Griotte: Improving Code Review with Fine-Grained IDE Events

Skip Lentz
EEMCS
Delft University of Technology

Martín Dias
RMoD
INRIA Lille-Nord Europe

Damien Cassou
RMoD
INRIA Lille-Nord Europe

***Abstract*—Code review is a difficult process because: (1) developers often create *tangled commits*, (2) a change may be scattered across many different parts of a project, (3) many changes are shadowed, (4) commit messages can be inaccurate or wrong.**

This work aims to propose a solution to these problems by exploiting the information provided by fine-grained IDE events.

To put this solution in practice, we will develop a code review tool named *Griotte* in the *Pharo* IDE.

***Index Terms*—code review, fine-grained IDE events, Epicea, Griotte, Pharo**

I. Problem Description

Modern Code Review (MCR) is an important mechanism for quality assurance in software development: it provides feedback and helps to avoid introducing bugs. However, it is a difficult task for a reviewer to perform because:

a) Tangled commit: A tangled commit is a commit which is the result of multiple, mutually unrelated changes. For example, a commit could consist of a refactoring, some formatting changes, and a bug fix.

The process of understanding a change is difficult on its own. In the case of a tangled commit, the reviewer in addition has to understand several unrelated changes at the same time.

Herzig & Zeller found in a study of five open-source Java projects, that up to 15% of all bug fixes consist of multiple unrelated changes [1].

b) Line-based view: Commits may have changes scattered across many different parts of a project. For example, a method rename impacts all method references. However, even though the changes belong together, they are not shown together within the changes browser of the code review tool. For this reason, we say that these review tools are *line-based*.

Thus, the reviewer has to analyze the changes line-by-line to conclude that it was a refactoring, which is not a trivial task.

c) Shadowed changes: Negara *et al.* found that 37% of changes are *shadowed*, *i.e.* overridden by subsequent changes in the same line, file and commit [2]. The shadowed changes can be important information for the reviewer to find out how the developer arrived at their solution.

d) Wrong or lack of commit descriptions: Commit descriptions can help the reviewer in understanding the reason of a change. Without the description, the reason for a change is lost. Furthermore, a wrong description has the potential to mislead the reviewer.

II. Proposed Solution

This section aims to propose several approaches which use the information provided by *fine-grained IDE events*. First, we will explain what fine-grained IDE events are. Then we will discuss how they might be used to solve the problems mentioned in Section I.

A. Fine-grained IDE events

Normally when a developer works on a feature or fixes a bug, the changes are recorded at the point of commit. The actions the developer has performed to get to that state are lost. The actions performed between starting work and the point of commit are known as *fine-grained IDE events* — in contrast with the *coarse-grained* changes one might find in a VCS (Version Control System) commit.

Some examples of fine-grained IDE events are a method modification, a class addition, a refactoring or a test run. The events are recorded and stored as the developer is working in the IDE. A minimized version of a model can be found in Figure 1.

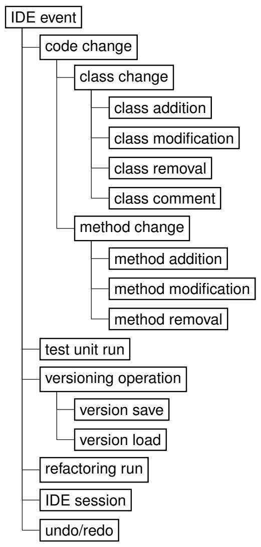

Fig. 1. A minimized class hierarchy of a model for fine-grained IDE events as described by Dias *et. al.* [3]

B. *Code review using fine-grained IDE events*

There are several approaches where fine-grained IDE events are useful in the context of code review. Each approach corresponds to one or more of the problems listed in Section I.

e) Displaying fine-grained history: An approach is to display the information to the reviewer. This allows the reviewer to see how a developer evolved to the submitted change. This solves problem c) in that the review tool shows extra information which is typically shadowed by other changes.

f) Grouping changes: An approach to solve problems a) and b) is to group changes. A group of changes consists of one or more code changes which are mutually related, along with a descriptive label (e.g. a refactoring of method name `foo` to `bar`). By grouping the changes, we change the perspective from a *line*-based review tool to a *change*-based review tool.

Work has been done in this area: Tao and Kim [4] have explored grouping (referred to as *partitioning*) tangled commits (*composite changes*) in the context of code review. However, they partitioned changes using only the information available from the VCS.

Dias *et al.* [3] have done work on untangling commits using fine-grained IDE events, with good results; namely 88% accuracy in determining whether two code changes belong together.

We aim to combine both approaches — groups in the context of review and grouping using fine-grained IDE events.

g) Generating commit descriptions: To solve problem d), we can assist the developer in writing commit descriptions. For example, if the IDE recorded the renaming of a method `foo` to the name `bar`, a commit description such as `Method rename: foo -> bar` can be generated. Using this approach, the developer is partially alleviated from coming up with a proper description.

III. Implementation: Griotte

To put the approaches mentioned in Section II in practice, we will develop a code review tool named *Griotte*[1]. The core concept in the architecture of Griotte is the usage of existing services which provide code review features (*e.g. GitHub* or *Gerrit*). See Figure 2 for a simplified diagram of the architecture.

Griotte will be implemented in the *Pharo* IDE [5][2]. Furthermore, we will use *Epicea* [6] to monitor and access fine-grained IDE events.

First, a brief description of Epicea is be given. Afterwards, we discuss the implementation of Griotte in more detail.

A. *Epicea*

Epicea is a Pharo project providing a model for first-class code changes and related IDE tools. Epicea records IDE events during development, allowing us to analyze a more fine-grained history of the codebase. This is in contrast with the code change information available from a VCS repository, which is much more coarse-grained.

[1]Current work in progress: http://smalltalkhub.com/#!/~Balletie/Griotte

[2]Pharo: http://pharo.org

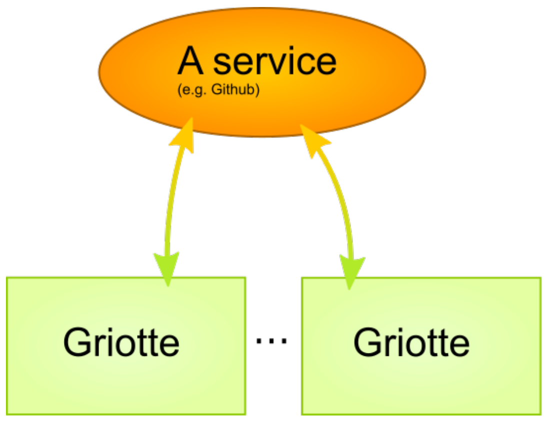

Fig. 2. Simplified architecture diagram of Griotte

The model of Epicea is described by Figure 1. It can be divided into *low-level* and *high-level* IDE events.

Low-level IDE events are code changes — additions, deletions or modifications — of program components such as classes or methods. Examples are a method modification, a class addition, etc.

Secondly, Epicea records IDE events such as a test run (and its outcome), refactorings, and loading a version from the VCS. These are known as high-level IDE events.

Epicea records this data by means of the *Epicea Monitor*, which listens to events in Pharo as they happen, and converts them to the first-class code change model objects described in Figure 1. These objects are then stored and persisted in a log.

B. *Griotte*

An important part of Griotte is the usage of existing services which provide code review features. Examples of these services are *GitHub* or *Gerrit*. The extra information provided by the fine-grained IDE events will be stored as metadata. For example, in `git`-based repository services, we can use `git-notes`[3] to store metadata on commits. The Griotte client within Pharo will then be used for the features using the fine-grained IDE events.

A good advantage of using existing services is that this approach makes it possible to use Griotte in parallel with other workflows, since the metadata is stored in a manner transparent to common workflows.

Furthermore, one does not need to maintain their own servers if one chooses to use GitHub or similar services. The maintenance of both server-side code and the servers will be done by third-parties.

A disadvantage is that we need to sacrifice some of our own flexibility, and rely on the flexibility of the API's of external services like GitHub. In a research project this might seem less ideal, as one would like to be free to explore different options.

[3]Documentation: http://git-scm.com/docs/git-notes

IV. Conclusion

To conclude and summarize, we presented the following difficulties with code review:

(1) Developers often create *tangled commits*. Reviewers have to understand multiple unrelated changes at the same time.
(2) Code review tools are line-based. Commits may touch many different parts of a project, and the reviewer has to analyze each change line-by-line even if they are related to the same change.
(3) Many changes are shadowed, and are thus not accessible by the reviewer.
(4) Commit messages can be inaccurate or wrong, which misleads the reviewer or does not provide enough information.

We proposed several approaches which use fine-grained IDE events in the context of code review:

(1) Display fine-grained history to the reviewer, to provide extra information if it's needed.
(2) Group mutually related changes together using the fine-grained IDE events as input.
(3) Generate accurate commit descriptions using the information provided by fine-grained IDE events.

Furthermore, we presented our proposal for an implementation of a code review tool named Griotte which (1) uses these approaches, (2) is implemented in the Pharo IDE and (3) uses Epicea as a source for fine-grained IDE events. A key idea of our implementation is the use of existing external services such as GitHub and Gerrit. We discussed the benefits and limitations of this approach.

References

[1] K. Herzig and A. Zeller, "Untangling changes," *Unpublished manuscript*, Sep. 2011. [Online]. Available: https://www.st.cs.uni-saarland.de/publications/files/herzig-tmp-2011.pdf

[2] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?" in *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[3] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *SANER'15: Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, Montreal, Canada, 2015, pp. 341–350, (candidate for IEEE Research Best Paper Award). [Online]. Available: https://hal.inria.fr/hal-01116225

[4] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR 2015, 2015.

[5] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Kehrsatz, Switzerland: Square Bracket Associates, 2009. [Online]. Available: http://pharobyexample.org/,http://rmod.lille.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf

[6] M. Dias, D. Cassou, and S. Ducasse, "Representing code history with development environment events," in *IWST'13: International Workshop on Smalltalk Technologies 2013*, 2013. [Online]. Available: http://rmod.lille.inria.fr/archives/papers/Dias13a-IWST13-Epicea.pdf