BugMap: A Topographic Map of Bugs

Jiangtao Gong[1] and Hongyu Zhang[1,2]
[1]Tsinghua University
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
gongjiangtao2@gmail.com, hongyu@tsinghua.edu.cn

ABSTRACT

A large and complex software system could contain a large number of bugs. It is desirable for developers to understand how these bugs are distributed across the system, so they could have a better overview of software quality. In this paper, we describe BugMap, a tool we developed for visualizing large-scale bug location information. Taken source code and bug data as the input, BugMap can display bug localizations on a topographic map. By examining the topographic map, developers can understand how the components and files are affected by bugs. We apply this tool to visualize the distribution of Eclipse bugs across components/files. The results show that our tool is effective for understanding the overall quality status of a large-scale system and for identifying the problematic areas of the system.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance and Enhancement**]: Restructuring, Reengineering

General Terms

Experimentation, Human Factors, Measurement

Keywords

Software visualization, bug, topographic map, bug location.

1. INTRODUCTION

Although a range of measures have been taken to assure software quality, in reality released software systems still contain bugs. For a large-scale, widely-used software system, the project team could receive a large number of bug reports over time. For example, 49,422 bugs were reported for the Eclipse system in 2010, on average 135 bugs per day. These bugs are typically stored and maintained by a bug tracking system such as BugZilla. It is interesting for developers to understand how the bugs are distributed across the system, so they could identify the problematic areas of the system.

Visualization is an important way to comprehend large amounts of information. It has been applied to many areas including software engineering. Many software visualization techniques have been developed to help developers comprehend software-related information, such as bug database [2], code metrics [16], evolution history [9], vocabulary [10], dependencies [13], and clones [8].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2494582

In this paper, we propose to visualize bug location information to help developers understand the overall quality status of a large-scale software system. Traditional tabular format is not suitable for visualizing a large amount of information. We propose a novel tool called BugMap, which visualizes bug location information on a topographic map. Using BugMap, component/files are displayed on a two-dimensional map, where the distance between two components/files indicates the dependency between them. The bug numbers are visualized using contour lines. The "height" of a contour line indicates the number of bugs in a component/file. Different colors are also used to denote different heights of contours.

A screenshot of BugMap for Eclipse 2.0 is shown in Figure 1, which displays the distribution of Eclipse bugs across all the Eclipse components. It enables the users to understand the overall quality status of the system. In this paper, we describe the design of the BugMap tool and the experiment on Eclipse 2.0.

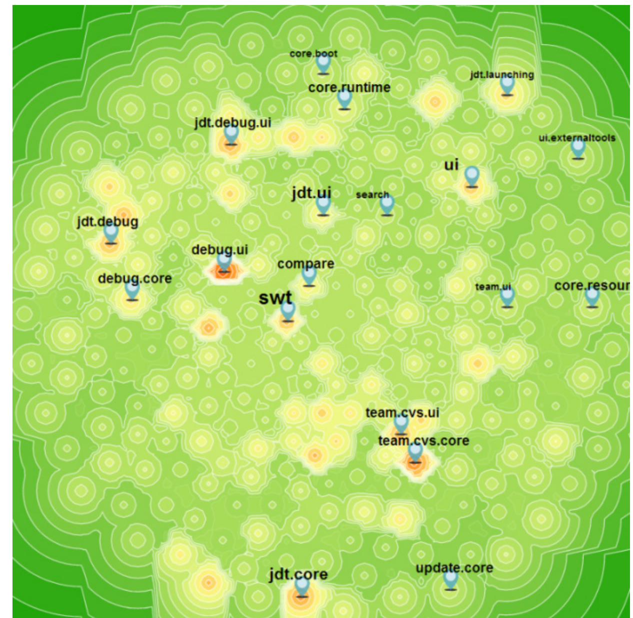

Figure 1. The bug map of Eclipse 2.0

2. BUGMAP

2.1 Topographic Map

Topographic map is a type of map that uses contour lines to visualize geological information. A contour line[1] is a curve along which all points have equal height. Traditionally, topographic maps are applied to areas such as geography and civil engineering. Recently, it has been applied to visualize data obtained from information retrieval and knowledge discovery [5, 7].

[1] http://en.wikipedia.org/wiki/Contour_line

On a topographic map, each contour line indicates that all the points on this line have the same height (z position). On a topographic map for displaying geological information, the orange color often denotes mountains, while the green color represents plains. Therefore, if the color of an area is darker (more orange), it is a higher terrain. If the color is lighter (more green), it is a lower-lying area. For example, Figure 2 gives a sample topographic map. There are 4 points (A, B, C, D) on 3 contour lines on the map. The points B and C are on the same contour line, therefore these two points have the same height (z position). Based on the contour lines, we know that $Z_A<Z_B=Z_C<Z_D$ (Z_i represents the z position of the corresponding point i).

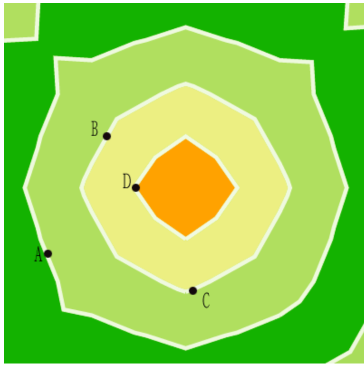

Figure 2. A sample topographic map

2.2 Visualization of Bug Locations on a Topographic Map

We visualize the bug localization information using a topographic map as follows.

2.2.1 The Layout of Topographic Map

In our topographic map for bugs, topography is generated from an undirected graph $G = (V, E)$, where each vertex v in V is a component or a file and each edge e_{ij} in E is the dependency between two nodes v_i and v_j.

We measure the degree of dependency as the number of function calls between two components/files. The stronger the dependency, the closer the two components/files on a topographic map.

To depict the dependency relationship, we calculate the (x, y) position for each node in the graph G. There are many graph layout algorithms. However the computation cost of these algorithms is often high for a large-scale graph. In our work, we choose the Force-Directed layout algorithm [3], which is a graph drawing algorithm based on physical simulations [2]. The Force-Directed algorithm assigns forces among the set of E and the set of V of graph G, and uses the repulsive forces between any v_i and v_j and attractive forces between adjacent v_i and v_j. The force models can be described using Hooke's law and Coulomb's law as follows:

$$\text{Hooke's law: } F_a(e_{ij}) = {d_{ij}}^2/k$$

$$\text{Coulomb's law: } F_r(e_{ij}) = -\,k^2/d_{ij}$$

In the above formula, d_{ij} represents the distance between v_i and v_j, and k denotes a constant of the optimal length of E.

In our work, we use dependency as attractive force and give each pair of nodes a relatively large repulsive force to ensure that the nodes can be dispersed in the entire plane. We calculate the relative displacement of each node based on the attractive force and repulsive force until its value is no longer changed.

2.2.2 Visualizing the Number of Bugs

In our topographic map, we use the contour lines to depict the number of bugs in each component/file. The height of a contour line (z position) indicates the number of bugs. Like ordinary topographic map, our bug map also uses different colors to indicate the height of each point. In Figure 1, a point with a darker color (more orange) indicates that the component has more bugs, and a point with a lighter color (more green) indicates that the component has fewer bugs. For example, the highest point in this map is the component "org.eclipse.debug.ui", which has 1155 bugs. The component "org.eclipse.ui.externaltools" has 71 bugs therefore it has a lighter color. Because the distribution of bugs could be highly skewed, we transform the bug numbers using a log function to fit the height scale.

After the z position of a component/file is computed, we need to calculate the z position of other points. This can be achieved by an interpolation algorithm for image processing. In our work, we adopt an inverse distance weighting interpolation algorithm [6]:

$$z_0 = \sum_{i=1}^{n} \frac{1}{(d_i)^2} z_i \left[\sum_{i=1}^{n} \frac{1}{(d_i)^2} \right]^{-1}$$

In this formula, z_0 represents the estimated value, z_i is the z position of the ith $(i = 1,2,3,\ldots,n)$ points, d_i represents the distance between z_0 and z_i.

2.2.3 Generating a Topography Image

In our work, we use the CONREC [1] algorithm to generate contour lines. Contouring aids in visualizing three dimensional surfaces on a two dimensional medium. In the CONREC algorithm, line segments that make up a given contour are generated from a two-dimensional array, where the z positions are stored in array elements and the horizontal (x) and vertical (y) coordinates are array indexers. We use the CONREC algorithm to generate output as vector shapes [4] and then plot the topography image.

2.3 Tool Development

We develop BugMap, a tool for visualizing bug location information on a topographic map. Figure 3 shows the overall structure of BugMap. Given a software project, BugMap acquires the dependency data, metric data (such as lines of code), and bug data from project repositories. In our implementation, we extract the dependency data using the DependencyFinder [3] tool, the source code metric data using Eclipse Metrics plugin, and the bug data from the Bugzilla bug-tracking system. We store all the data into a MySQL database for further analysis and visualization.

BugMap calculates the map layout using the Force-Directed algorithm described in the previous section. After all the coordinates of components/files are computed, BugMap uses the inverse distance weighting interpolation algorithm to calculate other points' z position in a 900*900 data grid. Finally, BugMap uses this data grid to generate a topographic map through the CONREC algorithm.

[2] http://en.wikipedia.org/wiki/Force-directed_graph_drawing

[3] http://depfind.sourceforge.net/

Figure 4. The file-level view of bug map for Eclipse 2.0

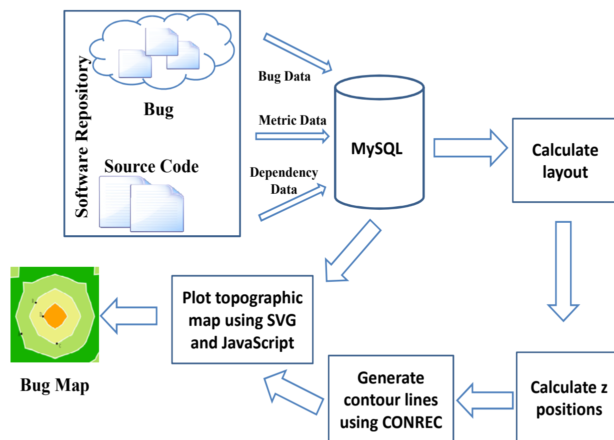

Figure 3. The structure of BugMap

BugMap utilizes the HTML5 inline SVG (Scalable Vector Graphics)[4] as its display platform. It plots the contour lines using a JavaScript implementation of the CONREC contouring algorithm[5]. BugMap also supports the zooming and panning functions. By clicking a component label in the bug map, the detailed information about the distribution of bugs across files in that component can be displayed.

3. AN EXPERIMENT ON ECLIPSE

We use Eclipse version 2.0 as an example to illustrate the usage of our tool. The Eclipse bug information is fetched form Bugzilla. We collected 8,000 bug reports in 45 Eclipse components and 6,730 source code files. We extracted 162,387 dependency relationships among the components/files using the DependencyFinder tool.

Having collected the data, we visualize it using BugMap. Figure 1 shows an overview of the topographic map for Eclipse 2.0. Some of the problematic components are highlighted on the map. It can be found that that the largest component "org.eclipse.swt" (its label has the largest font) is not the component having the most of the bugs (its color is not the darkest). The color of component "org.eclipse.debug.ui" is the darkest among all components, meaning that it is the most problematic component. By simply examining the bug map, the developers can quickly obtain an overview of the quality status of the entire system.

[4] http://www.w3.org/TR/SVGTiny12/

[5] http://paulbourke.net/papers/conrec/

We can also select a component on BugMap and zoom into it. Figure 4 shows an enlarged bug map with a file-level view. We can see that the four Java source files in the "org.eclipse.jdt.debug.ui" component: *JDIModelPresentation*, *JavaDebugOptionsManager*, *JDIDebugUIPlugin,* and *JavaDebugPreferencePage* have strong inter-dependencies and also have high numbers of bugs. This indicates a potential problematic area of source code that should be reviewed. Our approach can help developers quickly understand the quality of files in a component and identify areas for improvement.

4. RELATED WORK

Software visualization is "the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software" [14]. Over the years, many software visualization techniques have been proposed. For example, D'Ambros et al. [2] proposed a system radiography technique to display bug information at the system level. It is a high level indicator of the system health. They also proposed a "Bug Watch" technique to visualize information about a specific bug. Kuhn et al. [10] proposed a "software map", in which in the position of a software artifact reflects its vocabulary, and distance corresponds to similarity of vocabularies. Wettel et al. [15, 16] proposed CodeCity, which uses a city metaphor to describe large-scale software systems (e.g., classes as buildings and packages as districts). Design problems (disharmonies) can be identified by examining a code city map [16]. Lanza and Ducasse [12] presented the concept of polymetric view, which can help understand the structure and detect problems of a software system in a reverse engineering process. Minelli and Lanza [11] presented SAMOA, a web-based visualization tool to comprehend mobile apps, leveraging three factors: source code, usage of third-party APIs, and historical data. Their results reveal that apps differ significantly from traditional software systems in a number of ways. In our work, we apply the topography technique to visualize bug location information.

5. DISCUSSIONS

In this section, we answer the questions specified in the call for paper[6]:

- **What is the new idea?** In this paper, we propose to use the topographic map to visualize bug location information.
- **Why is it new?** Many software visualization techniques have been proposed to visualize software data including bug data. To our best knowledge, it is novel to visualize bug location data on a topographic map.
- **What is the single most related paper by the same author(s)? By others?** The authors have not yet published in the area of software visualization. The single most related paper by others is the work on CodeCity [16], which visualizes a software system based on a city metaphor.
- **What feedback do the authors expect from the forum?** We would like to exchange ideas with other researchers and explore the applicability of the BugMap in understanding software quality.

6. CONCLUSION

A large and complex software system could contain a large number of bugs. In this paper, we propose a tool called BugMap, which can help developers understand how the bugs are distributed across the system by visualizing the bug locations on a topographic map. By examining the bug map, developers could have a better overview of the quality status of the entire system and identify problematic areas.

In future, we will apply BugMap to a variety of open source and industrial projects, and evaluate its usefulness in practice by obtaining feedback from actual developers.

7. ACKNOWLEDGMENTS

This research is supported by the NSFC projects 61073006 and 61272089.

8. REFERENCES

[1] P. Bourke, Contouring Algorithm. *Byte*, Volume 12 Issue 6, pp 143-, July 1987.

[2] M. D'Ambros, M. Lanza, M. Pinzger: "A Bug's Life" Visualizing a Bug Database. *Proc. VISSOFT 2007*: 113-120

[3] T. Fruchterman and E. Reingold, Graph drawing by force-directed placement. *Software - Pract. Exp.,* 21, 1129–1164, 1991.

[4] J. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer Graphics: Principles and Practice in C* (2nd ed.), Addison-Wesley, 1995

[5] K. Fujimura, S. Fujimura, T. Matsubayashi, T. Yamada, and H. Okuda. 2008. Topigraphy: visualization for large-scale tag clouds. *Proc. WWW '08*, ACM, 1087-1088.

[6] P. Longley, *Geographic Information Systems and Science*, John Wiley & Sons, 2005.

[7] T. Matsubayashi and K. Ishiguro, Mobile topigraphy: large-scale tag cloud visualization for mobiles. *Proc. WWW '11*, ACM, 89-90.

[8] F. Hermans, B. Sedee, M. Pinzger, A. van Deursen: Data clone detection and visualization in spreadsheets. *Proc. ICSE 2013*: 292-301

[9] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance*, 15(2):87–109, 2003.

[10] A. Kuhn, P. Loretan, and O. Nierstrasz, Consistent Layout for Thematic Software Maps, *Proc. WCRE'08*, pp. 209—218, October 2008.

[11] R. Minelli, M. Lanza, Software Analytics for Mobile Applications—Insights and Lessons Learned, *Proc. CSMR 2013*, pp. 144-153.

[12] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[13] M. Pinzger, K. Grafenhain, P. Knab, H. Gall: A Tool for Visual Understanding of Source Code Dependencies. *Proc. ICPC 2008*: 254-259

[14] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization*. The MIT Press, 1998.

[15] R. Wettel, M. Lanza, and R. Robbes, Software Systems as Cities: A Controlled Experiment, *Proc. ICSE 2011*, pp. 51 - 560, ACM Press, 2011.

[16] R. Wettel and M. Lanza. Visually localizing design problems with disharmony maps. *Proc. Softvis 2008*, pp. 155–164. ACM Press, 2008.

[6] http://esec-fse.inf.ethz.ch/cfp_new_ideas.html