Information and Software Technology 55 (2013) 755–775

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Answering software evolution questions: An empirical evaluation

Lile Hattori [a,*], Marco D'Ambros [a], Michele Lanza [a], Mircea Lungu [b]

[a] REVEAL @ Faculty of Informatics, University of Lugano, Switzerland
[b] Software Composition Group, University of Berne, Switzerland

A R T I C L E I N F O

Article history:
Available online 1 October 2012

Keywords:
Software evolution
Empirical evaluation
Controlled experiment
Software change history
Mining software repositories

A B S T R A C T

Context: Developers often need to find answers to questions regarding the evolution of a system when working on its code base. While their information needs require data analysis pertaining to different repository types, the source code repository has a pivotal role for program comprehension tasks. However, the coarse-grained nature of the data stored by commit-based software configuration management systems often makes it challenging for a developer to search for an answer.
Objective: We present Replay, an Eclipse plug-in that allows developers to explore the change history of a system by capturing the changes at a finer granularity level than commits, and by replaying the past changes chronologically inside the integrated development environment, with the source code at hand.
Method: We conducted a controlled experiment to empirically assess whether Replay outperforms a baseline (SVN client in Eclipse) on helping developers to answer common questions related to software evolution.
Results: The experiment shows that Replay leads to a decrease in completion time with respect to a set of software evolution comprehension tasks.
Conclusion: We conclude that there are benefits in using Replay over the state of the practice tools for answering questions that require fine-grained change information and those related to recent changes.

1. Introduction

When evolving a code base, during software development or software maintenance, developers keep a mental model of the system—an internal working representation of the software under consideration [1]. This individual understanding of the system is constantly being updated by the developer's interactions with the code and the team, and by seeking answers to various questions [2–5]. These questions span multiple areas [6] such as program comprehension, software evolution, collaborative software development, and program analysis; therefore, they require a variety of information sources (e.g., colleagues, code bases, issue trackers, documentation, communication history), and multiple tools (e.g., [7–10]) to fulfill them.

Although there are a number of resources (data and tools) available to ease the comprehension of a system and its evolution, the amount of resources actually used by developers is often limited to talking to colleagues and exploring the code.

In an exploratory study [11], LaToza et al. report that most teams have a *team historian*, the go-to person for questions about the code; and that most team members subscribe to the check-in messages to keep themselves updated with the code evolution, though many expressed dissatisfaction with the lack of detail provided by their teammates when describing the changes in commit messages.

We argue that this *lack of detail* is a fundamental problem for understanding software evolution, i.e., the changes made by other developers. The problem is related to the coarse granularity at which changes are recorded and, consequently, seen by others. When trying to understand the evolution of the code, the delta between subsequent changes can be complex enough to prevent developers from inferring the design decision behind the changes in the code. Moreover, as indicated by previous studies [12,13], large commits can also lead to merge conflicts, duplicated work, and conflicting design decisions.

In our previous work [14,15] we presented Syde, an Eclipse plug-in that records fine-grained changes in multi-developers projects by continuously tracking code edits performed in the Integrated Development Environment (IDE).

In recent work we presented Replay [16,17], an Eclipse plug-in that allows developers to explore the rich change repository created by Syde. Developers can search for fine-grained changes made by a set of people to a set of artifacts and watch them in the chronological order as originally performed in the IDE. This counts for a better user experience [18] than the aggregated form of commit-based Software Configuration Management (SCM) tools, such as CVS and Subversion.

* Corresponding author. Tel.:+1 778 8835375.
E-mail address: lile.hattori@usi.ch (L. Hattori).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.09.001

In a previous version of this paper [17], we conducted a controlled experiment to assess whether Replay is at least as effective and efficient as the state of the practice at supporting developers with their questions related to software evolution [17]. The design of the experiment involved the selection, from previous catalogs [2–5], of a set of common questions that developers ask. We converted them into a set of tasks to measure both the correctness of the task solutions and their completion time. We conducted additional runs of the controlled experiment, involving new participants, expanding our analysis of the experiments results, and making our experiments replicable by sharing the necessary information.

The contributions of our previous paper are:

- *Replay*, a toolset to replay and exploit a fine-grained change software repository to aid developers in answering questions related to software evolution;
- a report on the design and operation of a series of controlled experiments with 45 subjects to compare the performance of Replay with the baseline tool (SVN client) in performing selected software evolution comprehension tasks;
- a quantitative analysis of the results, which shows a statistically significant advantage of Replay over the baseline in time, and indicates advantages of Replay on correctness.

The additional contributions this article makes are:

- an extended quantitative analysis of the results obtained with a larger sample size, which shows a statistically significant advantage of Replay over the baseline in time, and shows an improvement on the indication of the advantages of Replay on correctness;
- a qualitative analysis of the tool's usefulness based on the subjects' feedback, discussing the tool's current flaws, and potential improvements;
- the complete experimental data to make our experiment replicable.

1.1. Structure of the article

In Section 2 we review Syde and its change model to subsequently present Replay. In Section 3 we describe the design and operation of our controlled experiment. In Section 4 we analyze the experiment results and discuss the threats to validity. In Section 6 we present work related to the tool, and to the controlled experiment. In Section 7 we present the concluding remarks. Finally, in A we present the complete dataset that makes this experiment replicable.

2. Tool support: Syde and Replay

2.1. Syde

Syde is a client–server application that records fine-grained information about the evolution of a system developed in a multi-developer setting [14,15]. It extends Robbes' change-based software evolution (CBSE) model [19] into a multi-developer context by modeling the evolution of a system as a set containing sequences of changes, where each sequence is produced by one developer. A change takes a developer's copy of the system from one state to the next by means of semantic operations. These operations are captured by Syde's client, an Eclipse plug-in, triggered at every build action. Thus, the evolution of a system comprises the combination of the sequences of changes produced by each individual.

2.1.1. System representation

Syde models and captures changes of Java systems. It stores and analyzes constructs such as classes and methods, instead of files and lines. To this aim, a system is modeled as an abstract syntax tree (AST) containing nodes—which represent packages—classes, methods, and fields. 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. The current state of a system is therefore represented by keeping track of one AST per developer.

2.1.2. Change operations

In CBSE, change operations represent the evolution of the system instead of file versions. A change operation is the representation of a change a developer performs in the 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 [20]). Atomic changes (e.g., insertion, deletion and change of the property of a node) 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 their chronological order, it is possible to generate all the states of a program's evolution.

2.1.3. System architecture

Syde is a client–server application, in which the server records the change operations, maintains the current state of a project and publishes information about current and past activities 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.

2.2. Replay

Replay is one of the plug-ins that compose Syde's client. Its goal is to allow developers to explore the evolution of a system by chronologically replaying the changes collected by Syde. Since atomic changes are too fine-grained to be shown individually, Replay groups them by timestamp, author and artifact (package or class), i.e., all the changes that were performed by a developer in a class between two subsequent builds are grouped together based on the last build's timestamp. Within a group there cannot be more than one change to one artifact, thus we maintain the granularity of the changes.

2.2.1. Change groups

Each change group contains the following information:

- the set of changed artifacts, such as packages, classes, methods, or fields;
- the type of change for each artifact, which can be insertion, deletion or change;
- the timestamp of the change, more precisely of the build in which the changes in this group were captured;
- the author of the changes,
- the SCM revision that was the baseline for the change.

2.2.2. Change filters

To help developers address different problems, Replay offers three orthogonal categories of filters applicable to the changes of a system under analysis:

- *Time-based.* They filter the changes based on the time period in which they were performed, specified as a combination of begin and end time.