BugVis: Commit Slicing for Fault Visualisation

David Bowes
Lancaster University
Lancaster, United Kingdom
d.h.bowes@lancaster.ac.uk

Jean Petrić
Lancaster University
Lancaster, United Kingdom
j.petric@lancaster.ac.uk

Tracy Hall
Lancaster University
Lancaster, United Kingdom
tracy.hall@lancaster.ac.uk

ABSTRACT

In this paper we present BugVis, our tool which allows the visualisation of the lifetime of a code fault. The commit history of the fault from insertion to fix is visualised. Unlike previous similar tools, BugVis visualises only the lines of each commit involved in the fault. The visualisation creates a commit slice throughout the history of the fault which enables comprehension of the evolution of the code involved in the fault.

KEYWORDS

commit, fault, bug, fix, visualisation

ACM Reference Format:
David Bowes, Jean Petrić, and Tracy Hall. 2020. BugVis: Commit Slicing for Fault Visualisation. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3387904.3389299

1 INTRODUCTION

Fixing code faults can be challenging and it is not uncommon for secondary faults to be introduced during a fix. Implementing effective fixes relies on developers understanding all the lines of code involved in a fault. Ko et al [6] and LaToza & Myers [7] report that understanding historical changes in code is one of the most time-consuming activities in software development. Understanding previous fault fixes is particularly difficult as fault fixing commits are often bundled up with other changes and refactorings. Such bundled commits make it difficult to understand the history of specific faults. Using code analysis in relation to faults is valuable as important questions related to code maintenance can be answered, such as "where, when and why was the fault inserted?" [8].

A range of existing tools are available to assist developers to make sense of historical changes. Many of these tools focus at a higher level of granularity than the line of code level at which BugVis works. BugMaps works at the class-level [4], whilst HATARI operates at the method-level [11]. Even though these tools provide insights to a fault's whereabouts, they do not pin point the exact location of faulty lines throughout their history. Other tools, such as CHRONOS [9] do visualise historical changes at the code line-level. However, the CHRONOS approach analyses changes of any code snippet, not specifically the lines of a code fault from insertion to fix. With CHRONOS the developer still needs to identify the exact location of the fault. BugVis not only visualises the lines of code specific to a particular fault from insertion to fix, but also enables developers to identify changes not associated with the fix under examination.

In this paper we present our tool BugVis which allows developers to see where and how fault reports have been fixed in the history of a file. BugVis also allows the developer to see all files and the changes that were made to them, highlighting the changes to lines and commits while still giving the full context of the code around the changes. Some faults go back beyond the initial indicated problem and can be missed by previous approaches, but BugVis tracks back to the possible origin of the fault and beyond. Unlike other visualisation tools for fault investigation, BugVis provides the minimum complete history of faulty lines presented in the context of the file in which the faulty lines occurred. BugVis identifies for the developer *where* and *when* a fault was introduced. BugVis provides a program slice based only on committed lines of code associated with the fault being analysed. Viewing only lines of code relevant to the fault in question makes reasoning about that fault easier.

BugVis implements the SZZ algorithm [10] for initially linking fault reports to fault fixes and backtracking to identify the fault insertion commits. We have improved the original SZZ algorithm by: a) linking deleted as well as modified lines b) using advanced diff commands which follow blocks of code being moved. Both of these changes improve linkage from fault fix to insertion point [1]. In addition, BugVis is an interactive tool allowing developers to select individual lines and see where they came from and where they go to. The interactive ability of the tool allows people to investigate code which may be surrounding the fault changes.

In the next section we discuss related work and previous fault analysis tools. In Section 3 we describe how BugVis was developed and works. Finally we discuss how BugVis has been used and evaluated in Section 4.

2 RELATED WORK

Fault analysis is challenging due to the availability of suitable tools for tracking historical changes. Previous tools for fault analysis have mostly focused on tracking the history of faulty code at the file and method level. Other tools track changes at the line level, however the historical analysis is limited to any code snippets. Currently no tools offer a complete visualisation solution to follow faults at the line level across their lifetime.

Various visualisation tools exist to assist developers in understanding software faults. Hora et al. introduced BugMaps, a visualisation tool for faulty code [4]. Their tool links bug tracking systems with version control commits to find classes involved in the fix. BugMaps presents the developer with two visualisation modes, one to visualise the history of faulty code and the other to visualise

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389299

the commits of interest. Both modes show meta information about the code, which includes the classes involved, static code metrics and historical numbers of faults. Couto et al. extended BugMaps to include causality relationships between source code metrics and faults in the tool called BugMaps-Granger [2]. BugMaps-Granger uses past changes of source code metrics to predict changes in the number of faults over a period of time. The tool is primarily designed to assist developers in refactoring and unit testing activities. Whilst the BugMaps tools focus on visualising the file level (i.e. class) changes of already fixed faults, the HATARI tool marks risky methods within classes that are likely to be faulty [10]. HATARI uses the original SZZ algorithm [11] to build models for predicting fault-prone code. The tool visualises methods with high and low risk allowing the developer to navigate through the historical changes of the code. BugVis on the other hand uses the improved SZZ algorithm [1] to follow faults from insertion to fix at the line-level of granularity.

Several tools support the historical analysis of code at the line level. Servant and Jones introduced CHRONOS, a tool which enables the visualisation of change history of target lines across all historical versions of code [9]. Their tool uses a query mechanism to select the lines of interest, where the lines can be contiguous or disparate. Wittenhagen et al. developed Chronicler, a tool that uses a tree representation of code to track changes across the file's history [13]. In Chronicler each line of code is part of a tree which a developer can use to analyse its evolution. Yoon et al. introduced Azurite which helps developers navigate source code by tracking its history using diff [14]. Azurite uses a "replay" approach, where the historical code changes are supported by the timeline which the developer can use to replay the development of code as it happened at a specific time frame. To date, no tool working at the line level granularity offers the tracking of faulty lines of code.

Existing tools have limited functionalities for analysing historical changes of faulty code. The BugMaps tools do not indicate specific location in code where a fault was fixed. However, developers are often interested in source code at the line level [3, 7]. The HATARI tool mostly focuses on the present state of the code being designed to indicate fault-prone methods. Other tools such as CHRONOS, Chronicler and Azurite offer the visualisation of historical changes at the line level, however without targeting faulty lines. On the other hand, BugVis combines and extends the strengths of existing tools to enable developers to effectively analyse the history of faulty code at the line level. BugVis is a complete visualisation approach that enables comprehension of the evolution of a code fault.

3 HOW *BugVis* WORKS

BugVis is a visualisation tool which combines information from code repositories and bug databases and visualises the history of potentially faulty lines of code. BugVis has three modules which extract information from different sources in order to produce the final visualisation. These modules are a Linker which matches closed bugs to file commits, a Line Mapper, which allows lines to be followed during the history of a file and a Bug Backtracker which identifies the changes in the fix commit and the lines previous to them which are likely to be faulty. Different modules require information from different sources, some of which are external to BugVis, and others are generated on the fly. Figure 1 shows a schematic of BugVis indicating how different modules interact in order to generate the visualisation of the changed files.[1]

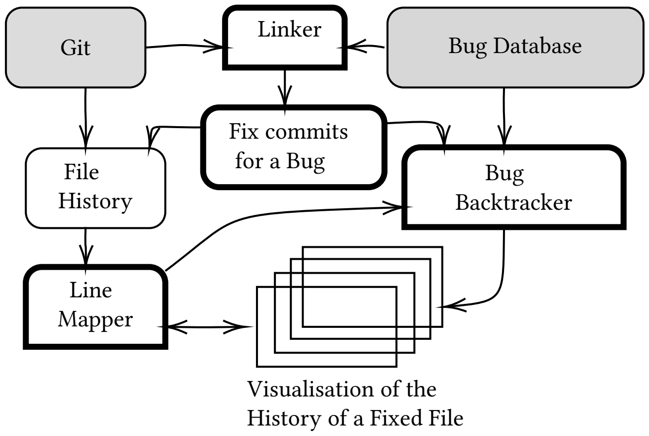


Figure 1: Schematic of BugVis. Gray boxes are external to BugVis. Bordered boxes implement the SZZ algorithm.

3.1 Implementation

3.1.1 Connections to data sources. BugVis collects data from a cloned Git repository and an online bug database (e.g. GitHub issues). BugVis uses the operating system's Git installation to interact with the cloned Git repository to find all commits using Git log. BugVis can connect other bug databases, including, GitHub, Jira, Bugzilla and XML. Using a connection to a bug database, BugVis extracts all bugs which are labeled as faulty. The label may be different for different projects, therefore, BugVis allows the label to be specified at run-time.

3.1.2 Bug Linking. To identify commits responsible for fixing a bug, BugVis matches the commit messages from Git log with the bug ids from the bug database. BugVis is configurable so that a customisable regex is used to find relevant links. The connection to the data sources and the bug linking occur at the start of the program. The bug database, the file history and the fix commits for a bug are held in memory. The bug database can be saved locally to a reduced XML format to allow BugVis to load a cached version of the bugs, reducing the need to connect remotely to the bug database.

3.1.3 Line Mapping. BugVis implements an algorithm for annotating lines of a file similar to Williams and Spacco [12]. The native Git log with diffs is used to both follow renames and chunk movements. We did not include line matching using Levenstein's distance because we are interested in the general location of faults which can be achieved using the Git log commands. Git reverse blame also allows us to identify lines in the previous commit which did not end up in the final fix commit. Line mappings are held in memory while generating the bug backtracking information needed for interactively visualising a file.

[1]It is possible to use BugVis without a bug database purely to visualise the changes to a file up to a particular commit.

3.1.4 Bug Backtracking. Although a line might have been changed in the fix commit, it may have been part of a change started after the bug was reported. As such, the line was not part of the fault insertion because it has changed since the original report and we cannot determine if the change was due to fixing the bug, or a refactoring. We use the date of bug reporting, combined with the line mapping data and the lines changed in the fault fix commit to generate annotations in BugVis to indicate lines which are faulty and commits which insert faults. We assume that any changes prior to a line marked as a fault insertion are not faulty. Clearly, this may not be true, however, BugVis allows an easy manual investigation of this hypothesis.

3.2 Tool application scenario

The scenario for this demo asks: "For the joda-time project on GitHub: what commits fixed bug #93 and what lines in which files were fixed and where did the faulty lines originate?". In Figure 2a, the tool has extracted the bugs from the repository and displayed them as a list. Figure 2 shows a typical scenario for using the tool which starts with the BugVis having been configured to use the joda-time GitHub repository[2] and the associated GitHub issue tracking database. Once BugVis starts, the data is loaded from both sources, or from locally cached data. Once the data has been loaded, bug links are discovered. At its simplest, BugVis can be used to show which commits are linked to bugs.

If we select bug #93 from Figure 2a, BugVis shows a list of commits (`9a62b`[3] and `8612f`[4]) which have been identified to fix bug #93, Figure 2b. Selecting the first commit (`9a62b`) populates the next list with the files changed in that commit. The list shows all the files changed in the fix commit.

In this scenario, selecting the file `org/joda/time/Partial.java` displays a timeline of windows each showing the code for the file in a particular commit Figure 2c. Each line of the file is coloured. Green indicates the addition of lines. Yellow indicates modified lines and red shows lines which are deleted. Lines highlighted in blue show that the user has interactively selected a particular line. The gutter of prior files also contain icons which show if a change in a line is associated with the fault fix using the SZZ algorithm. Clicking on a line in the last version of the file Figure 2c (highlighted in blue), then backtracks through the history to show how that line has been affected Figure 2d. Clicking on a line will also align all of the code windows to show roughly the equivalent line. In Figure 2d, the penultimate file shows the lines which are modified and the hash of the last commit of that line. The gutter contains markers which indicate that the line has been fixed but was present prior to the bug being reported[5]. Figure 2e shows that the bug had been traced back to the original version of the file.

4 HOW HAS *BugVis* BEEN USED

BugVis has been evaluated and used in a range of different circumstances. The tool has been an integral part of working with industrial collaborators and has allowed us to gain new insights into faults and their fixes. BugVis has also allowed us to evaluate and improve our version of the SZZ algorithm. The visualisation of the code has allowed us to check the lines which SZZ marks as faulty and it has enabled us to verify the point where SZZ has identified the fault insertion point.

Table 1: Bugs identified by different systems for joda-time

	BugVis		
Defects4J	Not linked	Linked	
Not linked	92	38	130
Linked	11	14	25
Totals	103	52	155

We evaluated BugVis against the extensively used Defects4J data [5]. We used the GitHub repository for joda-time and the issue tracking system on GitHub. We linked bugs using a simple regex:
`(?i:(^|\\W)fix.*#bugid(\\W|$))`
Where `bugid` is replaced by the id from the bug database before linking starts. We restricted our analysis to the same time range as the original Defects4J analysis. Defects4J uses a manual approach to bug linking and therefore finds links to commits which contain no indication of the `bugid` from the bug database. Table 1 shows the number of bugs linked to commits by Defects4J and BugVis. In total, there are 92 faults which neither tool associated a bug to a commit. Defects4J identified 11 commits linked to bugs which BugVis did not find. BugVis identified 38 bugs which were linked to commits and both tools agreed on links for 14 bugs. Although the agreement is only slight (Kappa= 0.186), BugVis allowed us to manually confirm the links made by BugVis and those made by Defects4J. `Bugid` #96 was found by BugVis and not Defects4J. While inspecting the code changes for `bugid` #96 using BugVis, it was possible to see how `bugid` #93 had not been correctly closed by commits (`9a62b` or `8612f`) and further work had been needed.

We have also successfully used BugVis with professional software developers to investigate the type of faults software professionals insert into their code. Our aim being to understand whether particular software developers insert particular types of faults. This study involved 15 professional software developers. We used BugVis to visualise faults inserted by individual developers then asked each developer to explain the context and the characteristics of that fault. This is an on-going study in which BugVis has proved to be effective at communicating information about individual faults between professional developers and researchers.

ACKNOWLEDGEMENTS

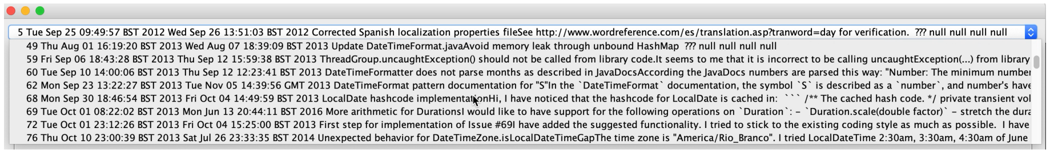
This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number: EP/S005730/1

REFERENCES

[1] D. Bowes, S. Counsell, T. Hall, J. Petric, and T. Shippey. 2017. Getting Defect Prediction Into Industrial Practice: the ELFF Tool. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 44–47. https://doi.org/10.1109/ISSREW.2017.11

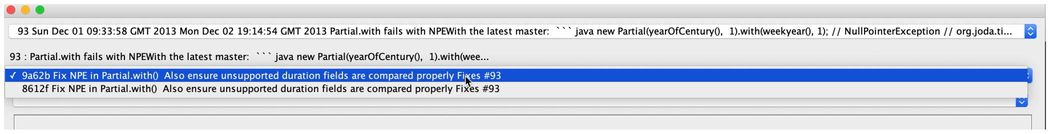
[2] Cesar Couto, Marco Tulio Valente, Pedro Pires, Andre Hora, Nicolas Anquetil, and Roberto S Bigonha. 2014. BugMaps-Granger: a tool for visualizing and predicting bugs using Granger causality tests. *Journal of Software Engineering Research and Development* 2, 1 (2014), 1.

[2] `https://github.com/JodaOrg/joda-time`
[3] `https://github.com/JodaOrg/joda-time/commit/9a62b06`
[4] `https://github.com/JodaOrg/joda-time/commit/8612f9e`
[5] Lines changed after the bug is first reported are less likely to be the original fault and are therefore not marked as faulty.



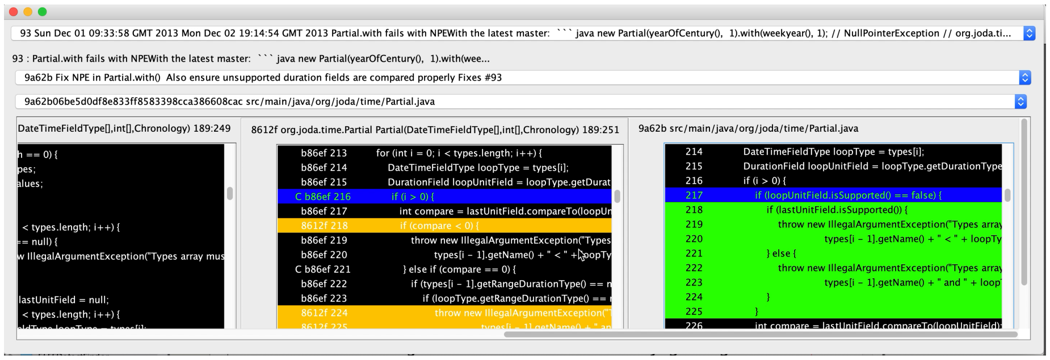
```
5 Tue Sep 25 09:49:57 BST 2012 Wed Sep 26 13:51:03 BST 2012 Corrected Spanish localization properties fileSee http://www.wordreference.com/es/translation.asp?tranword=day for verification.  ??? null null null null
  49 Thu Aug 01 16:19:20 BST 2013 Wed Aug 07 18:39:09 BST 2013 Update DateTimeFormat.javaAvoid memory leak through unbound HashMap  ??? null null null null
  59 Fri Sep 06 18:43:28 BST 2013 Thu Sep 12 15:59:38 BST 2013 ThreadGroup.uncaughtException() should not be called from library code.It seems to me that it is incorrect to be calling uncaughtException(...) from library
  60 Tue Sep 10 14:00:06 BST 2013 Thu Sep 12 12:23:41 BST 2013 DateTimeFormatter does not parse months as described in JavaDocsAccording the JavaDocs numbers are parsed this way: "Number: The minimum number
  62 Mon Sep 23 13:22:27 BST 2013 Tue Nov 05 14:39:56 GMT 2013 DateTimeFormat pattern documentation for "S"In the `DateTimeFormat` documentation, the symbol `S` is described as a `number`, and number's have
  68 Mon Sep 30 18:46:54 BST 2013 Fri Oct 04 14:49:59 BST 2013 LocalDate hashcode implementationHi, I have noticed that the hashcode for LocalDate is cached in:  ``` /** The cached hash code. */ private transient vol
  69 Tue Oct 01 08:22:02 BST 2013 Mon Jun 13 20:44:11 BST 2016 More arithmetic for DurationsI would like to have support for the following operations on `Duration`: – `Duration.scale(double factor)` – stretch the dura
  72 Tue Oct 01 23:12:26 BST 2013 Fri Oct 04 15:25:00 BST 2013 First step for implementation of Issue #69I have added the suggested functionality. I tried to stick to the existing coding style as much as possible.  I have
  76 Thu Oct 10 23:00:39 BST 2013 Sat Jul 26 23:33:35 BST 2014 Unexpected behavior for DateTimeZone.isLocalDateTimeGapThe time zone is "America/Rio_Branco". I tried LocalDateTime 2:30am, 3:30am, 4:30am of June
```

(a) Selecting from the list of bugs retrieved from the yoda-time bug database.



```
93 Sun Dec 01 09:33:58 GMT 2013 Mon Dec 02 19:14:54 GMT 2013 Partial.with fails with NPEWith the latest master:  ``` java new Partial(yearOfCentury(),  1).with(weekyear(), 1); // NullPointerException // org.joda.ti...

93 : Partial.with fails with NPEWith the latest master:  ``` java new Partial(yearOfCentury(),  1).with(wee...

✓ 9a62b Fix NPE in Partial.with()  Also ensure unsupported duration fields are compared properly Fixes #93
  8612f Fix NPE in Partial.with()  Also ensure unsupported duration fields are compared properly Fixes #93
```

(b) Selecting from the list of commits in the Git repository which probably fix bug #93.



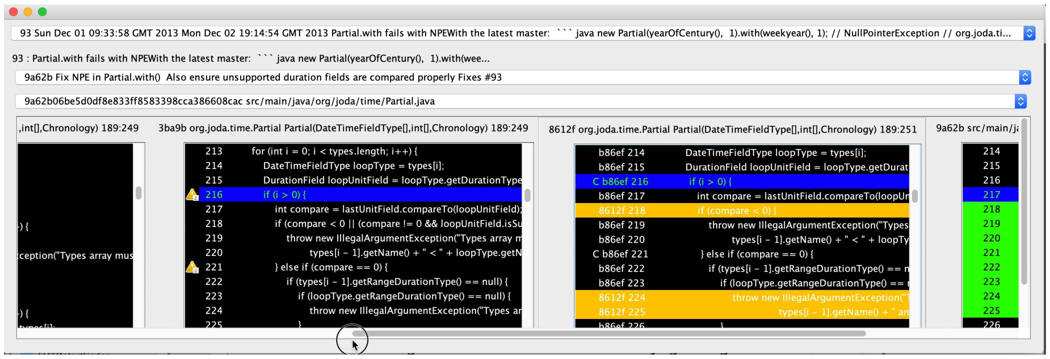
```
93 Sun Dec 01 09:33:58 GMT 2013 Mon Dec 02 19:14:54 GMT 2013 Partial.with fails with NPEWith the latest master:  ``` java new Partial(yearOfCentury(),  1).with(weekyear(), 1); // NullPointerException // org.joda.ti...

93 : Partial.with fails with NPEWith the latest master:  ``` java new Partial(yearOfCentury(),  1).with(wee...
9a62b Fix NPE in Partial.with()  Also ensure unsupported duration fields are compared properly Fixes #93
9a62b06be5d0df8e833ff8583398cca386608cac src/main/java/org/joda/time/Partial.java
```

```
DateTimeFieldType[],int[],Chronology) 189:249
h == 0) {
pes;
alues;

 < types.length; i++) {
== null) {
w IllegalArgumentException("Types array mus

lastUnitField = null;
 < types.length; i++) {
```

```
8612f org.joda.time.Partial Partial(DateTimeFieldType[],int[],Chronology) 189:251
  b86ef 213        for (int i = 0; i < types.length; i++) {
  b86ef 214            DateTimeFieldType loopType = types[i];
  b86ef 215            DurationField loopUnitField = loopType.getDurat
C b86ef 216            if (i > 0) {
  b86ef 217                int compare = lastUnitField.compareTo(loopUn
  8612f 218                if (compare < 0) {
  b86ef 219                    throw new IllegalArgumentException("Types
  b86ef 220                            types[i - 1].getName() + " < " + loopTy
C b86ef 221                } else if (compare == 0) {
  b86ef 222                    if (types[i - 1].getRangeDurationType() == n
  b86ef 223                        if (loopType.getRangeDurationType() == n
  8612f 224                            throw new IllegalArgumentException("T
```

```
9a62b src/main/java/org/joda/time/Partial.java
214            DateTimeFieldType loopType = types[i];
215            DurationField loopUnitField = loopType.getDurationType
216            if (i > 0) {
217                if (loopUnitField.isSupported() == false) {
218                    if (lastUnitField.isSupported()) {
219                        throw new IllegalArgumentException("Types array
220                                types[i - 1].getName() + " < " + loopTy
221                    } else {
222                        throw new IllegalArgumentException("Types array
223                                types[i - 1].getName() + " and " + loop
224                    }
225                }
226                int compare = lastUnitField.compareTo(loopUnitField);
```

(c) The View of `Partial.java` showing the fix commit 9a62b. Blue lines indicate the line clicked. Green shows lines added. Yellow shows lines changed.



```
93 Sun Dec 01 09:33:58 GMT 2013 Mon Dec 02 19:14:54 GMT 2013 Partial.with fails with NPEWith the latest master:  ``` java new Partial(yearOfCentury(),  1).with(weekyear(), 1); // NullPointerException // org.joda.ti...

93 : Partial.with fails with NPEWith the latest master:  ``` java new Partial(yearOfCentury(),  1).with(wee...
9a62b Fix NPE in Partial.with()  Also ensure unsupported duration fields are compared properly Fixes #93
9a62b06be5d0df8e833ff8583398cca386608cac src/main/java/org/joda/time/Partial.java
```

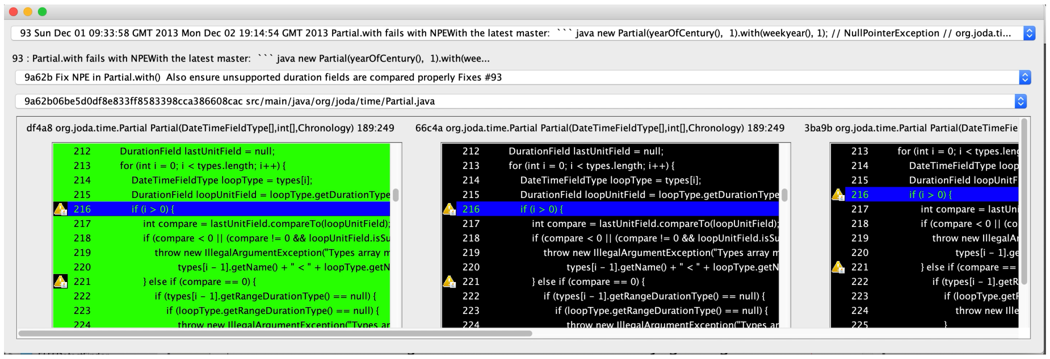
```
,int[],Chronology) 189:249

-) {

ception("Types array mus

-) {
```

```
3ba9b org.joda.time.Partial Partial(DateTimeFieldType[],int[],Chronology) 189:249
  213        for (int i = 0; i < types.length; i++) {
  214            DateTimeFieldType loopType = types[i];
  215            DurationField loopUnitField = loopType.getDurationType
⚠ 216            if (i > 0) {
  217                int compare = lastUnitField.compareTo(loopUnitField);
  218                if (compare < 0 || (compare != 0 && loopUnitField.isSu
  219                    throw new IllegalArgumentException("Types array m
  220                            types[i - 1].getName() + " < " + loopType.getN
⚠ 221                } else if (compare == 0) {
  222                    if (types[i - 1].getRangeDurationType() == null) {
  223                        if (loopType.getRangeDurationType() == null) {
  224                            throw new IllegalArgumentException("Types ar
  225                        }
```

```
8612f org.joda.time.Partial Partial(DateTimeFieldType[],int[],Chronology) 189:251
  b86ef 214            DateTimeFieldType loopType = types[i];
  b86ef 215            DurationField loopUnitField = loopType.getDurat
C b86ef 216            if (i > 0) {
  b86ef 217                int compare = lastUnitField.compareTo(loopUn
  8612f 218                if (compare < 0) {
  b86ef 219                    throw new IllegalArgumentException("Types
  b86ef 220                            types[i - 1].getName() + " < " + loopTy
C b86ef 221                } else if (compare == 0) {
  b86ef 222                    if (types[i - 1].getRangeDurationType() == n
  b86ef 223                        if (loopType.getRangeDurationType() == n
  8612f 224                            throw new IllegalArgumentException("T
  8612f 225                                    types[i - 1].getName() + " an
  b86ef 226                        }
```

```
9a62b src/main/ja
214
215
216
217
218
219
220
221
222
223
224
225
226
```

(d) Clicking on a line re-aligns all windows to show the history of the same line of code over time. The marks in the gutter indicate the possible location of the bug.



```
93 Sun Dec 01 09:33:58 GMT 2013 Mon Dec 02 19:14:54 GMT 2013 Partial.with fails with NPEWith the latest master:  ``` java new Partial(yearOfCentury(),  1).with(weekyear(), 1); // NullPointerException // org.joda.ti...

93 : Partial.with fails with NPEWith the latest master:  ``` java new Partial(yearOfCentury(),  1).with(wee...
9a62b Fix NPE in Partial.with()  Also ensure unsupported duration fields are compared properly Fixes #93
9a62b06be5d0df8e833ff8583398cca386608cac src/main/java/org/joda/time/Partial.java
```

```
df4a8 org.joda.time.Partial Partial(DateTimeFieldType[],int[],Chronology) 189:249
  212        DurationField lastUnitField = null;
  213        for (int i = 0; i < types.length; i++) {
  214            DateTimeFieldType loopType = types[i];
  215            DurationField loopUnitField = loopType.getDurationType
⚠ 216            if (i > 0) {
  217                int compare = lastUnitField.compareTo(loopUnitField);
  218                if (compare < 0 || (compare != 0 && loopUnitField.isSu
  219                    throw new IllegalArgumentException("Types array m
  220                            types[i - 1].getName() + " < " + loopType.getN
⚠ 221                } else if (compare == 0) {
  222                    if (types[i - 1].getRangeDurationType() == null) {
  223                        if (loopType.getRangeDurationType() == null) {
  224                            throw new IllegalArgumentException("Types ar
```

```
66c4a org.joda.time.Partial Partial(DateTimeFieldType[],int[],Chronology) 189:249
  212        DurationField lastUnitField = null;
  213        for (int i = 0; i < types.length; i++) {
  214            DateTimeFieldType loopType = types[i];
  215            DurationField loopUnitField = loopType.getDurationType
⚠ 216            if (i > 0) {
  217                int compare = lastUnitField.compareTo(loopUnitField);
  218                if (compare < 0 || (compare != 0 && loopUnitField.isSu
  219                    throw new IllegalArgumentException("Types array m
  220                            types[i - 1].getName() + " < " + loopType.getN
⚠ 221                } else if (compare == 0) {
  222                    if (types[i - 1].getRangeDurationType() == null) {
  223                        if (loopType.getRangeDurationType() == null) {
  224                            throw new IllegalArgumentException("Types ar
```

```
3ba9b org.joda.time.Partial Partial(DateTimeFie
  213        for (int i = 0; i < types.leng
  214            DateTimeFieldType loop
  215            DurationField loopUnitF
⚠ 216            if (i > 0) {
  217                int compare = lastUni
  218                if (compare < 0 || (co
  219                    throw new IllegalAr
  220                            types[i - 1].ge
⚠ 221                } else if (compare ==
  222                    if (types[i - 1].getR
  223                        if (loopType.getF
  224                            throw new Ille
  225                        }
```

(e) View showing that the original file contained the bug.

Figure 2: Screen shots of using the tool

[3] Reid Holmes and Andrew Begel. 2008. Deep intellisense: a tool for rehydrating evaporated information. In *Proceedings of the 2008 international working conference on Mining software repositories*. 23–26.

[4] Andre Hora, Nicolas Anquetil, Stephane Ducasse, Muhammad Bhatti, Cesar Couto, Marco Tulio Valente, and Julio Martins. 2012. Bug maps: A tool for the visual exploration and analysis of bugs. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 523–526.

[5] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[6] Andrew J Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 344–353.

[7] Thomas D LaToza and Brad A Myers. 2010. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*. 1–6.

[8] Francisco Servant. 2013. Supporting bug investigation using history analysis. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 754–757.

[9] Francisco Servant and James A Jones. 2013. Chronos: Visualizing slices of source-code history. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–4.

[10] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. Hatari: raising risk awareness. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 107–110.

[11] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[12] Chadd C Williams and Jaime W Spacco. 2008. Branching and merging in the repository. In *Proceedings of the 2008 international working conference on Mining software repositories*. 19–22.

[13] Moritz Wittenhagen, Christian Cherek, and Jan Borchers. 2016. Chronicler: Interactive exploration of source code history. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 3522–3532.

[14] YoungSeok Yoon, Brad A Myers, and Sebon Koo. 2013. Visualization of fine-grained code change history. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 119–126.