Replaying Past Changes in Multi-developer Projects

Lile Hattori, Mircea Lungu and Michele Lanza
REVEAL @ Faculty of Informatics, University of Lugano - Switzerland

ABSTRACT

What was I working on before the weekend? and *What were the members of the my team working on during the last week?* are common questions that are frequently asked by a developer. They can be answered if one keeps track of who changes what in the source code. In this work, we present Replay, a tool that allows one to replay past changes as they happened at a fine-grained level, where a developer can watch what she has done or understand what her colleagues have done in past development sessions. With this tool, developers are able to not only understand what sequence of changes brought the system to a certain state (*e.g.,* the introduction of a defect), but also deduce reasons for why her colleagues performed those changes. One of the applications of such a tool is also discovering the changes that broke the code of a developer.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering*; D.2.9 [**Software Engineering**]: Management—*Programming teams*

Keywords

Fine-grained changes, program comprehension, software evolution

1. INTRODUCTION

Software development is typically a collaborative endeavor in which teams of developers work together to achieve a common result [6]. A number of development processes [3, 25], tools [9, 19], and communication strategies (*e.g.,* e-mails, instant messaging, etc.) have been adopted to support parallel development and team coordination.

Problems related to parallel development often cause project delays. Damian *et al.* [4] report that delays of delivery are caused by unexpected changes on the system due to communication breakdowns. In addition, developers often face problems, such as tracking the introduction of a defect or resolving conflicts, activities that have little or ineffective tool support, consequently encouraging developers to find workarounds. For example, Grinter [13] and de Souza *et al.* [6] report evidence that developers tend to rush to commit their changes, and even commit partial changes to avoid dealing with merging. This behavior can be a cause of low quality and defective code, which becomes problematic during later phases of the development.

In such situations developers have to track back the changes done to the system and understand the reason why these changes were performed. Unfortunately there is little tool support for navigating through past changes and understanding how and why they were performed. Software configuration management (SCM) systems offer text-based differencing algorithms [26] to detect changes from one version of a file to another. Researchers have proposed enhanced and domain-specific [1, 10, 17] algorithms, which detect changes more precisely, but the techniques are not able to recover the order in which the changes were performed.

A study conducted by Parnin and DeLine [27], in which they evaluate cues for resuming interrupted programming tasks, indicates that developers strongly prefer the cues that present the activities chronologically to the ones that summarize and aggregate them. This finding emphasizes the importance of preserving the chronological order in which changes to the system are performed.

In previous work [14] we presented an approach to record fine-grained changes in multi-developer projects by continuously tracking edits performed by developers in the IDE. Syde, our tool [15] that implements the approach, considers development as a continuous activity, thus recording all the steps that take the system from one state to another, as opposed to mainstream SCM systems, where developers commit changes at their own will and the evolution of the system is seen as a sequence of snapshots [29].

In this work we use the information our approach records in order to help developers understand past changes. We build a tool, called Replay, where developers can choose to watch changes made by a set of developers on a set of artifacts during a certain period of time, while preserving the order in which they were performed. In previous work Robbes and Lanza [31] argue that the sequence of changes recorded during the system's development can lead to valuable insights about the construction of the system, which could be useful for program comprehension (who changed which software artifacts and when?), reverse engineering (how did the architecture of a system evolve?), understand the phenomenon of evolution itself (how do programmers write code? how long are programming sessions? are there different types of sessions? are there development patterns that a developer follows?).

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWPSE-EVOL '10, September 20-21, 2010 Antwerp, Belgium
Copyright 2010 ACM 978-1-4503-0128-2/10/09 ...$10.00

We go a step further, replaying past changes of multi-developer projects, and argue that such changes are also useful for team collaboration (why is my teammate editing my class? why are my tests failing today if they were passing yesterday and I did not change anything? what change performed by whom and for which reason introduced this defect?). Such questions can be seen as programming comprehension questions, which are not only conceptually important, but also have intrinsic monetary value: The tool VE, a version editor for vi that integrates the change history into the editor, turned out to not only to generate an increase in productivity [2], but also produced an "estimated saving of approximately 270 million dollars over the ten year period"[1] that it was in use.

Structure of the paper. In Section 2 we present our approach to record fine-grained changes on multi-developer projects. In Section 3 we present the Replay tool. In Section 4 we demonstrate, through a case study, how developers can benefit from such a tool. In Section 5 we discuss the findings of our case study and address the next steps of this work. In Section 6 we present related work and conclude in Section 7.

2. CHANGE-CENTRIC EVOLUTION

Our goal is to record the evolution of a multi-developer system at a fine-grained level. To achieve that, we extend Robbes's CBSE (change-based software evolution) model [30], which treats changes as first-class entities with the aim of accurately modeling how software evolves. We extend the model to add support for multiple developers by modeling the evolution of a system as a set containing sequences of changes, where each sequence is produced by one developer. A sequence of changes takes a developer's copy of the system from one state to the next by means of syntactic operations. These operations are captured from the Eclipse IDE every time a developer modifies his copy of the system. Syde, our tool, is a client-server application that captures changes on the IDE at every compilation and stores it on the server. Thus, the evolution of a system is the combination of the sequences of changes produced by each individual.

System Representation. To record information about syntactic changes, we focus on object-oriented systems in Java. Therefore, we store and analyze constructs such as classes and methods, instead of files and lines. To facilitate this, a software system is modeled as an abstract syntax tree (AST) containing nodes, which represent packages, compilation units, classes, methods, and fields. Nodes have properties, which vary depending on their type. A node representing a class can have a super class and a set of interfaces; a compilation unit can have a set of imports; *etc.* Syde provides a unique identifier for each entity, and tracks name changes and entity moves. Syde stores on the server one AST per developer, which reflects the exact state of the system in a developer's workspace.

A compilation unit is not an object-oriented entity, but serves as the representation of a Java file that always contains at least one class. We model it as a tree node in the AST, as it contains information, such as import declarations, that would be lost if it was ignored.

In a multi-developer project the current state of a system is different for each developer, as it depends on the changes each has performed after a checkout. We represent the current state of the system per developer by keeping on the server one AST per developer.

[1]See `http://ix.cs.uoregon.edu/~datkins/ve.html`

Change Operations. In CBSE, change operations represent the evolution of the system, instead of file versions [30]. A change operation is the representation of a change a developer performs in his workspace, *i.e.,* it is the transition of a system from one state to the next. Syde captures both atomic changes and composite change operations (*e.g.,* refactorings [11]). Examples of both types of changes can be found in Table 1.

Atomic Operations	
Insertion	Insert a node n as a child of parent p
Deletion	Delete a node n from its parent p
Property Change	Change the value v of property r of node n
Composite Operations	
Rename	Change the name of a node n and change all references of this node from the old name to the new one
Move	Move a node n and all its descedants from old parent p_{old} to new parent p_{new}

Table 1: Change Operations supported by Syde

Atomic changes are the finest-grained operations on a system's AST, and contain all the necessary information to update the model. By applying a list of atomic changes in the order they were received on the server, it is possible to generate all the states of the program during its evolution. Although atomic change operations reflect the entire evolution of the system, they can lead to an overwhelming amount of information. In addition, some refactorings that a developer performs are automated by Eclipse and lose their meaning when seen as separated atomic changes. Currently, we capture two refactorings as composite operations: rename and move.

System Architecture. Syde is a client-server application, where the server records the change operations, maintains the current state of a project and publishes information about the activity of the team. The client is a collection of plug-ins that enriches the Eclipse IDE to track changes and to show awareness information to developers. Figure 1 shows the architecture of Syde.

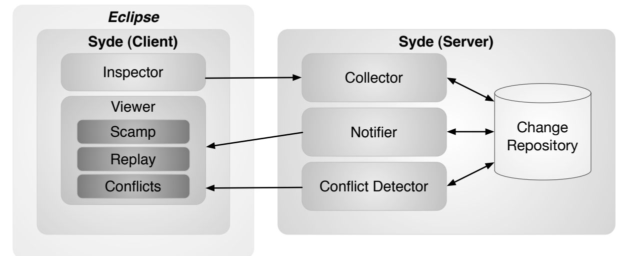


Figure 1: Syde architecture

Syde features the following components:

Inspector. It is the core plug-in, responsible for inspecting the changes on the IDE, translating them to change operations, and sending them to the server. The inspector also provides an API for gathering awareness information from the server, facilitating the creation of plug-in extensions.

Collector. This module receives the change operations from the Inspector, saves them to the repository, and keeps in memory the state of the system in each developer's workspace.

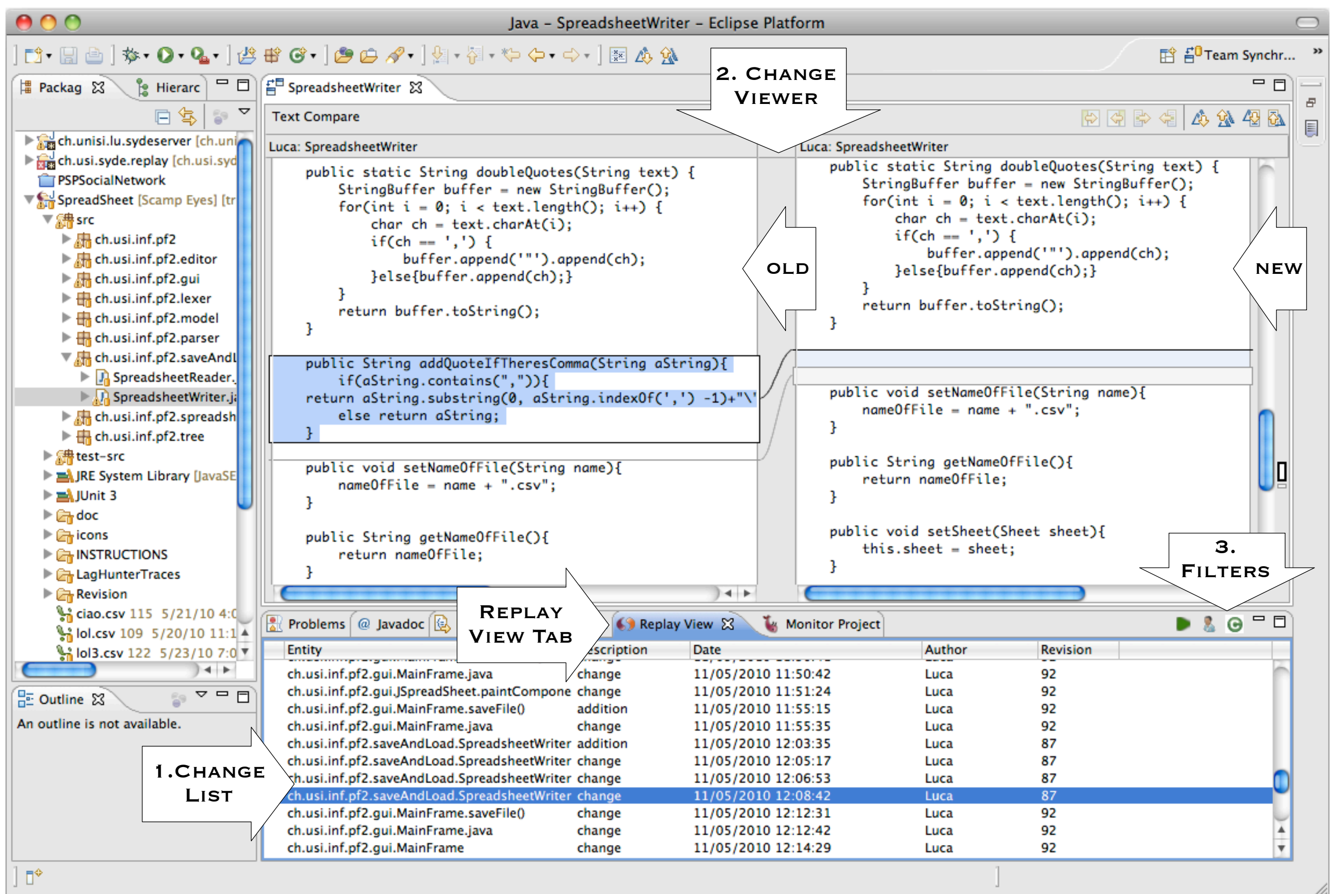


Figure 2: The user interface of Replay

Notifier. It is responsible for broadcasting awareness information of ongoing changes. Every time a new change arrives at the Collector, it communicates with the Notifier, who immediately broadcasts this information to other interested clients.

Conflict Detector. This module searches for potential conflicts that may have arisen from ongoing changes on the developersÕ workspaces. The conflict detector algorithm takes as input the ASTs kept by the Collector and the recent operation to search for new conflicts and to update existing ones.

Viewer. It is a collection of Eclipse plug-ins that uses the change information available on the Syde server to visually show the current activity of the team. It is composed of three plug-ins: Scamp – shows who is changing which parts of the system; Conflicts – shows potential merge conflicts that may arise when two developers change related parts of the code (for more information on these plug-ins, refer to [15]); Replay – allows developers to replay past changes in the order that they happened.

3. REPLAY

Syde collects atomic changes, which is what Replay uses. However, showing every atomic change individually would overwhelm the developer with too fine-grained information. Hence, we group these changes by timestamp, developer and artifacts, where artifacts are package or compilation unit. This means that all the changes that were performed by a developer, for instance, in a compilation unit within a small period of time will be grouped together. Within a group there can not be more than one change on one entity, thus we maintain the fine-grained granularity of the changes.

Visualizing Changes. Figure 2 presents the main view of the Replay plug-in, with the following components:

1. **The Change List.** The list at the bottom of the view contains all the changes that are loaded at a given moment and are ready to be viewed. The Change List shows the change groups sorted chronologically. The name of the artifact shown is one of the lowest level nodes on the AST. Hence, if there is more than one method on a group, only the information of one of this methods is displayed. By navigating through the change groups and by following the details of each change in the Change Viewer one will eventually replay all the activity in the system.

   In Figure 2 the change that is being visualized is the removal of the `addQuoteIfTheresComma` method from the `SpreadsheetWriter` class. The Change List shows the context of the selected change. It shows that the developer Luca has been making other three changes to this compilation unit, and that his next change was to the MainFrame class.

For each change the list provides the following information:

- The artifact that was changed
- The type of change
- The date and time of the change
- The developer who performed the change
- The SVN revision that was the baseline for the change

2. **The Change Viewer.** When a change is selected in the change list, the user can see what was the effect of that change on the system in the upper part of the view displayed with a classical diff viewer[2] in which the left part presents the artifact before the change, and the right part presents the artifact after the change.

3. **The Filters Toolbar.** The toolbar allows the selection of a subset of all the changes.

Filtering Changes. There are three orthogonal categories of filters that can be applied to the changes that represent the underlying model of the plugin:

1. **Time-based Filters.** Their goal is to filter the changes based on the time period when they were performed. The time period can be specified as a combination of begin and end time.

2. **Artifact-based Filters.** Their goal is to focus the replay on only a subset of the artifacts in the system. The artifacts can be methods, classes, compilation units, or packages.

3. **Developer-based Filters.** Their goal is to focus the change replay on the activity of a subset of the developers in the system. Such a subset can be a team of developers, or an individual developer.

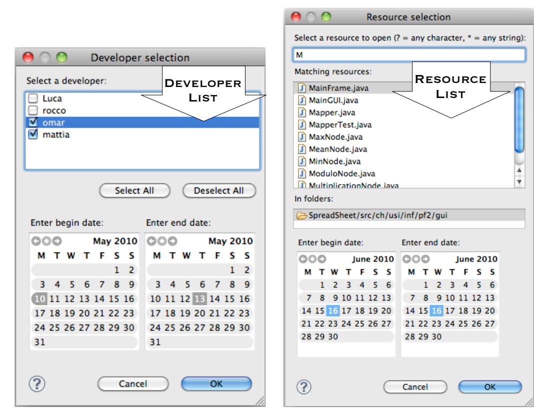
Combining various filters can support different activities:

- **Remembering previous work.** A developer needs to remember what was he working on in a previous session before he can move on and continue the work. One way of doing this is by replaying the changes he has already performed in chronological order. This can be done by combining a time-based filter with a developer-based filter.

- **Understanding changes made to a given resource.** When the goal is understanding a given resource it is useful to replay only the changes that happened to that resource, using an artifact-based filter combined with a time-based filter.

- **Discovering the origin and reason for the existence of defects.** This can be done, using a time-based filter, by replaying and analyzing all the changes performed on the system since a version known to work.

- **Understanding changes made by a given team.** This can be obtained by a developer-based filter that filters out all the changes that are not performed by the subject team.



(a) Developer selection (b) Resource selection

Figure 3: Replay Dialogs

Figure 3 presents two dialogs that support the first two activities.

Replaying the Activity of a Developer. Figure 3(a) presents one of the filter dialogs in the Replay. The user can select one or more developers to see their changes, and select a time interval for further filtering the changes to the system.

Replaying the History of a Resource. Figure 3(b) presents the resource selection dialog. The user can search through the artifacts of the system and select one. Then she can select a time interval for further filtering the changes to the system.

4. CASE STUDY

Our goal is to investigate how Replay can help developers to answer questions they have during a software development project. Previous studies have provided catalogs of such questions, from which two focus on questions related to source code [33, 5], and three explore questions related to development activities [23, 20, 35]. For our case study, we concentrate on answering a subset of the questions (shown in Table 2) raised by Fritz and Murphy [35], which cover a broad set of development activities.

People related	
Q1	What have people working on?
Q2	How much work have people done?
Q3	Who changed this code?
Code related	
Q4	What is the evolution of the code?
Q5	Who made a particular change and why?
Q6	What classes has my team been working on?
Q7	Who made changes to my class?
Q8	Who owns this piece of code?
Q9	Who to talk to if you have to work with packages you haven't worked with?
Q10	What classes have been changed?
Q11	What code is related to a change?
Mixed questions	
Q12	Who owns a test case?
Q13	Who has made a change that introduced a defect?

Table 2: A selection of questions developers ask from [35].

[2] We use the CompareUI already implemented in the Eclipse IDE, leveraging the familiarity of the user with the interface.

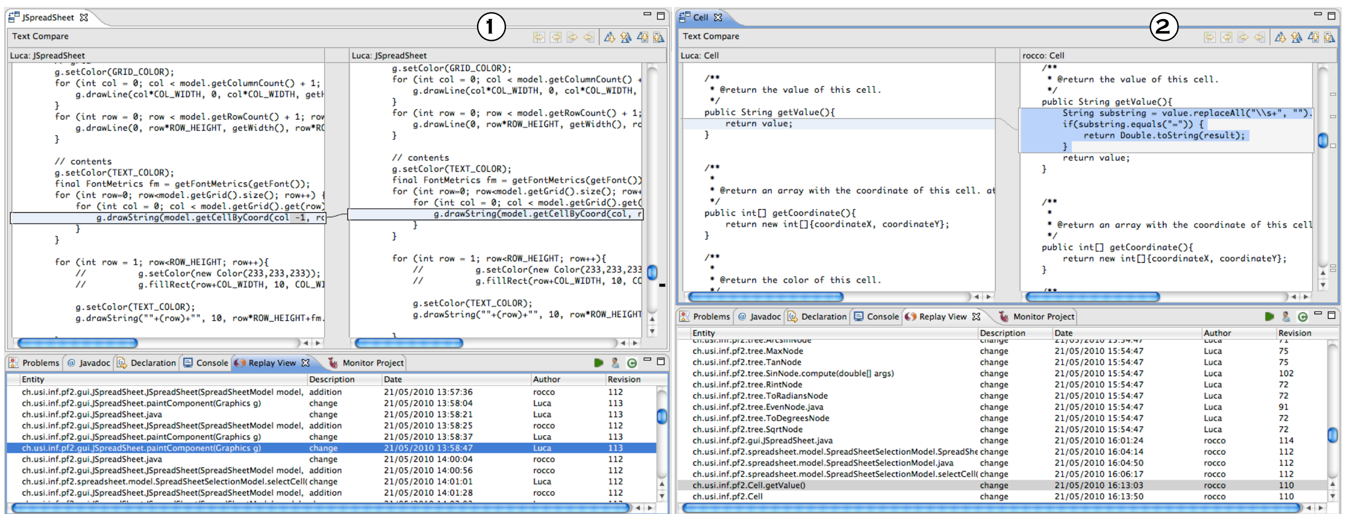


Figure 4: Looking at what team members have been working on

To investigate the above questions we use a project that has been developed in the context of the *Programming Fundamentals 2* course at the University of Lugano. In this project, the students had to create a spreadsheet application, with support for basic mathematic formulas, saving and loading files, and optional functionalities (*e.g.,* support for drawing graphs). The project was developed by four students, lasting approximately 6 weeks. At the end, the project counted at 13 packages, 77 classes, 286 methods, for a total of 1,882 lines of Java code. The number of SVN commits was 137, while the number of changes Syde recorded was 11,661.

In the following, we investigate how to answer the questions listed in Table 2 by proposing tasks that involve at least one of the questions, and walking through screenshots of Replay describing how to accomplish the task.

4.1 People Related Questions

In previous work [22] we have investigated questions related to activities occurring in the present, such as "Who is working on what?". Here we focus on the questions related to activities occurring in the past.

Task 1: *Find out what Luca and Rocco were doing yesterday (Figure 4).* To accomplish that, on the "Developer selection" dialog, we select Luca and Rocco, and choose the time interval of 1 day.

In the first screenshot we see that around 2:00 PM both Luca and Rocco were working on class `JSpreadSheet`, however Luca was editing a newer version of the class. We notice that Luca was working on the `paintComponent` method, while Rocco was mainly editing the constructor of this class. The second screenshot shows that, two hours later, these developers were working on different parts of the system. Luca was now refactoring the math operations (*e.g.,* `TanNode, SinNode`), while Rocco was changing classes `SpreadSheetSelectionModel` and `Cell`.

By selecting the target developers and the period to be observed, we are able to investigate Q1. The Change list shows us a complete list of all the changes Luca and Rocco have done, showing where (up to method level) and when they changed, and what was the type of the change. By navigating through the items on the list, it is possible to see the changes at the code level, where the item selected is shown on the right-hand side of the Change viewer, and the change made immediately before on the same item is shown on the left-hand side. The number of items on the list associated with each developer give a glimpse on how much they have worked on each source code artifact, addressing Q2 indirectly. Q3 is similar to Q5, and will be addressed in the next section.

This task can be rephrased as "understanding the changes made by a given group of people". It can also be used by a developer to remember what she has done on the previous day, week or month. One could argue that using differencing algorithms on SCM versions also answer these questions. The fundamental difference between our approach and mainstream SCM systems is the level of detail: While SCM screenshots would show where in the code there were changes, Replay also shows in which order they were performed.

4.2 Code Specific Questions

Theses questions are related to source code changes, where the focus in on what happened in the code, instead of what were people doing.

Task 2: *Investigate what has changed in class `Cell` for the past 4 days (Figure 4).*

Given the granularity of changes that Syde captures, we can say that there were few changes on `Cell` during this period, although all developers edited it. From the Change List we see that a couple of fields (`sheet`, and `coordinateX`) and methods (`setValue`, `getValue`, `changeCoordinates`, `updateValueType`, *etc.*) have changed. By navigating through the list, we see the changes at source code level. In the first change we can see that Luca added the method `changeCoordinates`. One hour later, Omar change method `updateValue` followed by the change of method `getValue` by Rocco. The fourth change reveals duplicated work between Omar and Luca, since both added the same method within 2 hours of difference.

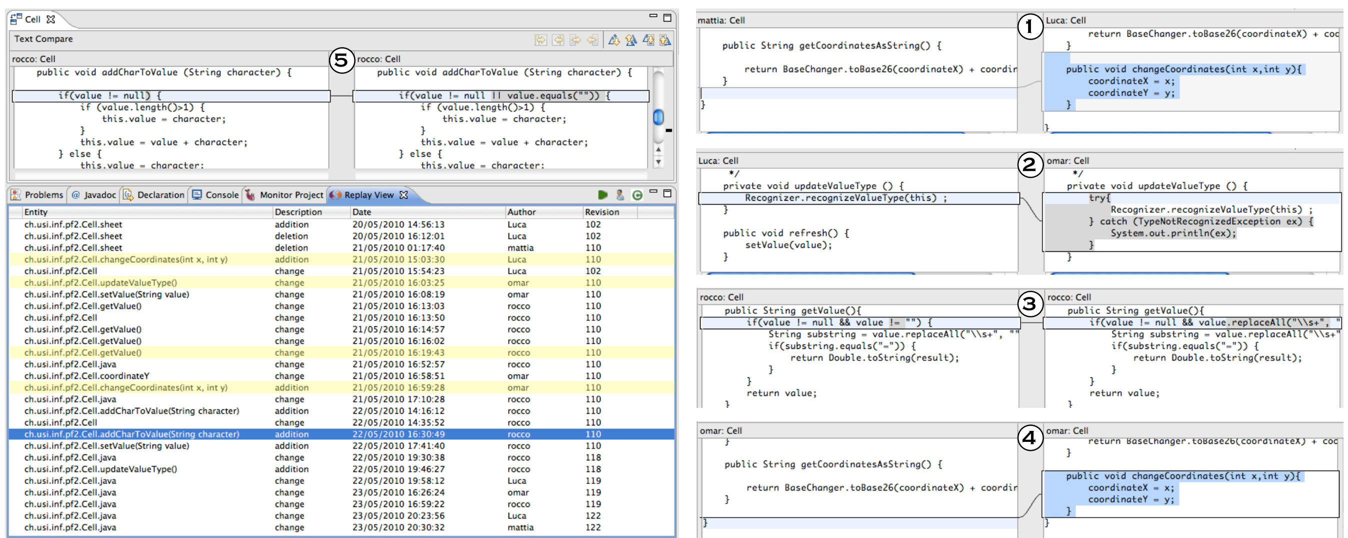


Figure 5: Looking at the history of changes on a specific artifact

With this task we tackle Q4 by showing how the code evolved and who was responsible for each stage of its evolutions. As consequence, Q3 is also answered, since we can see who changed the code up to method level. To answer Q5 with Replay, we need to make the following assumptions: the developer knows to which artifact the change was made, and he has a glimpse of when this was made. Then, she can watch the changes made to the artifact, find out who was responsible for it, and investigate the reason for performing them by watching what the other developer was doing before change that specific artifact.

Task 2 also gives straightforward answers to Q7, Q8, and Q9. Regardless if someone owns a software artifact or wants to know who has knowledge about it, she will filter the changes by this artifact to look for the answer to her question. Q6 and Q10 were already answered by task 1, where one can select a group of developers and watch what they did. Q11 is answered by navigating through the Change List and observing the source code changes that are shown on the Change Viewer.

4.3 Mixed Questions

During the software development process, there are mixed questions developers ask that are related to both people and code. Questions related to broken builds, defects, tests, team and task organization could benefit from Replay.

Task 3: *Bob is a new developer, and to start working on this project, he was assigned to fix a defect. The defect happens when loading a spreadsheet smaller than the default size (Figure 6).*

Bob starts his task by trying to replicate the bug, which he successfully does (Figure 6.1). After trying for a couple of times, he realizes that the index given by the variable `selectedColumn` is returning a number that is double the size of the array, causing the exception (in Figure 6 the exception happens at the line that is highlighted in the screenshot marked as 1). However, he does not understand why. He decides to use Replay to find out who is responsible for the `MainFrame` class to ask for help, discovering that Luca -who has left the team- is the most knowledgeable about it. Since Luca is not there to answer questions, Bob decides to replay the changes to the MainFrame class to understand how the code evolved to the defective state. First, he localizes when the method `addColumn` was added and who was responsible for it – Luca (Figure 6.2). Then, he follows the history to find when the `selectedColumn` variable was added (Figure 6.3). By looking at the previous state of the method he observes that the code looked correct before the offending line was added by the developer Luca. He tries to revert to the previous state, runs the program, and discovers that the defect is gone. After running a few tests Bob decides that he is satisfied with the fix.

Bob was able to answer Q13 by first searching for developers who changed a software artifact, and then narrowing down to the cause of the defect. The same sequence of actions taken to perform this task could have been used to solve question Q12. Tracking the origin of a defect is not essential for fixing the defect, but it has the potential to speed up the resolution process. In addition, situations such as a developer leaving a team are common, and even if the developer is still on the team, she might not remember the changes that she has done in the past. In addition, a defect might reveal itself long after it was introduced, and tracking it back with information provided by SCM systems is troublesome. We argue that Replay can bring valuable help in these situations by letting developers explore the change history of the system in a chronological order, and thus speeding up the resolution of the problem.

5. DISCUSSION

We have provided evidence that Replay can be useful for helping developers in various activities related to software development and program understanding. Constructing a software system is a continuous and collaborative activity, in which teams of developers work together to bring the system to a functioning state. The history of the changes that brought the system from one state to another is important both during development and maintenance phases, because it help developers to understand the reasons for a system to be at a certain state. It can also help them to understand how to revert or circumvent undesired behaviors based on decision taken in the past.

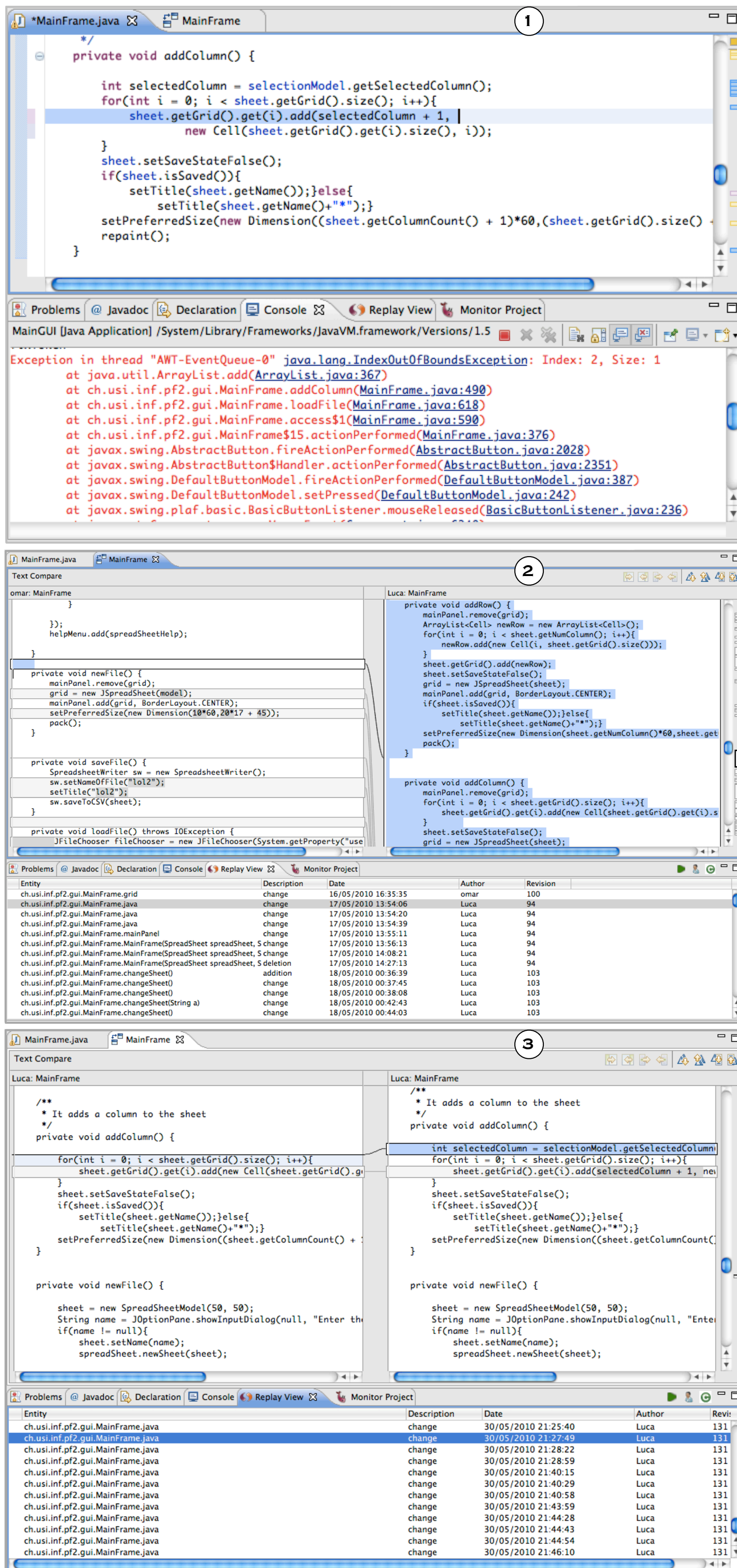


Figure 6: Tracking changes that might have originated a defect

With Replay, we make it possible for developers to watch past changes as they happened at source code level. To our knowledge, the only work that is directly related to ours, is Robbes's [31], in which changes are also recorded at fine-grained level, and metrics are used to characterize development sessions. The fundamental difference between our approach and Robbes's is that he takes a top-down approach, where development sessions are first visualized as "spark lines", then the developer can focus on a specific part of the development session to finally see the change at source code level. Our approach is source code driven, where navigating through the Change List, the changes at source code level will be immediately shown on the Change Viewer.

The filters provided by Replay (developer, artifact, and time) facilitate answering different questions, as illustrated on the three tasks we discussed. Replay is in initial prototyping stage and still needs improvements, such as allowing the combination of the three filters. Currently, it only allows the combination of developer and time, or artifact and time. We plan to enhance its filters and the visual presentation of changes to run a user study in which we can measure whether and how much Replay can help developer to answer program comprehension questions. To accomplish that, we plan to ask subjects to answer some of the questions cataloged in previous work [33, 5, 23, 20, 35], and compare their performances using Replay against using commonly adopted differencing algorithms for SCM revisions.

6. RELATED WORK

To our knowledge, we are the first to support replaying development sessions in a collaborative environment and there is no direct work that is related to ours. However, if we look at our work in the broader context of software development and reverse engineering, there are various lines of research that we took inspiration from. Due to the preliminary nature of this work our intention is not to fully cover the related work, but only briefly survey the work that bears some similarity to ours.

The first line of research that is related to ours is software evolution analysis. Various approaches make use of the changes performed to a system over its lifetime to support its understanding: Lanza summarizes the evolution of classes [21]; Gîrba summarizes the evolution of class hierarchies [12]; Lungu summarizes the evolution of inter-module relationships [24]. The main difference is that in all these works, and other similar ones, the history is not replayed, but rather summarized and the order in which the changes are performed is lost. In our work we specifically focus on replaying the change events in the order in which they happened.

There are a few works that focus on replaying the changes that happened in a system. Wettel visualizes the evolution of the entire system by allowing the user to *travel in time* and observe the changes of the system as they are represented in a 3D city metaphor [36]. In the Yarn tool, Hindle *et al.* present an animation of the evolution of the architecture of a system [16]. The animation presents the evolution of the relationships between the modules in the system. Both these systems allow the animation of the changes. However, they present the changes at high level of abstraction, from which the developer can not use to understand the code, but a level that is useful for other tasks. We believe that program comprehension needs reading the code, and can be eased by replaying the code in the same sequence in which it was written. The tool VE goes to a low level of abstraction, embedding information of changes submitted to version control systems into the source code editor [2].

One great difference between our work and all the work mentioned already is the level of detail of the data. In most of the approaches the data is extracted from SVN, which means, the changes between versions can be arbitrarily complex. An approach that uses fine-grained change information is the one of Robbes [28]. Although he collects fine-grained information from software systems, Robbes does not use it to support the replaying of the changes but instead he uses it to detect and characterize development sessions [31]. Dig, instead, uses a change-centric approach to record sequences of refactorings and to replay them on other library-based applications [7].

A final research direction that is related to ours are the IDE plug-ins that recommend related artifacts. NavTracks is an Eclipse plug-in introduced by Singer *et al.* [34] which captures the navigation patters of a developer working in Eclipse. Although theoretically they could replay the navigation, they do not do it; and even if they did, since the navigation is stored locally, replaying other developers' changes would not be possible. A similar work is Mylyn, a tool that uses interaction data to maintain task context [18]. Mylyn presents the same limitations as NavTracks does: it discards the history of the interactions with the IDE and works only locally.

7. CONCLUSION AND FUTURE WORK

We have presented an approach for replaying past changes at a fine-grained level for multi-developer projects. Traditionally, the evolution of the source code of a system is recorded using SCM systems. Although development is a continuous activity, SCM repositories only store snapshots of the system, which are committed by developers at arbitrary time intervals. Therefore, source code changes are represented as chunks of text changes with no particular ordering. There has been little noteworthy effort to provide operation-based SCM systems [8, 32], which are able to conserve the chronological order of changes and add semantics to them. We argue that the chronological order and the granularity of changes is important for understanding how a system evolves from one state to another.

Our proof-of-concept tool, Replay, allows developers to investigate how changes were performed in the past by filtering them according to three criteria: developer, artifact, and time. The combination of the filtering criteria facilitates replaying subsets of the changes that support answering different program comprehension questions. To test the usefulness of Replay we used it in a case study in which we set to answer a series of questions that have been presented in a recent study of software development practice [35]. Our case study provides evidence that Replay can help developers to speed up the process of answering such questions.

The preliminary evidence of the usefulness of Replay we provided in this article is certainly only anecdotal, and there is the need for a thorough evaluation, through a controlled experiment, to ascertain which questions Replay effectively helps to answer.

We plan to enhance Replay with new features to improve the comprehension process before performing the evaluation. First, we want to allow the combination of the three filters, which is not possible in the current version. Then, we want to replace the Change Viewer to one that makes better use of visual cues to show what has changed between subsequent changes. Some of the features that we think would be useful are timelines for navigation and colors associated with change types. Overall, our goal is take the necessary steps to make the Replay tool mature enough for evaluating it and making it available to the public.

Acknowledgments. We acknowledge the financial support of the Swiss National Science foundation for the project "GSync" (SNF Project No. 129496).

8. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[2] D. Atkins, T. Ball, T. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: A case study of the version editor. *IEEE Transactions on Software Engineering*, 28(5):625–637, May 2002.

[3] M. B. Chrissis, M. Konrad, and S. Shrum. *CMMI Guidlines for Process Integration and Product Improvement*. Addison-Wesley, 2003.

[4] D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Proceedings of the ICGSE 2007 (International Conference on Global Software Engineering)*, pages 81–90. IEEE Computer Society, 2007.

[5] B. de Alwis and G. C. Murphy. Answering conceptual queries with ferret. In *Proceedings of ICSE 2008 (30th International Conference on Software Engineering)*, pages 21–30. ACM Press, 2008.

[6] C. R. B. de Souza, D. Redmiles, and P. Dourish. Breaking the code, moving between private and public work in collaborative software development. In *Proceedings of GROUP 2003 (International ACM SIGGROUP Conference on Supporting Group Work)*, pages 105–114. ACM Press, 2003.

[7] D. Dig. *Automated Upgrading of Component-based Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[8] D. Dig, K. Manzoor, R. Johnson, and T. N.Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proceedings of ICSE 2007 (29th International Conference on Software Engineering)*, pages 427–436. IEEE Computer Society, 2007.

[9] J. Estublier. The adele configuration manager. In *Configuration management*, pages 99–133. John Wiley & Sons, 1995.

[10] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering (TSE)*, 33(11):725–743, 2007.

[11] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[12] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering)*, pages 2–11. IEEE CS Press, 2005.

[13] R. Grinter. Supporting articulation work using software configuration management systems. *Computer Supported Cooperative Work*, 5(4):447–465, 1996.

[14] L. Hattori. Enhancing collaboration of multi-developer projects with synchronous changes. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering), Doctoral Symposium*, pages 377–380. IEEE CS Press, 2010.

[15] L. Hattori and M. Lanza. Syde: A tool for collaborative software development. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 235–238, 2010.

[16] A. Hindle, Z. M. Jiang, W. Koleilat, M. W. Godfrey, and R. C. Holt. Yarn: Animating software evolution. In *Proceedings of VISSOFT 2007 (4th International Workshop on Visualizing Software for Understanding and Analysis)*, pages 129–136. IEEE CS Press, 2007.

[17] J. Hunt and W. Tichy. Extensible language-aware merging. In *Proceedings of ICSM 2002 (18th International Conference on Software Maintenance)*, pages 511–520. IEEE CS Press, 2002.

[18] M. Kersten and G. Murphy. Using task context to improve programmer productivity. In *Proceedings of FSE 2006 (16th SIGSOFT Symposium on the Foundations of Software Engineering)*, pages 1–11. ACM Press, 2006.

[19] J. Keyes. *Software Configuration Management*. Auerbach Publications, 2004.

[20] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of ICSE 2007 (29th ACM/IEEE International Conference on Software Engineering)*, pages 344–353. IEEE Computer Society, 2007.

[21] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (4th International Workshop on Principles of Software Evolution)*, pages 37–42. ACM Press, 2001.

[22] M. Lanza, L. Hattori, and A. Guzzi. Supporting collaboration awareness with real-time visualization of development activity. In *Proceedings of CSMR 2010 (14th IEEE European Conference on Software Maintenance and Reengineering)*, pages 207–216. IEEE CS Press, 2010.

[23] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE 2006 (28th ACM International Conference on Software Engineering)*, pages 492–501. ACM, 2006.

[24] M. Lungu and M. Lanza. Exploring inter-module relationships in evolving software systems. In *Proceedings of CSMR 2007 (11th IEEE European Conference on Software Maintenance and Reengineering)*, pages 91–100. IEEE CS Press, 2007.

[25] R. Martin. *Agile Software Development - Principles, Patterns, and Practices*. Prentice Hall, 2003.

[26] E. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, March 1986.

[27] C. Parnin and R. DeLine. Evaluating cues for resuming interrupted programming tasks. In *Proceedings of CHI 2010 (28th International Conference on Human Factors in Computing Systems)*, pages 93–102. ACM Press, 2010.

[28] R. Robbes. *Of Change and Software*. PhD thesis, University of Lugano, Switzerland, Dec. 2008.

[29] R. Robbes and M. Lanza. Versioning systems for evolution research. In *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pages 155–164. IEEE CS Press, 2005.

[30] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 166:93–109, Jan. 2007.

[31] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *Proceedings of ICPC 2007 (15th IEEE International Conference on Program*

Comprehension), pages 155–164. IEEE CS Press, 2007.

[32] R. Robbes and M. Lanza. Spyware: A change-aware development toolset. In *Proceedings of ICSE 2008 (30th ACM/IEEE International Conference in Software Engineering)*, pages 847–850. ACM Press, 2008.

[33] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proceedings of SIGSOFT 2006/FSE-14 (14th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, pages 23–34. ACM Press, 2006.

[34] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software. In *Proceedings of IWPC 2005 (13th International Workshop on Program Comprehension)*, pages 173–175. IEEE CS Press, 2005.

[35] G. C. M. Thomas Fritz, Jingwen Ou and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of ICSE 2010 (32nd ACM/IEEE International Conference on Software Engineering)*, pages 385–394. IEEE Computer Society, 2010.

[36] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Proceedings of WCRE 2008 (15th IEEE Working Conference on Reverse Engineering)*, pages 219–228. IEEE CS Press, 2008.