Representing Code History with Development Environment Events

Martín Dias Damien Cassou Stéphane Ducasse

RMoD
Inria Lille–Nord Europe — University of Lille — Lifl

Abstract

Modern development environments handle information about the intent of the programmer: for example, they use abstract syntax trees for providing high-level code manipulation such as refactorings; nevertheless, they do not keep track of this information in a way that would simplify code sharing and change understanding. In most Smalltalk systems, source code modifications are immediately registered in a transaction log often called a ChangeSet. Such mechanism has proven reliability, but it has several limitations. In this paper we analyse such limitations and describe scenarios and requirements for tracking fine-grained code history with a semantic representation. We present Epicea, an early prototype implementation. We want to enrich code sharing with extra information from the IDE, which will help understanding the intention of the changes and let a new generation of tools act in consequence.

Keywords Source-code change meta-model; Collaboration; Continuous Versioning; Explore-first Programming

1. Introduction

Modern integrated development environments (IDEs) can have information about the intent of the programmer: they use abstract syntax trees (ASTs) and provide high-level code manipulation (such as refactorings [3]). Nevertheless, they do not keep track of this information in a way that would simplify code sharing and change understanding. For example, after a few hours of work, developers might want to separately share the different changes they have worked on: documentation improvements, bug fixes, and feature additions are better committed separately to facilitate review and backtracking. If each change were semantically recorded, making separate commits would be much simpler: for example, a method renamed could be seen as just one high-level operation instead of many lines removes and added.

In this paper we describe scenarios, requirements, and an early prototype, named Epicea,[1] for tracking code history with a semantic representation. Based on Epicea, we want to enrich code sharing with extra information from the IDE, which will help understanding the intention of the changes and let tools act in consequence. For example, when a library developer updates an API (*e.g.,* by renaming a method), he can provide a dedicated semantic change to the library users so that they can update their client code automatically.

Structure of the paper. In Section 2 we describe the problem in current Smalltalk systems. In Section 3 a series of scenarios illustrate the key requirements for tracking changes semantically. We summarise such requirements in Section 4. We present the design of our prototype in Section 5. Section 6 has screenshots of our prototype in action. After a short overview of related work in Section 7 we conclude in Section 8.

2. Analysis of Current Smalltalk Systems

In most Smalltalk systems [4] source code modifications are logged immediately after any editing operation in a transaction log, often called a ChangeSet.[2] This transaction log acts as a tape recording source code changes. The programmer can navigate different versions of the code without requiring a traditional version control system (VCS), such as git, svn and Monticello. In addition, if the execution of the system is interrupted (*e.g.,* the virtual machine crashes or the process is killed), then such a log can be explored to recover and replay the sequence of changes.

While this log mechanism has proven to be reliable over the years, it has the following problems:

Barely structured text. There is a lack of abstraction. The log is a text file where each new event is appended at the end, as a sequence of chunks. Instead of represent-

[1] http://smalltalkhub.com/#!/~MartinDias/Epicea

[2] http://wiki.squeak.org/squeak/674

ing the events in a declarative format, the events are written as executable commands. The idea is that by re-evaluating them the original change is reproduced. This format makes it difficult for tools to recover semantic information.

Elementary model. A ChangeSet records only class, package and method definitions. As a result, ChangeSet lacks information about class modifications or high-level events such as refactorings.

Mixing sources and system events. ChangeSets mix source management (the state of a system) with system event recording (the steps to go from one state to the next). The same model and format is used for ChangeSets and the traditional in Smalltalk fileIn/fileOut mechanism. As a result, not all the events can be recorded (*e.g.,* refactorings, package loading). In addition, the granularity of the events is often too coarse, leading to problems on recovery. For example, instance variable addition and class addition are indistinguishable.

Losing intermediate states. ChangeSets only keeps track of what entities (*e.g.,* a class or method) has been modified. The intermediate states of such entities cannot be recovered but just the current one.

In this paper we introduce the notions of *Log* and *View* to fix the above-mentioned problems.

3. Scenarios for Changes as Programming Activity Traces

In this section we present several scenarios that illustrate the use of logs and their interplay in the IDE. We first define the vocabulary used in the rest of this paper.

Image. In a Smalltalk environment, an image is a snapshot of all the objects of the system, *i.e.,* a memory dump: this includes both the objects of the software under execution but also the classes and methods at the moment of the snapshot. An image acts as a cache with preloaded packages and initialised objects.

Session. An image can be launched, modified, and saved many times. We call each one of these periods a session.

Operation. We refer with this word to an action performed in a session. An operation can either have a duration in time (*e.g.,* an expression evaluation) or be a punctual fact (*e.g.,* a class addition). An operation can trigger other operations. In Figure 1, the list in the top represents a session where the developer has done three operations: (1) he has loaded the version 1 of a package named P using a VCS; (2) he has undone the addition of the class A from package P; (3) he has added a new class named B to package P. The light grey bullets and the horizontal alignment of the elements represent triggering (undoing the addition of class A has triggered the removal of A).

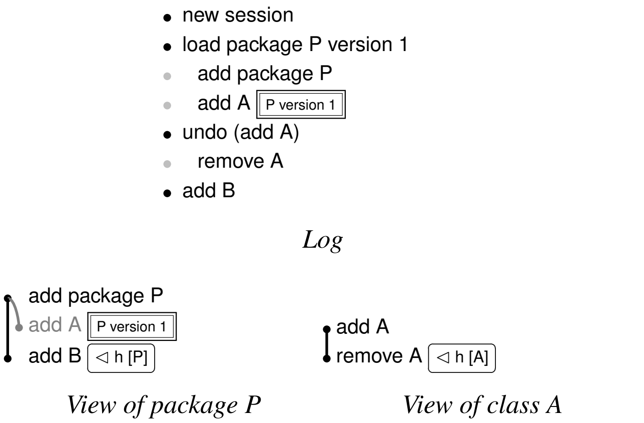
Event. We define an event as a representation of an operation. Some events represent a modification in the source code; we refer to them as *code changes*. Sometimes we say that an event triggered another event when the operation that the former event represents triggered the operation that the latter event represents.

Log. A log contains events recorded from the IDE. This includes, for example, class additions, method redefinitions, and refactorings. If the user does not save or if the system crashes, the log and the image will become desynchronised: *i.e.,* the log will contain information that is not in the image.

Code unit. In this paper we call code unit to a package, class, trait or method.

View. The log can have an overwhelming amount of information recorded about the system. This makes it difficult to understand the changes in a particular code unit. To solve this problem we include the concept of *view*. In Figure 1, views for the class A and package P are shown. The history of A is simple: it was added and then removed. The view of P is more complex: first, the class A was added, then this change got undone, and finally the class B got added (creating an implicit branch in the view). Each view has a head, marked as ◁ h [X], which represents the current state in the system for the code unit X. The current head will be the parent of the next change that affects this code unit and the head will be updated to point to this new change.

Commit. We call *commit* a particular version of source code stored in a VCS. In Figure 1, we mark the last change performed during the load of version 1 with the tag P version 1.



Log

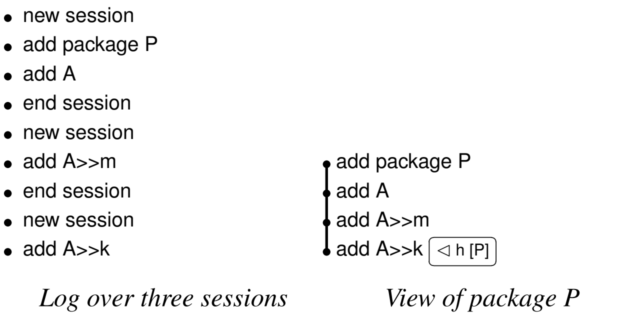
View of package P *View of class A*

Figure 1. Example.

3.1 Logs Transcend Sessions

Since a code unit can be edited over multiple sessions, the history of a code unit transcend history of images. In this section we discuss some scenarios that crosscut sessions.

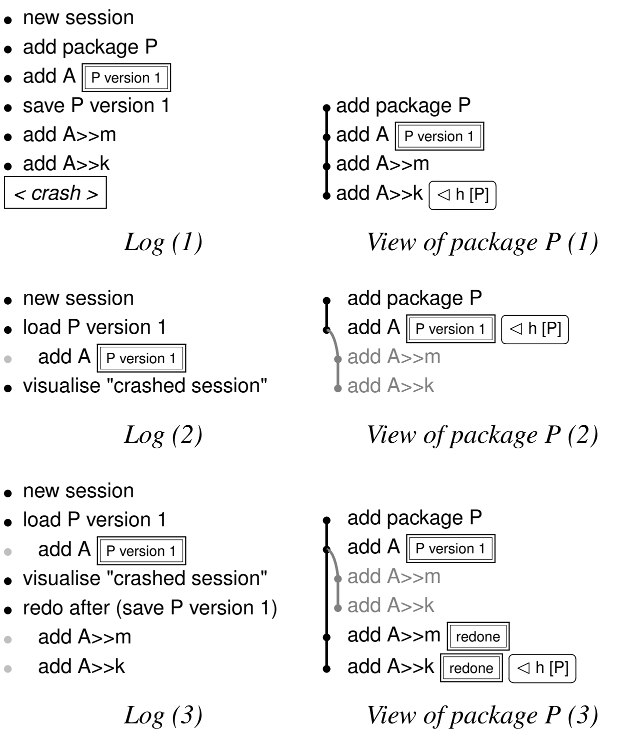
Tie the events of several sessions. In Figure 2 we show the history of the package P accumulated over three sessions. The view ignore session boundaries.



Log over three sessions *View of package P*

Figure 2. Views ignore session boundaries.

Recover lost changes after the IDE crashed. In Figure 3, the user created a package P with a class A and committed the package P to a VCS. After adding methods m and k, the IDE crashes. The user reopens the IDE, visualises the log of the crashed session, and redoes the lost changes. Such redone changes are shown as a new branch in the view. Each of those redone changes has a redone tag. Such a tag always references the original entry so the developer can analyse the event in the context where it was originally logged.



Log (1) *View of package P (1)*

Log (2) *View of package P (2)*

Log (3) *View of package P (3)*

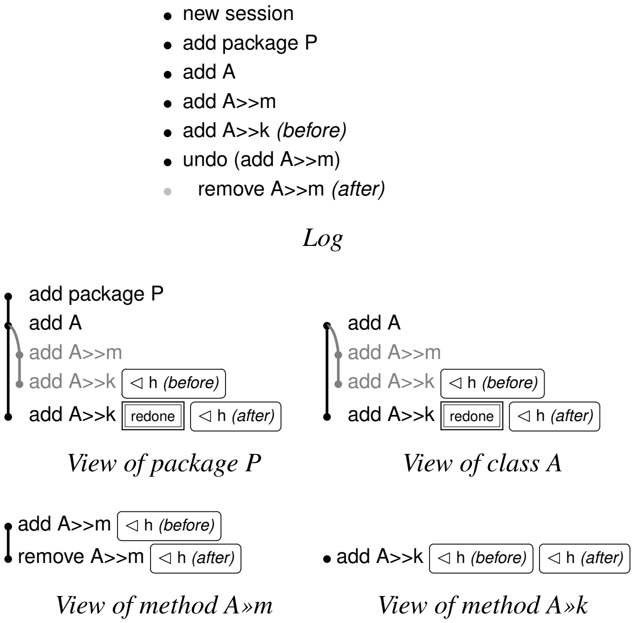
Figure 3. Redo lost changes after the IDE crashed.

Reload in fresh image. Since during experimentation images sometimes become unstable, it is a good practice to regularly rebuild from scratch the current head of development in a fresh image. Current infrastructure supports such practice by loading the code from the VCS, at the expense of losing the versions that occurred between two commits. The log overcomes such problems.

3.2 Code Operations

In this section we discuss some scenarios where navigation to previous versions of code or reorganisation of changes are important.

Undoing a code change. In Figure 4 we show that reverting the addition of method A»m has different effects on the different views. In the package and class views, the original method additions are shown in grey as a branch. In that way, the original history of events with the original chronology is available to be browsed. In the A»m view, the undo operation is seen as a removal of the method. For the A»k view the operation has no impact. Note that in each view there is a head pointing to a different event.



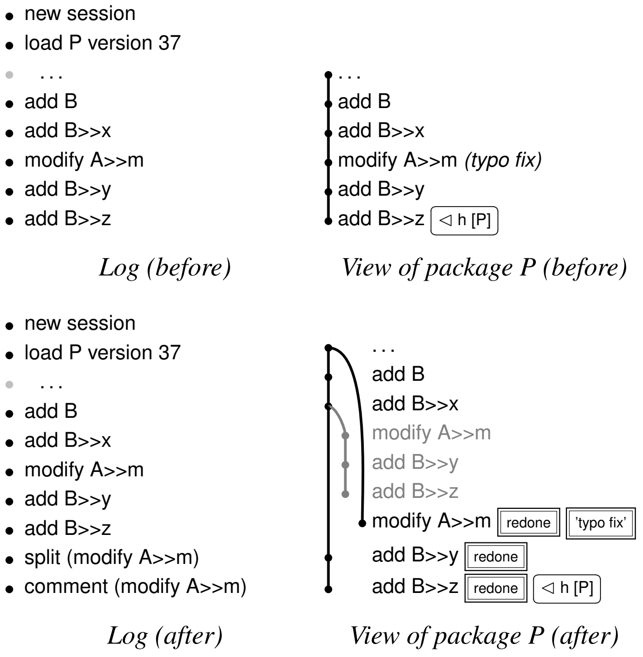
Log

View of package P *View of class A*

View of method A»m *View of method A»k*

Figure 4. Undoing the addition of A»m. The operation has different effects at package, class and method level.

Grouping changes before committing. When a developer is working for some time on a project, chances are that he will perform multiple independent tasks. This happens even when there is a concrete goal such as implementing a new feature or fixing a bug: either a typo, or some code that deserves a refactoring, or any other change that is unrelated to the goal can appear. Tools should make it easy for a developer to fix the off-topic issue and let him either mark it or split it to a different branch so the main branch stays focused and cohesive. We need a kind of cherry picking of the elements we want to commit. In Figure 5 we show an example of changes done in the package P, where the developer added a class B with some methods, and in the middle found and fixed a typo in the comment of A»m. He decides to create a new branch to keep this change separated from the other ones. He also adds a comment to the separated change (modify A»m) with a 'typo fix' tag.



Log (before) *View of package P (before)*

Log (after) *View of package P (after)*

Figure 5. Split changes for doing meaningful commits.

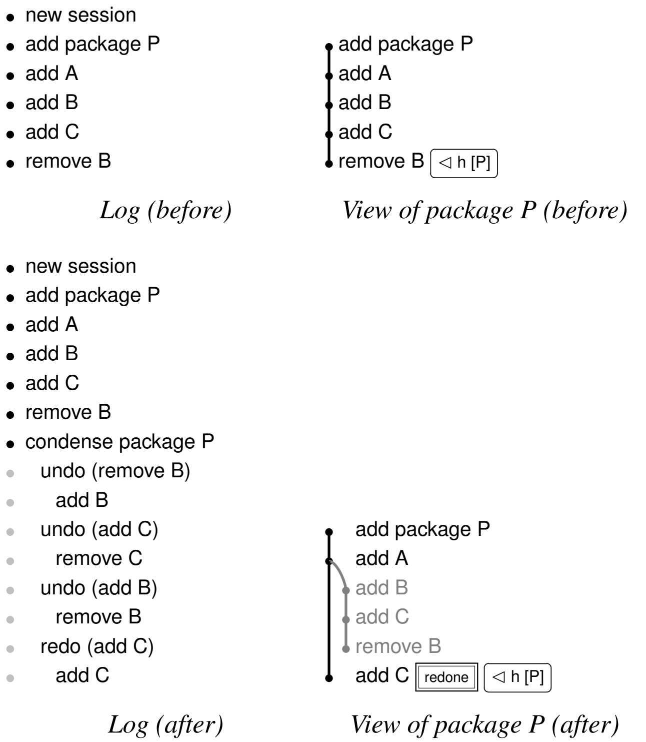
Commenting events. The developer can write arbitrary comments on an event (or group of events) to facilitate later understanding. We mentioned this feature in Figure 5, with the 'typo fix' tag. Additionally, the system can help the developer writing comments based on what triggered the related event.

Condensing code changes. The log might have changes that neutralise themselves (*e.g.,* a method is added and removed). In addition there are cases where the programmer may want to forget current history of certain entities. In Figure 6, we show in an example how the condense operation works when applied to the package P. Without any optimisation, the operation is done in two main steps: first, undo the events until the older neutralised event (remove B, add C, and add B); second, redo only the needed changes (add C).

Recording refactoring information. Some high-level operations, such as refactorings, group events. In Figure 7, a method is renamed (A»m) and all senders (B»k) of this method are updated. Each event related to the refactoring have a dedicated tag that references the high-level operation.

3.3 Sharing Events

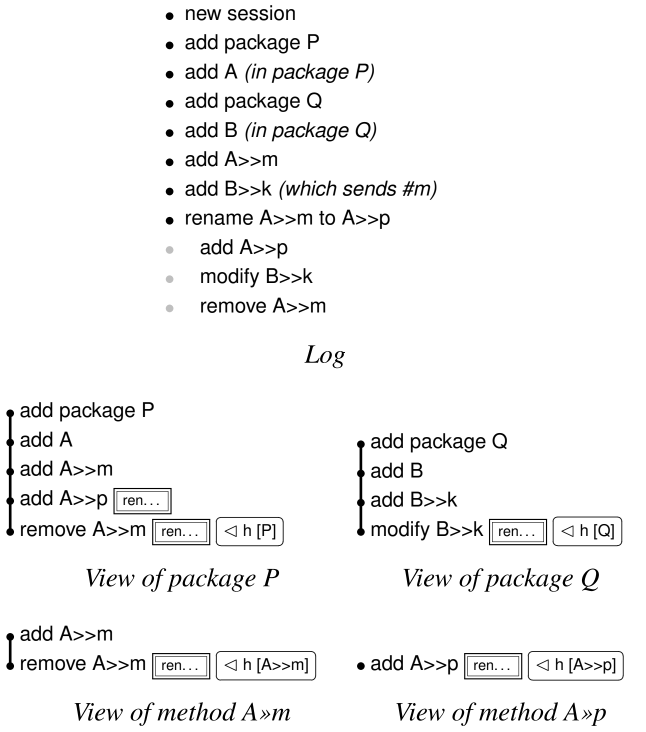
Logs and events can be shared between developers, projects, and images.



Log (before) *View of package P (before)*

Log (after) *View of package P (after)*

Figure 6. Condense operation.



Log

View of package P *View of package Q*

View of method A»m *View of method A»p*

Figure 7. Rename A»m to A»p. The method B»k uses it so it is modified by the refactoring.

Replaying a concrete event. When two projects are forks from each other, events of one fork can be replayed in the other.

Replaying the intent of a refactoring. When a library developer updates an API (*e.g.,* by renaming a method), he can provide high-level events which can be replayed by library users so that they can update their client code automatically.

4. Scenarios: an Analysis

We analysed several existing code change representations: ChangeSets, RingC [7], Cheops [2], NewChangeSystem,[3] and DeltaStreams.[4] From previous work and the scenarios presented above we define the following requirements.

4.1 Requirements

1. Replay and undo operations. Starting from the same or similar system, the information in the log should be enough for reconstructing the state of the system at any point of the log.
2. Log must be immediately persisted out of the volatile memory so information survives IDE crashes.
3. Log entries can have tags, *i.e.,* meta-information. A tag can reference another entry. Tags can be added after the entry has been persisted.
4. Events should be represented as first-class entities.
5. The change model should support modelling many different types of changes: structural elementary changes (method definitions), composed ones (refactorings), and system changes such as expression evaluation, redo, and branch creation.

5. Epicea

We implemented Epicea, an early prototype of the log and the event model. It was developed in Pharo [1]. Epicea model started as a branch of NewChangeSystem project and then was deeply modified and extended.

5.1 Event Model

In Figure 8 we show the class hierarchy of *events* we implemented in Epicea. The most important sub-hierarchy is the one of CodeChange, which represents the operation that made the code change, such as class creation, method modification, etc. Code changes hold enough information about the operation performed for either reverting the change or redoing it. Epicea uses Ring definitions to take snapshots of the involved code units.

We need to record information about the situation in which events are logged. That is the timestamp when it was done, the author who did it, the potential event that triggered it (for example, undoing a method addition triggers a method removal). This meta-information of the event is stored in log entries, as explained below.

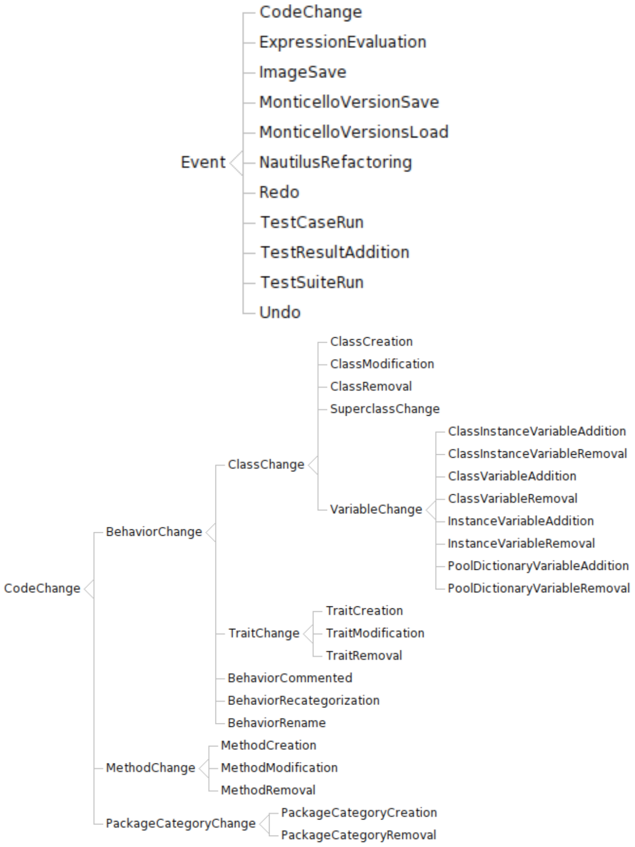


Figure 8. The hierarchies of Event and CodeChange used in our prototype.

5.2 Log Model

In Figure 9 an object diagram shows how a log is represented in the prototype. A log has a head pointing to the entry where the upcoming entry will be attached. Each entry points to a parent entry and the content event. In Figure 10 we show the design we implemented for Epicea. An entry has a dictionary of tags that allows attaching meta-information (author and timestamp). In the case of an event that triggers other events, each of these events has a tag pointing to the triggering event.

6. Revisiting the Scenarios

In Figure 11 an expression was evaluated. It triggered the load of the package named ConfigurationOfFuel. In turn, the load triggered many elemental code changes (package, class and method additions). In Figure 12 we show the log of an undo operation. The class A has been added in package P; then two methods have been added (A»m and A»k). Following, the undo of the addition of the method A»m

[3] http://smalltalkhub.com/#!/~EzequielLamonica/NewChangeSystem

[4] http://wiki.squeak.org/squeak/6001

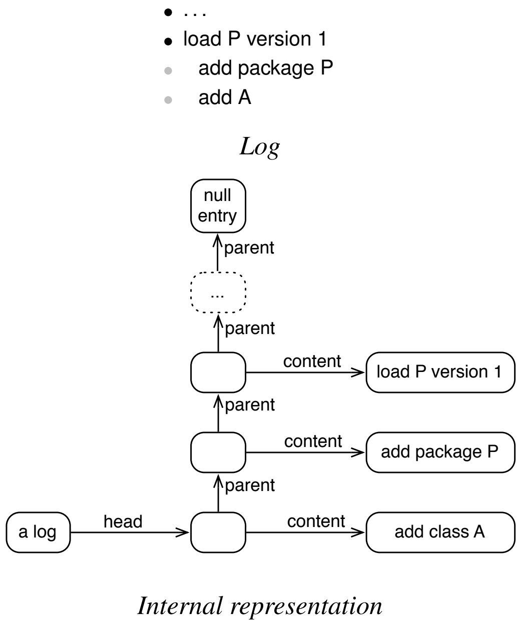


Figure 9. Object diagram of an Epicea log.

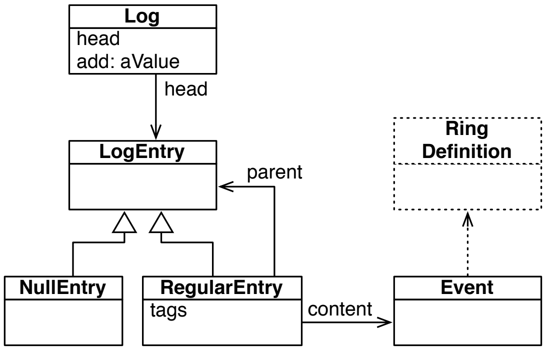


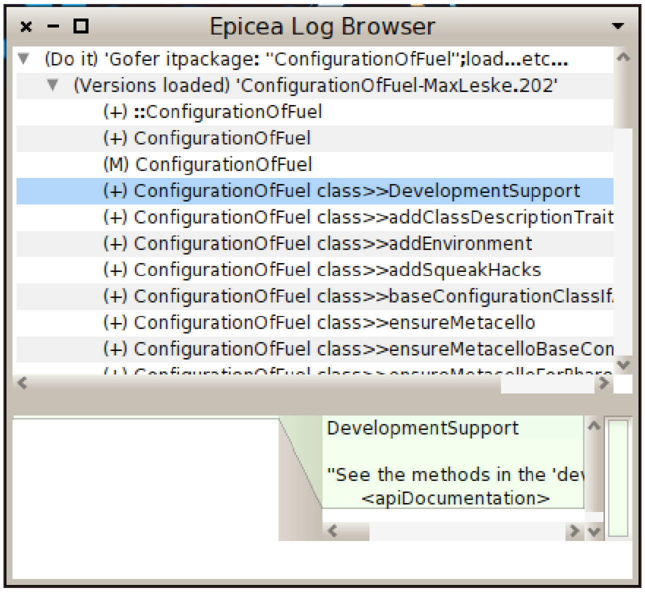
Figure 10. Design of Epicea logs.

triggered the removal of such method. In Figure 13 we show a class rename refactoring as it is logged by Epicea.

7. Related Work

SpyWare [5] captures and stores the code changes in a centralised repository in a extremely fine granularity. SpyWare records detailed changes such as a line added in a method, as well as more high-level changes like refactorings. The authors aim at post-mortem comprehension of developer work, while we focused on helping developers for their day-to-day work.

CoExist [6] is a Squeak/Smalltalk extension that preserves intermediate development states and provides immediate access to source code and run-time information of previous development states. CoExist allows for back-in time easily, automatic forks, inter-branch operations (such as re-

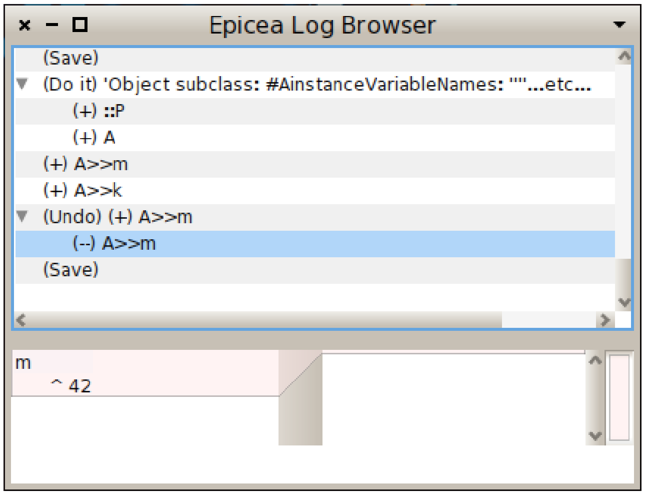


```
▼ (Do it) 'Gofer itpackage: ''ConfigurationOfFuel'';load...etc...
  ▼ (Versions loaded) 'ConfigurationOfFuel-MaxLeske.202'
      (+) ::ConfigurationOfFuel
      (+) ConfigurationOfFuel
      (M) ConfigurationOfFuel
      (+) ConfigurationOfFuel class>>DevelopmentSupport
      (+) ConfigurationOfFuel class>>addClassDescriptionTrait
      (+) ConfigurationOfFuel class>>addEnvironment
      (+) ConfigurationOfFuel class>>addSqueakHacks
      (+) ConfigurationOfFuel class>>baseConfigurationClassIf.
      (+) ConfigurationOfFuel class>>ensureMetacello
      (+) ConfigurationOfFuel class>>ensureMetacelloBaseCon
```

```
DevelopmentSupport

"See the methods in the 'dev
   <apiDocumentation>
```

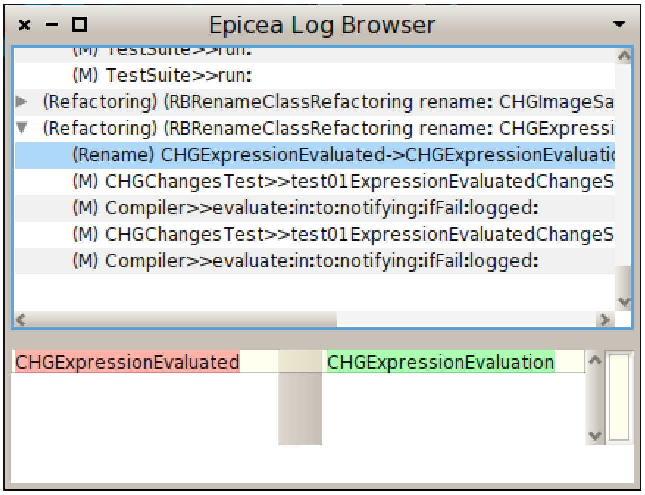
Figure 11. Epicea log browser screenshot: an expression was evaluated. It triggered the load of the package named ConfigurationOfFuel. In turn, the load triggered many elemental code changes (package, class and method additions).



```
  (Save)
▼ (Do it) 'Object subclass: #AinstanceVariableNames: ''''...etc...
     (+) ::P
     (+) A
  (+) A>>m
  (+) A>>k
▼ (Undo) (+) A>>m
     (--) A>>m
  (Save)
```

```
m
  ^ 42
```

Figure 12. Epicea log browser screenshot: Undo.



```
(M) TestSuite>>run:
(M) TestSuite>>run:
▶ (Refactoring) (RBRenameClassRefactoring rename: CHGImageSa
▼ (Refactoring) (RBRenameClassRefactoring rename: CHGExpressi
    (Rename) CHGExpressionEvaluated->CHGExpressionEvaluati
    (M) CHGChangesTest>>test01ExpressionEvaluatedChangeS
    (M) Compiler>>evaluate:in:to:notifying:ifFail:logged:
    (M) CHGChangesTest>>test01ExpressionEvaluatedChangeS
    (M) Compiler>>evaluate:in:to:notifying:ifFail:logged:

CHGExpressionEvaluated          CHGExpressionEvaluation
```

Figure 13. Epicea log browser screenshot: Class rename refactoring.

base and cherry-pick). However, the authors do not talk of a persistence mechanism for the captured code changes. CoExist is not meant to be used to share code between images and projects. Still, CoExist is a source of inspiration for the Epicea model.

JET [7] allows analysing the dependencies between VCS versions. The authors extend the Ring meta-model [8] to perform the computations. Epicea uses Ring as well. It would be interesting to apply JET dependency analysis to logs to get fine-grained results.

8. Conclusion

Modern tools for sharing code lose extra information from IDE. We want to work on a new generation of tools that use such information to help understanding the intention behind code changes. In this paper we have presented our initial steps working in this direction. We have first described a series of scenarios that help discovering main requirements of our approach. Then, we have analyzed the problems found in current Smalltalk systems, focusing on the case of Change-Sets. Finally, we have presented our early prototype with an overview of the design, as well as some screenshots that show it in action.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council.

References

[1] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[2] Peter Ebraert. First-class change objects for feature-oriented programming. In *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE'08, pages 319–322. IEEE Computer Society, 2008.

[3] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999. ordered but not received.

[4] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[5] Romain Robbes and Michele Lanza. SpyWare: a change-aware development toolset. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 847–850, New York, NY, USA, 2008. ACM.

[6] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. CoExist: Overcoming aversion to change - preserving immediate access to source code and run-time information of previous development states. In *DLS'12: Proceedings of the 8th Dynamic Languages Symposium*, DLS '12, pages 107–118, New York, NY, USA, 2012. ACM.

[7] Verónica Uquillas Gómez. *Supporting Integration Activities in Object-Oriented Applications*. PhD thesis, Vrije Universiteit Brussel - Belgium & Université Lille 1 - France, October 2012.

[8] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools. *Journal of Computer Languages, Systems and Structures*, 38(1):44–60, April 2012.