119 of Mozilla between Oct 19 1999 and Oct 16 2001.

5 The Bug Watch View

With the System Radiography view we obtained a big picture of the system from the bug perspective and we detected the critical components. The purpose of the Bug Watch view is to ease the analysis of single bugs. Our goals are to characterize the bugs affecting a given set of components during a given time interval and to detect the most critical bugs. Our underlying assumption is that the criticality of a bug does not depend only on its severity and priority, but also on its life cycle. For example, a bug reopened several times indicates a deeper problem than expected.

For this type of analysis we need a visualization which fulfills the following requirements:

- *Considering time.* The visualization has to be time-based in order to show the life cycle history of a bug.
- *Considering severity and priority.* These bug properties, in the context of a bug's life cycle, allow us to detect critical bugs.

Visualization Principles. Figure 4 shows a *Bug Watch* visualization of a Mozilla bug. This visualization technique uses a watch metaphor to represent time: The initial timestamp is mapped to 00:00 on the watch, the final timestamp is mapped to 11:59. The figure is composed of three layers:

1. The *Status layer* represents the bug life cycle. Each status the bug passed through is visualized as a sector, using the same color schema as in Figure 1. The position and the size of each sector map the initial timestamp and the duration of the corresponding status.
2. The *Activity layer* visualizes modifications of any bug property. Each activity is represented as a black bar and positioned according to when the modification happened. An activity is an event and therefore its visual representation has a fixed size. Wider bars denote several activities in the same time interval.
3. The *Severity/Priority layer* depicts information about the severity and the priority of a bug. Dark colors denote high priority and *blocker* or *critical* severity, while bright colors denote low priority and *minor*, *trivial* or *enhancement* severity.

We can map different bug metrics on the radius of a Bug Watch. We usually choose the number of statuses in the considered time interval which also corresponds to the number of sectors. This, besides facilitating the detection of bugs with an intense life cycle, also has the benefit of making the statuses more readable: Bugs with many statuses are represented with larger figures. Figure 5 shows an example of this: Statuses are always readable, as long as they do not last for a very short time.

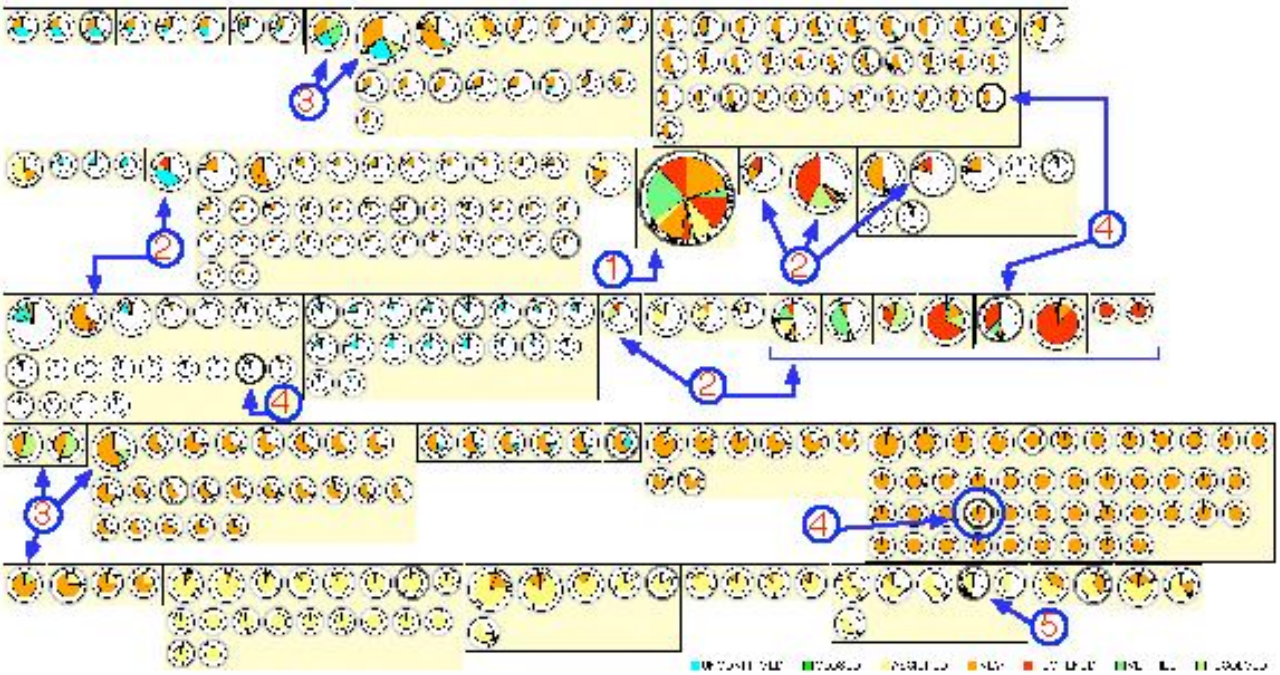

Figure 5. Bug Watches applied to the bugs affecting the Browser::Networking component of Mozilla. The reference time interval is from November 2002 to April 2003.

One problem of the Bug Watch visualization is that it is not possible to distinguish between different activities, since they are all visualized in the same way. We opted against the use of different colors for different activities because this would make the figure too complex to interpret. A second problem arises when there are statuses which last for short periods of time. They are represented as narrow sectors, which are difficult to distinguish, especially in zoomed out views as for example in Figure 5. To overcome these issues, we provided our tool with panels showing complementary information about the bug in focus (see Figure 4).

Considering the high number of bugs, the question is which bugs to visualize using the *Bug Watch*. The starting point is usually the previously presented *System Radiography* view: We select/draw a rectangle in the System Radiography which will be converted in a set of bugs and in a time interval. To build the set of bugs, all the matrix cells covered by the rectangle are considered, and all the corresponding bugs are added. Depending on its height, the rectangle can cover one or several system components. The time interval is created by considering the first and the last matrix columns covered by the rectangle.

The selected bugs are then visualized by means of Bug Watches, with the same reference time interval. Figure 5 shows a Bug Watch view (without the information panels for space reasons) applied to the rectangle annotated with 2 in Figure 3. The area covers the Networking component from November 2002 to April 2003. In the visualization shown in Figure 5, we grouped the bug watches using the following criterion: Bugs having similar life cycles (in the considered time interval) are grouped and placed in the same box. Bugs with a diverse life cycle form single-element boxes. The boxes are placed in a way that similar boxes, with respect to the similarity of the contained bugs, stay close to each other. This grouping technique facilitates the characterization of bugs, according to their life cycles, and the identification of "exceptional bugs".

Visualization Interpretation. Figure 5 shows the open bugs affecting the Browser::Networking component of Mozilla from November 2002 (mapped to 00:00 on the watch) to April 2003 (mapped to 23:59). We observe 5 interesting facts, annotated in the figure:

1. It is a crucial bug in the component, since it has blocker severity and maximum priority (dark severity/priority layer), and it was reopened four times. The activity layer of the figure shows that the history of the bug is rich of activities. The details of these activities, given in the information panels, tell us that the developer in charge of fixing the bug changed six times and that many people are interested in the bug, since many e-

mail addresses were added to the *CC*. The bug id is 31174 and the problem is related to SSL channels and proxies, as stated in the short description.

2. All these bugs are hard to fix, since they were reopened at least one time (and at most three). They have various levels of severity and priority, but only one of them (marked also as 4) has a critical severity.

3. These bugs have an unusual life cycle: They pass from a resolved or verified status to an unconfirmed or new status, without being reopened. By reading the activity details, we discovered that this behavior is associated with a change of person in charge (*assignedTo*) and often with a change of quality assurance person (*qa*).

4. All these bugs have the maximum level of priority and severity (critical or blocker).

5. This bug has a life cycle composed of one status only, but its history is full of activities. All these activities are an addition of a *CC*, meaning that increasingly more people are interested in that bug.

All the other bugs have a "normal" life cycle for not-yet-fixed bugs, following for example the transitions unconfirmed → new or new → assigned. None of these bugs has both maximum priority and blocker or critical severity.

6 Discussion

The proposed visualizations support the analysis of a bug database at two different but complementary levels of granularity. This has two main benefits: (1) The technique scales up to the size of Mozilla, *i.e.,* 250'000 bugs and 2'700'000 activities, and (2) it is possible to visualize and inspect any individual bug together with its history.

The interaction capabilities the tool offers, allows the user to "jump" from the large scale System Radiography to a detailed Bug Watch view, which visualizes a selected part of the system or even only one bug. The tool allows also to customize the views by applying filters to bug properties such as priority and severity.

Another advantage of the approach is that both visualizations include the time dimension. In particular the Bug Watch view has the time embedded in each figure. This allows any type of layout and grouping, without loosing or modifying time-based properties of bugs (*e.g.,* life cycle and activities). The last benefit comes from the sorting and grouping techniques used in both views, which ease the detection of critical areas of the system and the characterization of bugs.

Concerning shortcomings, the data set we used to experiment with our visualizations is limited, since it does not contain recent data (up to April 2003). Recent versions of Bugzilla are still based on the bug and life cycle models used in this paper and a validation with recent data is required. We propose a general technique based on a single data set. To prove that the approach is general we need to apply it to other case studies.

7 Related Work

The approach presented in this paper is a follow up of our previous work in which we showed how to extract and integrate versioning data from CVS and bug reporting data from Bugzilla into a Release History Database (RHDB) [6]. The data is used to analyze the evolution of software systems such as presented by D'Ambros *et al.* in [4] and by Pinzger *et al.* in [11]. These approaches focused on providing visualizations for CVS, Bugzilla, and source code data whereas in this paper we take into account the bug report details and in particular their life cycles.

Fischer and Gall used CVS and Bugzilla data to depict the evolution of features [5]. They use graph-based visualizations to project bug report dependence onto feature-connected files and the project directory structure. Dependencies occur through source files that were modified to fix a particular bug. The two different views allow the spectator to uncover hidden dependencies between features and shortcomings in the design.

Some open source projects, *e.g.,* KDE[5] customized Bugzilla to create bug statistics and charts. These statistics basically consists in database queries to detect recently reported bugs, the most voted ones, the products with most bugs *etc.* The main difference with our visualization is that we provide a big picture of the system with the entire history and a detailed view, both interactive. This allows the user to detect interesting areas and to inspect the details on-the-fly.

Similar to our approach Halverson *et al.* presented a number of visualizations [7] to support the coordination of work in software development, namely the Work Item History and the Social Health Overview. The Work Item History shows status changes of bug reports and makes problematic patterns such as resolve/reopened visible. The Social Health Overview provides an interactive overview of bug reports with drill down capabilities. Whereas they focus on analyzing team coordination we focus on analyzing the evolution of software systems.

Mozilla has been addressed in a number of case studies such as by Mockus *et al.* [9]. They used data from CVS and Bugzilla focusing on the overall team and development process such as code contribution, problem reporting, code ownership, and code quality including defect density in final programs, and problem resolution capacity as well.

The knowledge stored in bug repositories has been used by researchers to aid in bug triaging. For instance, in [2]

[5] See http://bugs.kde.org

Anvik *et al.* presented an approach to semi-automatically assign a developer to a newly received bug report. Using the data who solved what kinds of bugs from Bugzilla repositories they apply machine learning algorithms to recommend to a triager a set of developers who may be appropriate for resolving the bug. A similar approach has been presented by Čubranić *et al.* in [3]. Instead of recommending developers, our approach provides visualizations to analyze and understand bug life cycles.

A number of approaches use bug report data to train mining algorithms and validate their results. For instance, Śliwerski *et al.* [13] and later on Kim *et al.* [8] investigated changes that induce bug fixes. They start with a bug report indicating a fixed problem, extract the associated changes in the source code, and analyze the earlier changes that might have induced the reported bug. In contrast to these approaches we focus on a detailed analysis of bug reports, considering them as first class entities.

8 Conclusion

In this paper we have proposed an approach to support the analysis of a bug database, by means of two visualizations: The *System Radiography* and the *Bug Watch* views. The System Radiography is aimed at studying the bug database in the large: The visualization is helpful to understand how the bugs are distributed in the system products and components and over time. It also highlights the critical parts of the system, *e.g.,* the components affected by most of the bugs. The Bug Watch view supports the analysis of the bugs affecting a limited part of the system, *e.g.,* one or few components. The visualization facilitates the characterization of bugs and the identification of critical bugs.

The main contributions of the paper are:

1. Introducing the concept of a Bug's Life, *i.e.,* bugs are considered as evolving entities which change over time. Studying the history and, in particular, the life cycle of bugs permits an accurate characterization.

2. Using the watch metaphor, and introducing the corresponding Bug Watch figure, to visualize and analyze the life cycle of bugs.

3. Introducing a new criterion for bug criticality. Besides the severity and priority we also consider the life cycle. The underlying assumption is that bugs reopened several times are more critical.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "COSE - Controlling Software Evolution" (SNF Project No. 200021-107584/1).

References

[1] G. Antoniol, M. Di Penta, H. Gall, and M. Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. In *Proceedings Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 83–94, Amsterdam, 2004. Elsevier.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM Press.

[3] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *SEKE*, pages 92–97, 2004.

[4] 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 Computer Society Press, 2006.

[5] M. Fischer and H. Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):385–403, 2004.

[6] 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.

[7] C. A. Halverson, J. B. Ellis, C. Danis, and W. A. Kellogg. Designing task visualizations to support the coordination of work in software development. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work (CSCW 2006)*, pages 39–48, New York, NY, USA, 2006. ACM Press.

[8] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[9] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[10] M. Pinzger, H. Gall, and M. Fischer. Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science*, 127(3):183–196, 2005.

[11] 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.

[12] C. R. Reis and R. P. de Mattos Fortes. An overview of the software engineering process and tools in the mozilla project. In *The Open Source Software Development Workshop*, pages 155–175, 2002.

[13] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of International Workshop on Mining Software Repositorie – MSR'05*, Saint Lous, Missouri, USA, 2005. ACM Press.