Software Evolution Comprehension: Replay to the Rescue

Lile Hattori, Marco D'Ambros, Michele Lanza
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

Mircea Lungu
SCG @ University of Berne, Switzerland

***Abstract*—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 spanning over 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.**

We present Replay, an Eclipse plug-in that allows one 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. 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. The experiment shows that Replay leads to a decrease in completion time with respect to a set of software evolution comprehension tasks.

I. Introduction

When evolving a code base, during the development or the maintenance phase, developers keep a mental model of the system – an internal working representation of the software under consideration [27]. 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 [25], [2], [6], [14]. These questions span multiple areas [24] 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.,* [26], [16], [1], [29]) 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 [18], LaToza *et al.* report that: 1) almost all teams have a *team historian*, who is the go-to person for questions about the code; 2) most team members subscribe to the check-in messages to keep themselves updated with the code evolution, though many of them 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 more fundamental problem than poor commit messages: It is related to the coarse granularity at which changes are checked-in 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, large commits can also lead to merge conflicts, duplicated work and conflicting design decisions [8], [3].

In our previous work [10], [11] 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 this paper we present Replay [12], 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 [20] than the aggregated form of commit-based Software Configuration Management (SCM) tools, such as CVS and Subversion.

We present a controlled experiment that we conducted 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. From previous catalogues [25], [2], [6], [14] we selected a set of questions that developers ask, and converted them into a set of tasks to measure both the correctness of the task solutions and their completion time. The contributions of this paper are:

1) *Replay*, a toolset to replay and exploit a fine-grained change repository of a system, thus aiding developers at answering their questions related to software evolution;
2) a report on the design and operation of a controlled experiment to compare the performance of Replay with the baseline tool (SVN client) in performing selected software evolution comprehension tasks;
3) an analysis of the results, which shows a statistically significant advantage of Replay over the baseline in time, and indicates advantages of Replay on correctness.

Structure of the paper. In Section II we review Syde and its change model to subsequently present Replay. In Section III we describe the design and operation of our controlled experiment. In Section IV we analyze the experiment results and discuss the threats to validity. In Section V we present work related to the tool, and to the controlled experiment. In Section VI we present the concluding remarks.

II. Syde and Replay

Syde is a client-server application that records fine-grained information about the evolution of a system developed in a multi-developer setting [10], [11]. It extends Robbes' change-based software evolution model (CBSE) [23] 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.

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.

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 [5]). 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.

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.

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 Syde collects. 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.

Change Group Information: Each change group contains the following information:

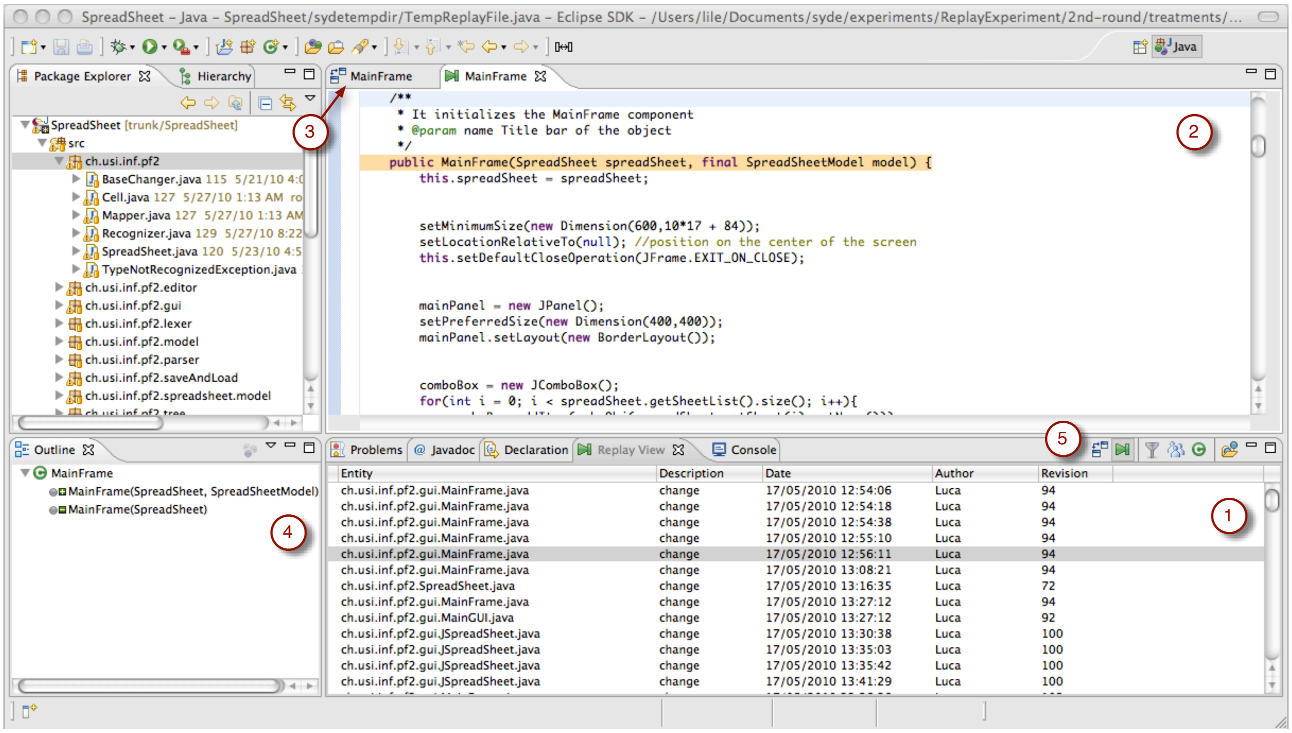
- the set of artifacts that were changed, which can be package, class, method, or field;
- the type of change for each artifact – changes are classified as insertion, deletion or change;
- the timestamp of the change, more precisely of the build in which the changes in this group were captured;
- the author of the changes;
- the SCM revision that was the baseline for the change.

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.
- *Artifact-based.* They focus the replay on a subset of the system artifacts, i.e., classes or packages.
- *Author-based.* They focus the replay on the activity of a subset of the authors in the system. Such a subset can be a team of developers, or a single person.

Visualizing Changes: Figure 1 presents the main components of the Replay plug-in:

- *The Replay View* (Point 1) lists the changes resulting from a search. It shows the entity from the group that is the upper-most node of the AST model of these changes. The change description refers to the entity shown. The other pieces of information provided are the timestamp, author and SCM revision for that group of changes. Selecting on change in the list determines the code to be displayed on one of the editors of the user's choice.
- *The Replay Editor* (Point 2) shows the source code of the change selected on the Replay View. It colors different types of changes with different colors. In the example from Figure 1, the orange text indicates that there was a change on the signature of the constructor in class `MainFrame`. Alternatively, the user can also switch to the Compare Editor (Point 3) to view the changes, which shows the structural and textual comparison between the change selected on the Replay View and the prior change.
- *The Customized Outline View* (Point 4) is a complement to the information shown in the Replay Editor. It gives structural information about the highlighted changes. In the example in Figure 1, it indicates that a parameter was added in the signature of the constructor `MainFrame`.
- *The Toolbar* (Point 5) allows one to (1) choose the way in which the changes are displayed in the main editor (the first two icons), (2) filter changes based on criteria like author, artifact, or time (the next three icons), and (3) improve tool performance by caching the changes locally instead of accessing the server for every search (last icon).


SpreadSheet – Java – SpreadSheet/sydetempdir/TempReplayFile.java – Eclipse SDK – /Users/lile/Documents/syde/experiments/ReplayExperiment/2nd-round/treatments/...
Java
Package Explorer
Hierarchy
SpreadSheet [trunk/SpreadSheet]
src
ch.usi.inf.pf2
BaseChanger.java 115 5/21/10 4:0
Cell.java 127 5/27/10 1:13 AM ro
Mapper.java 127 5/27/10 1:13 AM
Recognizer.java 129 5/27/10 8:22
SpreadSheet.java 120 5/23/10 4:5
TypeNotRecognizedException.java
ch.usi.inf.pf2.editor
ch.usi.inf.pf2.gui
ch.usi.inf.pf2.lexer
ch.usi.inf.pf2.model
ch.usi.inf.pf2.parser
ch.usi.inf.pf2.saveAndLoad
ch.usi.inf.pf2.spreadsheet.model
ch.usi.inf.pf2.tree
MainFrame
MainFrame
Outline
MainFrame
MainFrame(SpreadSheet, SpreadSheetModel)
MainFrame(SpreadSheet)
Problems
Javadoc
Declaration
Replay View
Console
1
2
3
4
5

```
    /**
     * It initializes the MainFrame component
     * @param name Title bar of the object
     */
    public MainFrame(SpreadSheet spreadSheet, final SpreadSheetModel model) {
        this.spreadSheet = spreadSheet;


        setMinimumSize(new Dimension(600,10*17 + 84));
        setLocationRelativeTo(null); //position on the center of the screen
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


        mainPanel = new JPanel();
        setPreferredSize(new Dimension(400,400));
        mainPanel.setLayout(new BorderLayout());


        comboBox = new JComboBox();
        for(int i = 0; i < spreadSheet.getSheetList().size(); i++){
```

Entity	Description	Date	Author	Revision
ch.usi.inf.pf2.gui.MainFrame.java	change	17/05/2010 12:54:06	Luca	94
ch.usi.inf.pf2.gui.MainFrame.java	change	17/05/2010 12:54:18	Luca	94
ch.usi.inf.pf2.gui.MainFrame.java	change	17/05/2010 12:54:38	Luca	94
ch.usi.inf.pf2.gui.MainFrame.java	change	17/05/2010 12:55:10	Luca	94
ch.usi.inf.pf2.gui.MainFrame.java	change	17/05/2010 12:56:11	Luca	94
ch.usi.inf.pf2.gui.MainFrame.java	change	17/05/2010 13:08:21	Luca	94
ch.usi.inf.pf2.SpreadSheet.java	change	17/05/2010 13:16:35	Luca	72
ch.usi.inf.pf2.gui.MainFrame.java	change	17/05/2010 13:27:12	Luca	94
ch.usi.inf.pf2.gui.MainGUI.java	change	17/05/2010 13:27:12	Luca	92
ch.usi.inf.pf2.gui.JSpreadSheet.java	change	17/05/2010 13:30:38	Luca	100
ch.usi.inf.pf2.gui.JSpreadSheet.java	change	17/05/2010 13:35:03	Luca	100
ch.usi.inf.pf2.gui.JSpreadSheet.java	change	17/05/2010 13:35:42	Luca	100
ch.usi.inf.pf2.gui.JSpreadSheet.java	change	17/05/2010 13:41:29	Luca	100

Figure 1. The main components of the Replay plug-in

To watch the changes replayed in chronological order, the user can navigate through the change list in the Replay View and observe the information shown on the Replay (or the Compare) Editor and in the Outline View. Watching a development session gives the user access to which classes were changed, which parts within the classes were changed, the order in which they were changed, by whom, *etc.*

III. Experimental Design

We want to quantitatively evaluate the effectiveness and efficiency of Replay on helping developers to answer questions related to the evolution of a system.

A. *Research Questions and Hypotheses*

We raise the following research questions:

RQ1 Does the use of Replay reduce the *time* for answering software evolution questions compared to SCM-based tools?

RQ2 Does the use of Replay increase the *correctness* of the answers to software evolution questions compared to SCM-based tools?

RQ3 Does the user's *experience level* affect the potential benefits of using Replay in terms of correctness and time?

RQ4 Which *type* of questions can we identify that benefit most from the use of Replay?

Table I
Null and Alternative Hypotheses

	Null Hypotheses		Alternative Hypotheses
$H1_0$	The tool does not impact the time required to answer soft. evolution questions.	$H1$	The tool impacts the time required to answer soft. evolution questions.
$H2_0$	The tool does not impact the correctness of the answers to soft. evolution questions.	$H2$	The tool impacts the correctness of the answers to soft. evolution questions.

The null and alternative hypotheses associated with the first two questions are formulated in Table I.

To test the hypotheses $H1_0$ and $H2_0$ we define a series of tasks that are to be addressed by the control and the experimental group. The control group (Eclipse+SVN) uses an Eclipse installation with default development tools and Subclipse[1] to answer the questions accessing the change history from SVN. The experimental group (Eclipse+Replay) uses the same Eclipse installation with default development tools and Replay to access Syde's change history. We maintain a between-subject design, meaning that each subject is part of either the control or the experimental group.

To answer RQ3 we analyze the data within blocks to check whether the experience level influences the participants' performance. For the last question we perform a separate analysis of correctness and completion time for each task.

[1] Subclipse provides support for SVN in Eclipse http://subclipse.tigris.org/

Table II
TASKS' DESCRIPTION AND GOAL

Id	Description	Goal
1	Imagine that you are joining the project's team to replace a member. Find out what he was working on, so you can start from what he left unfinished. Identify the two classes he changed the most in the past days.	Becoming familiar with someone else's work
2	You have just started to work on a set of classes, and want to find out whether someone else has recently changed them before you commit your changes. Identify the methods that someone else has also changed.	Becoming aware of team activity
3	You have identified one of the main classes of the system but cannot quite understand it. Look for experts who can help you by searching who has recently changed it the most.	Finding experts at the class level of abstraction
4	You are taking over the responsibility of a class and your first task is to refactor the code to improve its design and readability. You want to start with the most complicated feature, because it will need most of your effort. From the list below, identify the feature that provoked the largest number of changes.	Relating a feature to code changes
5	You are given the description of a defect and instructions to reproduce it. Find out the origin of this defect, when and by whom it was introduced. Propose a fix to it.	Tracking back the introduction of a defect
6	Before you joined the team the system underwent a major refactoring, which involved the deletion of a class and restructuring of other classes. Investigate why this refactoring took place.	Understanding the rationale behind past refactorings

B. Object

The system we chose to be the object of this experiment is called *SpreadSheet*. Developed by a team of 4 BSc students, it is a simple spreadsheet application with support for basic mathematic formulae. Its development lasted for 6 weeks, and at the end, the project counted 13 packages, 77 classes, 286 methods, totaling 1,882 lines of Java code. The number of SVN commits was 137, while the number of recorded Syde changes was 11,661. The choice of this specific system is constrained by the need of having the change history, collected from both Syde and SVN, for the entire development cycle. Thus, it was not possible to choose an open-source system, nor did we have a team working on a commercial system at our disposal. This choice implies some threats to validity discussed in Section IV-G.

C. Task Design

We are interested in evaluating whether Replay outperforms a baseline (Subclipse) for assisting developers in answering comprehension questions related to a system's evolution. We considered previous catalogues of questions [25], [2], [6], [14], selected those that can be answered by investigating the change history of the system, and created corresponding tasks. Table II provides a short description of each task together with its goal. Each task is an adapted version of a question from previous catalogues.

In the handout distributed to the subjects, the descriptions of the tasks are integrated in the following hypothetical scenario: the subject is joining a team to replace a developer that left. The scenario makes the subjects feel as if they have become part of the team and are gradually learning the system while solving the experimental tasks.

D. Subjects

We conducted the experiment with 29 subjects: 5 MSc students, 22 PhD students, one postdoc, and one professor. The participant's average age was 28.85, comprising 10 different nationalities. The MSc students, the postdoc, and the professor have a software engineering background. The PhDstudents have diverse backgrounds, such as information retrieval, human-computer interaction, security, and software engineering. Participation was on a voluntary basis. None of the participants had previous experience with Replay.

E. Operation

The operation is composed of several experimental runs. Each run includes a training session of *ca.* 15 minutes and one experimental session. Each training session consists of a tutorial on the tool usage given by the experimenter, followed by a hands-on session, where the subjects perform some small tasks and can ask clarification questions. The experimental session is composed of six tasks, with time limit as follows: 10 minutes for each of the first four tasks, 20 minutes for task five, and unlimited time for task six.

The session is conducted by using the subject's laptop, and instructions are provided to configure it for the experiment. The control group had Eclipse and Subclipse installed; a local SVN server was provided. For the experimental group, the subjects were asked to install Eclipse and Replay.

There were 7 experimental runs in 3 locations: 1 run with 7 participants at the Univ. of Berne; 1 with 6 participants at the Univ. of Zurich; 4 with 2 participants, 2 with 1 participant, and 1 with 6 participants at the Univ. of Lugano.

F. Pilot Studies

To refine the experiment design and make Replay mature enough to guarantee an operation without technical impediments, we ran 5 pilot studies involving 19 people over the course of 4 months. Regarding Replay, we put most of the effort in improving its performance on retrieving the change history to be comparable to Subclipse's performance. Several experimental parameters were adjusted after each pilot, including the description and quantity of the tasks, the time limit of each task, the computer configuration, and the handout. One of the parameters that was carefully evaluated was the slowdown caused by the tools' configurations, which led to the experimental group running Syde's server locally, and the control group having access to a local SVN repository.

G. Data Collection

Personal information. Before the experiment, we collected information about the subject (*e.g.,* age, affiliation) and the subject's experience with Java, Eclipse and Subversion.

Timing data. To time the participants, we adopted two strategies. When the session involved up to two subjects, the experimenter timed them manually. When the session had more than two subjects, the experimenter used a timing web application to time each subject, and also to show them their remaining time. In both cases, the experimenter notified the subjects when they went overtime, and allowed them to write down their findings before going to the next task.

Correctness data. To convert the solutions into quantitative values, we established a grading system. Each task is worth 1 point, evenly distributed according to the number of correct answers that must be entered., *e.g.,* if there are 4 correct answers, each is worth 0.25, while each wrong answer counts as -0.25. The correct answers were determined by the experimenter, and double-checked by two other persons.

Participant feedback. The experiment ended with a debriefing questionnaire, where the subjects assessed the time pressure, the difficulty of the tasks and whether they were realistic. The subjects were also given the opportunity to write down their opinions about the experiment and the tools.

IV. Analysis and Interpretation

We performed a preliminary analysis on the subjects' opinion about the tasks to check for exceptional conditions.

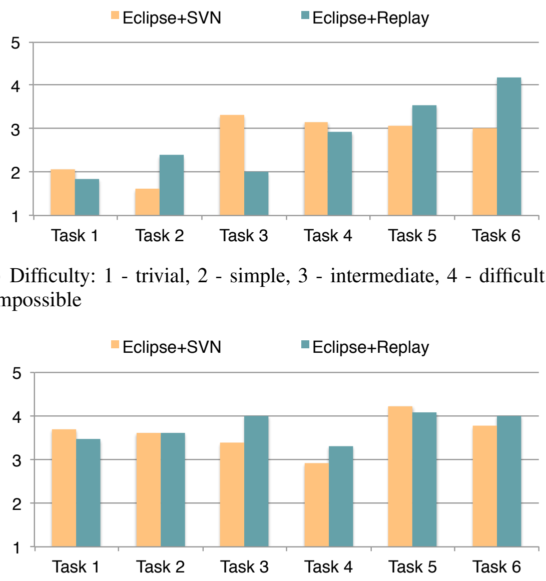
We asked the subjects to indicate how much time pressure they felt during the experiment from 1 (no pressure) to 5 (too much pressure). The average time pressure reported is 3.08 (stdev. 0.86) for the control group and 3.50 (stdev. 0.80) for the experimental group, who felt slightly more time pressure.

We sorted the tasks in increasing order of difficulty throughout the experiment, which is confirmed by the subjects' assessment in Figure 2(a). Although there is a great difference on the perceived difficulty between control and experiment groups in tasks 3 and 6, they do not characterize a high discrepancy in terms of both completion time and correctness, thus we decided to maintain these tasks in the analysis. Since task 6 required a subjective answer in the form of a short essay, it is not included in the statistical test.

The participants felt that the tasks reflect situations that happen in real development scenario (cf. Figure 2(b)). Task 4 received the lowest grading, especially from the control group. We believe that this grading is due to the formulation of the task description rather than the task's goal.

A. Subject Analysis

We followed the suggestions of Wohlin *et al.* [30] regarding the removal of outliers caused by exceptional conditions before performing our statistical test. One subject from the control group was unable to finish the experiment in the allotted time due to lack of experience with Eclipse. One subject from the experimental group did not follow the instructions provided in the handout regarding the tools he was allowed to use, and used the tools reserved to the control group instead. One subject from the experimental group did not understand the concept of fine-grained changes provided by Replay. His answers clearly showed that he did not use the tool, but rather answered randomly, characterizing himself as an outlier both in terms of correctness (low grading) and time (low completion time). We excluded these three cases from the statistical analysis.



(a) Difficulty: 1 - trivial, 2 - simple, 3 - intermediate, 4 - difficult, 5 - impossible

(b) Realism: 1 - strongly disagree, 2 - disagree, 3 - undecided, 4 - agree, 5 - strongly agree

Figure 2. Average perceived difficulty and realism of the tasks

Table III
Subject Distribution

	Eclipse+SVN	Eclipse+Replay	Total
Beginner	7	6	13
Advanced	6	7	13
Total	13	13	26

After removing the outliers, we were left with 26 subjects. We previously assigned treatments to subjects using randomization and blocking according to their experience level. We asked the subjects to indicate the number of years of experience they have in programming in Java, using Eclipse, and using SVN. The criterium used for the blocking was: A subject is considered advanced only if he has at least four years of experience with Java and Eclipse, and at least one year of experience with SVN. If one of these criteria is not met, the subject is classified as beginner. As a result of the random assignment and after the removal of the outliers, we obtained a fair distribution of subjects, as shown in Table III.

Table IV
DESCRIPTIVE STATISTICS OF THE EXPERIMENT RESULTS

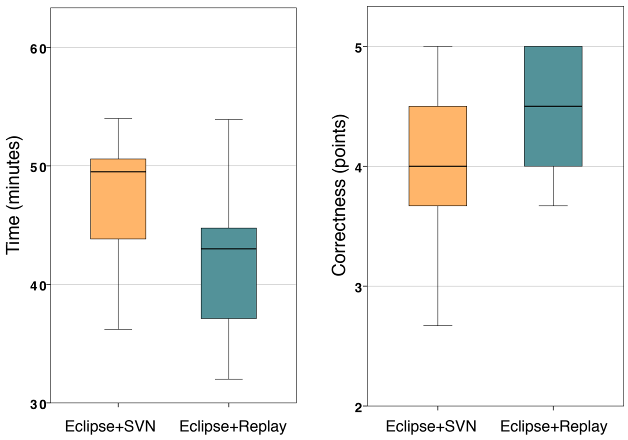
	Group	Mean	Diff.	Min.	Max.	Stdev.	S-W p-value	Levene	Student's t-test t	df	p-value	MWU p-value
Time (minutes)	Eclipse+SVN	44.75		34.00	54.00	6.20	0.213					
	Eclipse+Replay	41.61	-6.84%	32.00	53.92	6.11	0.576	0.759	2.222	24	0.036	
Correctness (points)	Eclipse+SVN	3.94		2.67	5.00	0.65	0.669					
	Eclipse+Replay	4.44	+12.69%	3.67	5.00	0.53	0.022					0.072

B. *Interpretation of the Results*

The design of our experiment is a between-subjects with balanced design, and one independent variable, *i.e.,* the tool. The choice of the hypothesis test depends on whether the sample distributions are normal and have equal variances. If it meets these two requirements, we can choose the parametric Student's t-test, otherwise, we should use the nonparametric Mann-Whitney U test.

We performed the Shapiro-Wilk test of normality, which only rejected the hypothesis that the experimental sample for correctness is normal. For completion time, we also performed the Levene test and verified that the samples have equal variances. The descriptive statistics related to correctness and completion time are presented in Table IV.

Since the completion time is normally distributed with equal variances, we use the Student's t-test for its analysis. For correctness we must use the Mann-Whitney U test.



(a)

(b)

Figure 3. Box plots of completion time and correctness

C. *Results on Completion Time*

We first test the null hypothesis $H1_0$, which states that the use of the tool Replay does not impact the time required to complete the assigned tasks.

Table IV shows that the experimental (Eclipse+Replay) group took on average 6.84% less time to complete the tasks than the control (Eclipse+SVN) group, and that this result is statistically significant at the 95% confidence interval (p-value $= 0.036 < 0.05$ for the t-test). With these results, we can reject the null hypothesis $H1_0$ in favor of the alternative hypothesis $H1$, and positively answer RQ1.

Figure 3(a) shows a box plot[2] of the total time (in minutes) spent by the subjects on the first five tasks. As we can see, the 75th percentile of the experimental group is roughly at the same level of the 25th percentile of the control group. This means that 75% of the subjects from the experimental group completed the tasks before or at about the same time as 75% of the subjects from the control group.

The variability (or range) of completion time is slightly higher in the experimental group than in the control group. One factor that might have influenced this result is that Replay was unknown to everyone, while most of the subjects had some experience with SVN. In addition, some subjects spent a long time getting used to Replay while doing the warm up task, and asked for help when they were struggling with the tool, while others quickly completed the warm up task, and seldom asked for help. Therefore, previous experiences of the subjects of the experimental group with using other tools (including SVN itself) and their dedication on understanding Replay before starting the tasks might have resulted in a higher variability in completion time.

The fine granularity and the large amount of changes that the experimental group had to look through were common complaints, which might have influenced the completion time of the group. However, these "drawbacks" of the Replay tool did not prevent the experimental group to outperform the control group in terms of completion time.

D. *Results on Correctness*

Table IV shows that the experimental group obtained, on average, a score 12.69% higher than the control group. However, this result is not statistically significant at the 95% confidence interval (p-value $= 0.072 > 0.05$ for MWU test), but it is at the 90%. We are unable to fully reject the null hypothesis $H2_0$, and partially answer RQ2.

Even though the results are not statistically significant at the 95% confidence interval, Figure 3(b) shows evidence that the experimental group had a superior performance than the control group. The 50th percentile of the experimental

[2] The top of the box represents the 75th percentile, the bottom of the box the 25th percentile, and the line in the middle the 50th percentile (median).

group is at the same level of the 75th percentile of the control group, *i.e.,* 50% of the subjects from the experimental group obtained higher (or equivalent) score than 75% of the subjects from the control group.

Furthermore, parametric tests (*e.g.,* t-test) are more powerful than nonparametric tests (*e.g.,* MWU), meaning that nonparametric tests need more samples or greater difference in the values to yield statistically significant results. We argue that this is the main reason for the MWU test to have retained the null hypothesis at 0.05 significance level. As a simple test, we have duplicated the experiment dataset and reran the MWU test, obtaining a p-value$= 0.009$.

E. Influence of the Experience Level

We compared the correctness and completion time across the two levels of experience, i.e., beginner and advanced. Figure 4 shows that the experimental group outperformed the control group in both correctness and completion time, regardless of the experience level. Even though the number of subjects per experience level is too low to yield statistically significant results, we draw a couple of observations based on the box plots.

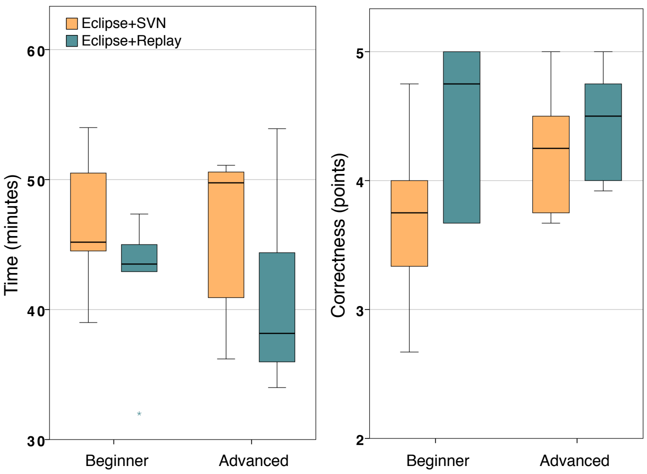


Figure 4. Beginner versus advanced

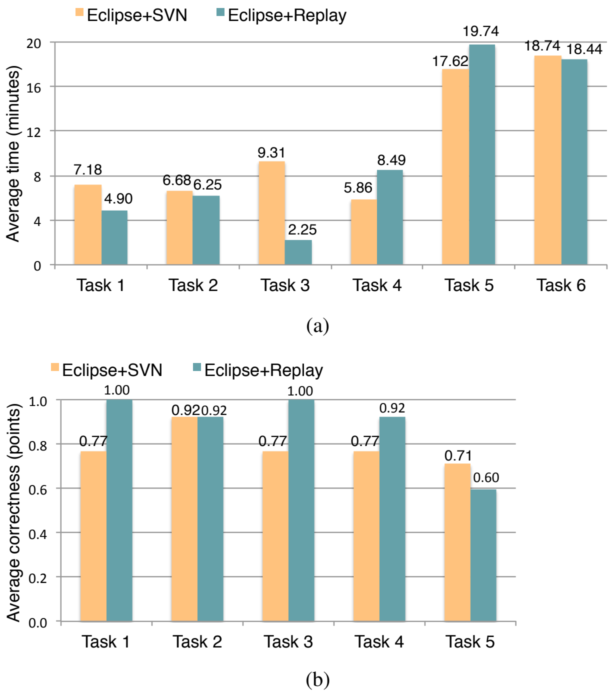
An interesting fact is that the variability of the experimental group was lower than the one from the control group for the beginners, while the inverse can be observed for the experts in terms of completion time. Our assumption is that in the case of beginners, since both Replay and Subclipse are new to them, the learning curve of Subclipse is steeper than the one of Replay. For those unfamiliar with Subclipse, we gave a tutorial on its usage and allowed the subjects to get used to it before starting to perform the tasks. An interesting feedback we collected from some experts is that they felt they were so used to look at the changes the "SVN way" that it was not easy to adapt to Replay, which hindered their performance.

The higher variability in completion time observed in the advanced-experimental group can also be explained by their efforts to adapt to Replay. In terms of correctness, both beginners and advanced from the experimental group had lower variability than their respective from the control group. A factor we attribute to this result is the coarse granularity of the information contained in the SVN repository, which can be the subject of multiple interpretations.

We can answer RQ3 by stating that the users' experience level does not affect the potential benefits of using Replay in terms of correctness. However, the results suggest that the users' experience might influence their efficiency.

F. Individual Task Analysis

To identify which type of tasks can benefit most from the use of Replay (RQ4), we examine the performance of the two tools per task. Figure 5 shows the average correctness and completion time for each task.



(a)

(b)

Figure 5. Average completion time and correctness per task

Task 1 – Becoming familiar with someone else's work. The subjects are asked to familiarize with recent changes made by one developer, and to identify the two classes this developer worked on the most. The experimental group achieved an excellent performance, while the control group took more time and had lower grading. Parnin and DeLine [20] have observed that when a developer resumes one of his interrupted tasks, he prefers to see past changes chronologically than in aggregated form. Our findings complement theirs by showing a better performance of those seeing the changes made by others chronologically.

Task 2 – Becoming aware of team activity. The goal of this task is to identify which methods were recently changed and by whom. Although we have developed a plug-in that directly targets awareness [17], Replay can also be used for this activity. The results show that Replay is slightly more efficient than the baseline for this task.

Task 3 – Finding experts at the level of abstraction of classes. In this task, the subjects are asked to identify experts of a class based on the recent changes it underwent. The results are very similar to the ones from task 1, and show that Replay outperforms the baseline for this task.

Task 4 – Relating a feature to code changes. This task involves identifying the different features inside a class and identifying the one that has changed the most. While the experimental group took, on average, more time than the control group to complete the task, they performed better. A possible explanation for this result is that while the number of fine-grained changes was much higher than the number of commits, it was easier to detect the different features by following the chronological sequences of changes in Replay, than by looking at the aggregated results on SVN.

Task 5 – Tracking back the introduction of a defect. This task was added to the experiment after collecting suggestions on the pilot study indicating that Replay could be useful for identifying the change that caused a defect. However, the control group performed better than the experimental group both in terms of completion time and correctness. The results and the feedback collected from the experimental group indicate that the fine granularity of changes shown by Replay is counter-productive for this task.

Task 6 – Understanding the rationale behind past refactorings. The goal is to understand the design decisions behind a major refactoring performed in the past. Since this is a task that requires a descriptive answer, only the quantitative scores of completion time are available. On average, both groups took similar time to complete the task, however, Figure 2(a) indicates that the experimental group struggled to understand the refactoring (the average difficulty was higher than 4). The essays confirm that the control group was able to understand better the reasons behind the refactoring. Similar to task 5, the experimental group complained that the information provided by Replay was too fine-grained to allow them to see the "big picture" of the refactoring.

Summary. For tasks 1 and 3, which needed fine-grained information about recent changes, Replay was more efficient and effective than the baseline. For task 4, in which the chronology of changes was important, Replay showed to be more effective. When the subjects had to relate recent changes to authors, Replay's performance is comparable to the baseline. At tasks that required looking at information over a long time span (tasks 5 and 6), the fine-granularity of Replay prevented the subjects from performing well.

G. Threats to Validity

Internal Validity

Subjects. To reduce the threat that the subjects may not have been competent enough, we ensured that they had sufficient skills on the tools used during the experiment. To lessen the threat that the subjects' expertise may not have been fairly distributed, we used randomization and blocking to assign treatments to subjects.

Tasks. The tasks were designed by the authors of this paper, and thus may have been biased toward Replay. To mitigate this threat, we have based the tasks on valid questions that developers ask, which were reported in previous catalogues. The tasks might have been too difficult and the allotted time per task may have been insufficient. To alleviate these threats, we conducted several pilot runs to fine-tune them. Furthermore, the task that was classified as too difficult by the experimental group (task 6) was not designed to be included on the statistical analysis.

Experimental runs. There were several runs and the differences among them may have influenced the results. However, the several pilot runs with different number of participants allowed us to have a stable and reliable experimental setup.

Training. We provided a training session on Replay to all subjects of the experimental group, while the subjects from the control group were assumed to have familiarity with the baseline tools. To mitigate the fact that lack of proper training might have influenced the results, we also provided a training session on Subclipse when a subject from the control group was unfamiliar with the tool.

External Validity

Subjects. The fact that the subjects of the experiment were from academia may have limited our ability to generalize the results to the industrial environment. It is difficult to recruit practitioners who are willing to dedicate 2 hours of their time to do an experiment. To mitigate the lack of practitioners, we assume a relatively high average expertise level of the 26 selected participants. This assumption is sustained by the subjective assessment of the expertise provided by the subjects prior to the experiment. They were asked to rank their perceived knowledge according to the scale: 1–none, 2–beginner, 3–knowledgeable, 4–advanced, and 5–expert. The results—Java (avg. 3.65, stdev. 0.93), Eclipse (avg. 3.42, stdev. 0.94), SVN (avg. 3.04, stdev. 1.25)—indicate an average of knowledgeable subjects.

Tasks. Our choice of tasks may not reflect real questions related to software evolution. This threat is neutralized by our reliance on existing catalogues [25], [2], [6], [14], which were mainly constructed through surveys and interviews with practitioners, to elaborate the tasks.

Object system. The representativeness of our object system is an important threat, since it is a small system that was developed by undergraduate students. Therefore, it may not reflect the complexity of large-scale industrial systems. The use of more than one object system may have yielded different or more reliable results. However, the choice of the object system was constrained by the need of having the change history from both Syde and SVN.

V. Related Work

A. Approaches Related to Replay

To our knowledge, Replay is the first tool to support replaying development sessions in a collaborative environment. However, other tools support programmers with their quests. Fritz and Murphy propose a prototype that combines different information fragments (source code, work items, change sets, teams, comments, wiki) to support 78 questions software developers ask about a development project [6]. The tool Ferret combines four different sources of information (static, dynamic, evolutionary, and Eclipse PDE) to build a knowledge base for answering conceptual queries [2]. James is a knowledge base, composed of IDE interactions and micro-blogging, to support developers with their quests [9].

Looking at our work in the broader context of software evolution, there are various lines of research that relate to ours. In the software evolution analysis context, several approaches make use of the changes performed to a system over its lifetime to support its comprehension: Lanza, Gîrba *et al.*, and Lungu *et al.* summarize and visualize respectively the evolution of classes [15], the evolution of class hierarchies [7], and the evolution of inter-module relationships [19]. In these works the history is not replayed, but 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.

A few approaches focus on replaying the changes that happened in a system. Wettel and Lanza visualize 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 [28]. Hindle *et al.* present an animation of the evolution of the architecture of a system [13] in the Yarn tool. The animation presents the evolution of the relationships between the modules of the system. These approaches allow the animation of the changes, but present the changes at a high level of abstraction, from which the code is not accessible.

One major difference between our work and the ones aforementioned is the level of detail of the data. In most of the approaches the data is extracted from commit-based SCM systems, implying that changes between versions can be arbitrarily complex. An approach that uses fine-grained change information was proposed by Robbes [22]. Although he collects fine-grained information from software systems, Robbes does not use it to support the replaying of the changes, but he exploits it for other purposes, *e.g.,* to detect and characterize development sessions [24]. Dig, instead, uses a change-centric approach to record sequences of refactorings, and to replay them on other library-based applications [4].

B. Empirical Studies

There are relatively few empirical studies by means of controlled experiments in software engineering. Further, there is no controlled experiment that directly relates to ours: answering developers' questions related to software evolution. However, there are a number of controlled experiments related to software evolution and program comprehension.

Quante evaluates through a controlled experiment the support provided by dynamic object graphs on answering a set of program comprehension tasks [21]. Cornelissen *et al.* performe a controlled experiment to evaluate the use of Extravis, an execution trace visualization tool, to answer program comprehension tasks [1]. Wettel *et al.* assess the use of CodeCity to perform program comprehension and quality assessment tasks [29].

The major difference between these controlled experiments and ours is that they evaluate tools that visualize data other than source code (dynamic graphs, execution traces, system models) to support program comprehension. We evaluate a tool that allows a developer to investigate the history of the system by looking directly at its source code.

VI. Conclusion

We presented Replay, a tool that allows developers to explore the evolution history of a system by chronologically replaying the fine-grained changes collected by Syde. We argue that Replay can be useful to help developers in finding answers to common questions they raise during development and maintenance that are related to the evolution of a system.

We conducted a controlled experiment to evaluate 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. The results indicate that Replay leads to an improvement in both correctness (12.7%) and completion time (6.8%), with the latter being statistically significant at 95% confidence interval. As an indication of a superior performance of the experimental group in terms of correctness, 50% of this group performed better than 75% of the control group. In terms of completion time, 75% of the experimental group was faster than (or equivalent to) 75% of the control group. These results show that there are benefits of using Replay over of the state of the practice tools for most of the tasks included in this empirical evaluation.

The per-task analysis of the results, which provided a number of insights on the type of tasks our approach supports best. For tasks that needed fine-grained change information, or in which the chronological order was important, Replay outperformed the baseline. However, when the tasks required a general overview of the changes, Replay did not perform better than the baseline.

Acknowledgment

We thank Alberto Bacchelli and Richard Wettel for helping us with the design; the subjects and the participants of the pilot study; Serge Demeyer, Harald Gall, Oscar Nierstrasz, Arie van Deursen, Anja Guzzi, Quinten Soetens, and Michael Würsch for helping us with local organizations. Hattori is supported by the Swiss Science foundation through the project "GSync" (SNF Project No. 129496).

References

[1] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Trans. on Software Engineering*, 99, 2010.

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

[3] 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 (Intl. ACM SIGGROUP Conf. on Supporting Group Work)*, pages 105–114. ACM Press, 2003.

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

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

[6] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of ICSE 2010 (32nd ACM/IEEE Intl. Conf. on Software Engineering)*, pages 175–184. IEE Computer Society, 2010.

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

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

[9] A. Guzzi, M. Pinzger, and A. van Deursen. Combining microblogging and ide interactions to support developers in their quests. In *Proceedings of ICSM2010 (IEEE Intl. Conf. on Software Maintenance)*, pages 1 –5, 2010.

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

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

[12] L. Hattori, M. Lungu, and M. Lanza. Replaying past changes on multi-developer projects. In *Proceedings of IWPSE-EVOL 2010 (Joint 11th Intl. Workshop on Principles of Software Evolution and 5th ERCIM Workshop on Software Evolution)*, pages 13–22, 2010.

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

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

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

[16] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. Codecrawler — an information visualization tool for program comprehension. In *Proceedings of ICSE 2005 (27th IEEE Intl. Conf. on Software Engineering)*, pages 672–673. ACM Press, 2005.

[17] 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 Conf. on Software Maintenance and Reengineering)*, pages 207–216. IEEE CS Press, 2010.

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

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

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

[21] J. Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *Proceedings of ICPC 2008 (16th Intl. Conf. on Program Comprehension)*, pages 73–82. IEEE CS Press, 2008.

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

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

[24] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *Proceedings of ICPC 2007 (15th IEEE Intl. Conf. on Program Comprehension)*, pages 155–164. IEEE CS Press, 2007.

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

[26] M.-A. D. Storey. Theories, methods and tools in program comprehension: past, present and future. In *Proceedings of IWPC 2005 (13th Intl. Workshop on Program Comprehension)*, pages 181 – 191, May 2005.

[27] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28:44–55, 1995.

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

[29] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *Proceedings of ICSE 2011 (33rd Intl. Conf. on Software Engineeering)*, page to be published, 2011.

[30] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.