In Hasso Plattner, Christoph Meinel, and Larry Leifer (eds.)
Design Thinking Research: Building Innovators (pages 241-261)
Springer 2015 (doi:10.1007/978-3-319-06823-7_13)

How Cost Reduction in Recovery Improves Performance in Program Design Tasks

Bastian Steinert and Robert Hirschfeld[1]

Abstract Changing source code often leads to undesired implications, raising the need for recovery actions. Programmers need to manually keep recovery costs low by working in a structured and disciplined manner and regularly performing practices such as testing and versioning. While additional tool support can alleviate this constant need, the question is whether it affects programming performance?
In a controlled lab study, 22 participants improved the design of two different applications. Using a repeated measurement setup, we compared the effect of two sets of tools on programming performance: a traditional setting and a setting with our recovery tool called CoExist. CoExist makes it possible to easily revert to previous development states even, if they are not committed explicitly. It also allows forgoing test runs, while still being able to understand the impact of each change later. The results suggest that additional recovery support such as provided with CoExist positively affects programming performance in explorative programming tasks.

1 Introduction

Changing source code easily leads to the need for recovery actions because the changes reveal implications that are not only unexpected but also undesired. They might suddenly turn out inappropriate, turn out more complex than expected, or they might have introduced an error. Programmers then need to withdraw these recent changes, recover knowledge from previous development versions, or locate and fix the error.

To keep the costs for potential recovery needs low, programmers have to follow a structured and disciplined approach. This involves the regular use of testing and versioning tools, but also to perform baby steps and to work only on one thing at a time [1]–[3]. Regular testing helps discover errors early and thus reduces fault localization costs, regular commits help to return to a previous state, and working on one thing at a time makes it easier to commit independent increments - to men-

[1] Software Architecture Group
Hasso Plattner Institute, University of Potsdam, Germany
e-mail: firstname.lastname@hpi.uni-potsdam.de

tion just a few examples. By following these recommendations, programmers can avoid the need for expensive recovery work that easily becomes frustrating.

However, while structure and discipline are certainly useful to get work done, it hardly seems sufficient to be forced to rely on them constantly. On the one hand, it is hard to exert the required discipline when being fascinated by an idea and having the desire to explore it. On the other hand, it is easy to forget to perform recommended practices and it requires much effort to avoid forgetting. This not only takes time but also easily disrupts working on the main task.

The need for structure and discipline is also present when programmers decide to first create a prototype on a separate branch, in order to evaluate a particularly risky idea. One reason is that they might want to reuse the source code and avoid the need to re-implement it. Another reason is that when working on a prototype, it is still likely that changes reveal undesired implications independent of the aspects being evaluated. So, the same rules apply: programmers will still need to recover, and they also need to manually keep recovery cost low, for example, by testing regularly, making meaningful commits, and making only small changes, one at a time.

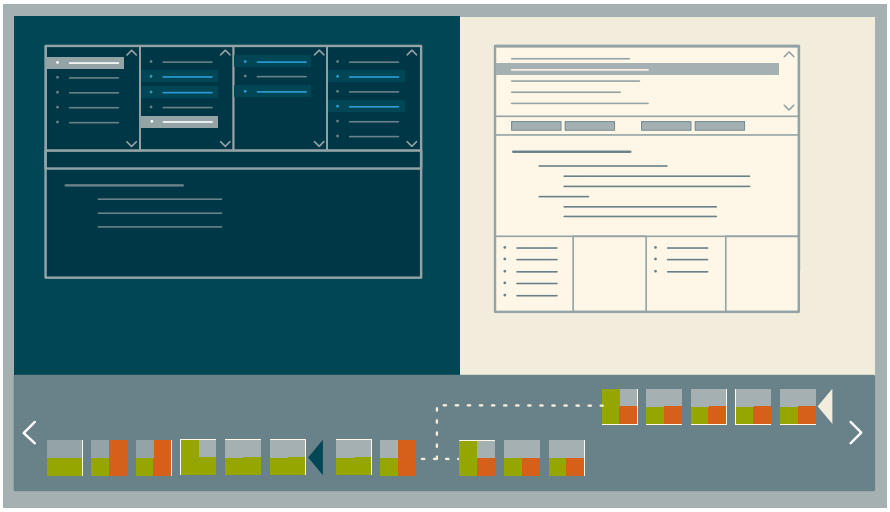

Fig. 1. The CoExist IDE extension featuring continuous versioning, running tests and recording test results in the background, side by side exploring and editing multiple versions.

Additional tool support can help to avoid the constant need for structure and discipline by keeping recovery costs low automatically. We previously presented an IDE extension called CoExist [4], which is implemented in Squeak/Smalltalk. Fig. 1 illustrates main concepts of the user interface. CoExist continuously versions the program under development, runs tests in the background, and provides immediate

access to intermediate development states. It allows programmers to easily recover from undesired situations, also when they forgot to make the appropriate commit or have failed to run the right set of tests regularly. These features enable programmers to ignore recommended practices. They can try out an idea when it comes to mind, make changes as they think of them, and explore the implications, without having to worry about tedious recovery scenarios and how to prevent them.

We hypothesized that such additional recovery support has an effect on programming performance, in particular on tasks that involve a high degree of uncertainty. We speculate that this is the case for three reasons: *A)* making changes directly as one thinks of them supports mental process and is thus more efficient; *B)* The constant need for structure and discipline is tiring and contradicts the need for creative thinking.

A) Why thinking is supported by doing

Programmers should be encouraged to make changes as they think of them, because it will facilitate inference, understanding, and problem solving, as suggested by research findings in design and cognition [5], [6].

It avoids mental overload and keeps working memory free. Making the changes instead of conducting what-if reasoning "frees working memory to perform mental calculations on the elements rather than both keeping elements in mind and operating on them" [6]. Freeing working memory is required because the number of chunks of new information that a human being can keep in mind and process is limited (3 to 4 chunks). Given too many chunks at once, a human being experiences cognitive overload, which impedes learning and problem solving [7], [8].

Making the changes allows for re-interpretation and unexpected discovery. Even if they turn out inappropriate, the changes can trigger new associations. Previously abstract concepts and thoughts will be associated with specific source code elements. When programmers revisit these specific elements, they can see them as something else. They associate abstract concepts with these elements that are different than the original ones. Making the changes brings to mind information from long-term memory that might otherwise not be retrieved [6], [9]. The particular arrangement can also lead to the discovery of unexpected relations and features [9], [10].

B) Why the need for structure and discipline is tiring and in contradiction with the need for creativity

Psychology distinguishes two modes of thinking: fast thinking and slow thinking, often labeled as System 1 and System 2 [11]. While creativity along with intuition is attributed to System 1 (fast thinking mode), while the analytic approach along with suspicion is attributed to System 2 (slow thinking mode). This implies that creative thinking and analytical reasoning don't go well together. Working on a creative programming task is impeded by the need to reflect about current and planned changes and the need to structure the work ahead. If programmers con-

stantly need to be analytical and careful, it will be difficult for them to be creative at the same time.

Furthermore, the need for a structured and disciplined approach to programming requires self-control, which is a form of exhaustive mental work, as the following quotes from [11, Chapter 3] should illustrate:

- "... controlling thoughts and behaviors is one of the tasks of System 2."
- "Too much concern about how well one is doing in a task sometimes disrupts performance by loading short-term memory with pointless anxious thoughts. ... self-control requires attention and effort."
- "an effort of will or self-control is tiring; if you have to force yourself to do something, you are less willing or less able to exert self-control when the next challenge comes around."

These findings give reason to believe that additional recovery support such as CoExist is preferable over a manual method-based approach. We hypothesize that CoExist improves the performance of programmers in explorative tasks. In the remainder of this article, after first describing CoExist ,we report on an experiment conducted to empirically examine our hypothesis.

2 BACKGROUND - THE COEXIST IDE EXTENSIONS

The basis of CoExist takes care of preserving potentially valuable information. It continuously performs commits in the background. Every change to the code base leads to a new version one can go back to. It thus gives users an impression of development versions to *co-exist*. To make the user aware of this background versioning and to allow for selecting previous versions, we have added a version bar (timeline) to the user interface of the programming environment (Fig. 1).

Fig. 2. The (blue) triangle marks the current position in the history - the version that is currently active. When a programmer goes back to a previous version (left), and then continues working, the new changes will appear on a new branch that is implicitly created (right).

By continuously preserving intermediate development states, CoExist enables programmers to go back to a previous development state and to start over as shown in Fig. 2. Starting over from a former development state will implicitly create a new branch of versions. This preserves the changes that are withdrawn, as they might be of use later on.

Fig. 3. Hovering shows which source code element has been changed (left). In addition, holding shift shows the total difference to the previous version (right).

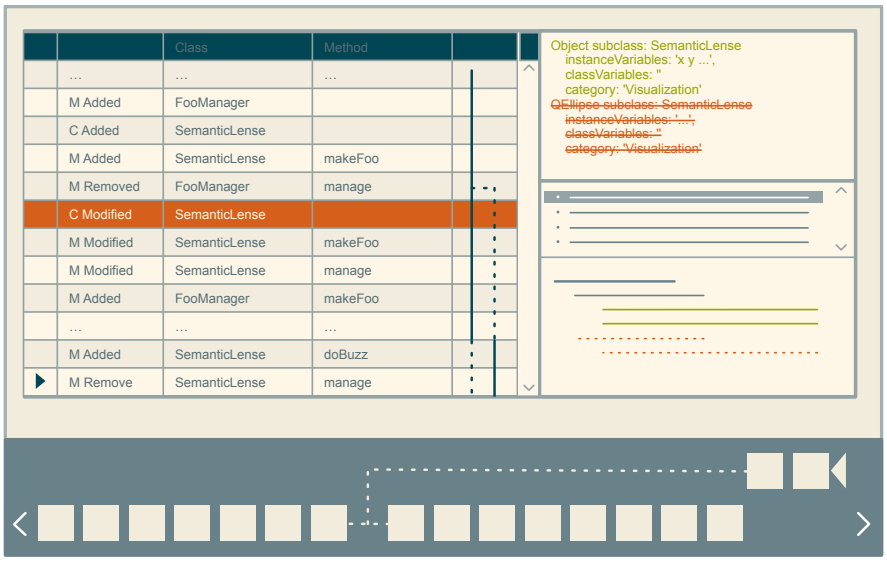

Fig. 4. The version browser provides a tabular view on change history. Selecting a row shows corresponding differences in the panes on the right.

CoExist provides two mechanisms to support programmers in identifying a previous version of interest. First, it provides the version bar, which will highlight

version items that match the currently selected source code element. Hovering the items will display additional information, such as the kind of modification, the affected elements, or the actual change performed (Fig. 3).

Second, programmers can use the version browser to explore information of multiple versions at a glance. The version browser displays basic version information in a table view (Fig. 4), which allows to scan the history fast.

CoExist is meant to close the gap between the undo/redo feature and Version Control Systems such as Git. It is not intended to replace either of them. Furthermore, we acknowledge that conscious and named commits can be useful, but we omitted the possibility of naming or flagging intermediate versions to avoid inducing users to think about it. We also want to explore how far one can go with our approach.

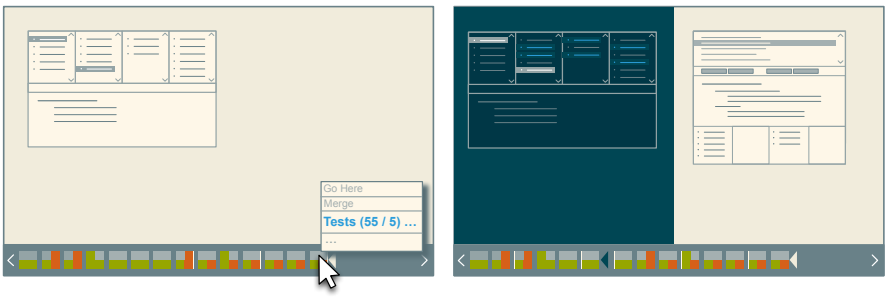

Fig. 5. The items in the version bar are now a visualization of the results of the tests that have been run in the background (left). A second inner environment allows the user to explore a previous version next to the current one (right).

CoExist also allows continuously running analysis programs for newly created versions. As a default, it runs test cases to automatically assess the quality of the change made. The test result for a version is presented in the corresponding item of the version bar (left of Fig. 5). This makes the effect of each change regarding test quality visible. The user can also run other analyses such as performance measurements. CoExist provides full access to version objects and offers a programming interface to run code in the context of a particular version. So, whenever programmers become interested in the impact of their changes, they can easily analyze it in various respects. This allows programmers to ignore these aspects of programming at other times.

Users of CoExist can explore the source code of a previous version and compare it to the current one. They can open a previous version in a separate working environment as shown on the right in Fig. 5, which is useful, when, for example, the programmer suddenly become curious about how certain parts of the source code looked previously or how certain effects were achieved. It is also possible to run and debug programs in the additional working environment. In doing so, CoExist is capable of efficiently recovering knowledge from previous versions, which avoids the need for a precise understanding of every detail before making any changes.

With CoExist, programmers can change source code without worrying about the possibility of making an error. This is because they can rely on tools that will help with whatever their explorations turn up. They no longer have to follow certain best practices in order to keep recovery costs low.

3 Method

3.1 Study Design

Fig. 6 illustrates the experimental setup. Participants have been assigned to either of two groups, the control group or the experimental group. Members of the control group used the regular development tools for both tasks. Members of the experimental group used the regular tools only for task 1, and could additionally rely on CoExist for task 2.

We kept participants unaware of what condition they had been assigned to. However, on day 2, participants in the experimental group could guess that they were receiving special treatment because they were introduced to a new tool and could make use of it. At the same time, participants in the control group were unaware about the experimental treatment. They did not know that the participants in the experimental group were provided with CoExist.

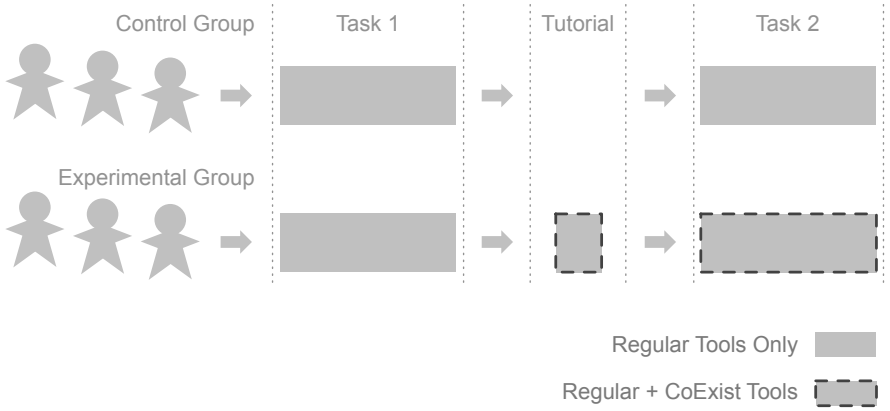

Fig. 6. Our experiment setup to compare performance in program design activities.

The setup resulted in two scores for every participant, which allowed testing for statistically significant differences between task 1 and task 2 as well as between the control and the experimental group. It also allows to test for an interaction effect of the two factors, which is the indicator of whether CoExist affects programming performance.

3.2 Materials and Task

On both days the task has been to improve the source code of relatively small computer games. More specifically, participants were requested to study the source code, to detect design flaws in general and issues of unnecessary complexity in particular, and to improve the source code as much as possible in the given time frame of two hours. The games needed to function properly at the end of the task. To help participants better understand the task, we provided descriptions of possible improvements such as the following:

- Extract methods to shorten and simplify overly long and complicated methods, and to ensure statements have a similar level of abstraction
- Replace conditional branching by polymorphism
- Detect and remove unnecessary conditions or parameters

Participants should imagine that they co-authored the code and now have time to improve it in order to make future development tasks easier. Also, participants were asked to describe their improvements and to help the imaginary team members better understand them. (Most participants followed this instruction by regularly writing commit messages).

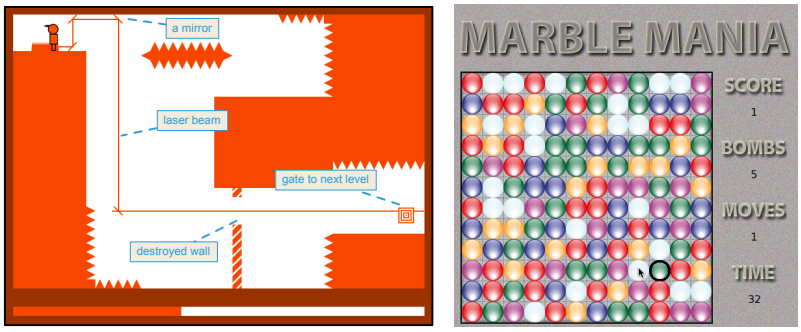

Fig. 7. Screenshots of the games whose source code was improved in the experiment: LaserGame (left) and MarbleMania (right).

On day 1, participants worked on a game called *LaserGame*, and on day 2 they worked on a gamed called *MarbleMania*. Screenshots of both games are shown in Fig. 7. For the LaserGame (on the left), the user has to place mirrors in the field so that the laser is redirected properly to destroy the wall that blocks the way to the gate to the next level. For MarbleMania (on the right), the user has to switch neighboring marbles to create one or more sequences of at least three equally colored marbles, which will then be destroyed, and gravity will slide down marbles from above.

Both games were developed by students in one of our undergraduate courses. The two selected games function properly and provide a simple but nevertheless fun game play. Accordingly, only a little time is required to get familiar with the functionality. Furthermore, for each of the two games, there is significant room for improvement concerning the source code (because they were created by young undergrads who were about to learn what elegant source code is). Furthermore, both games come with a set of tests cases, which also have been developed by the respective students. However, while the offered test cases are useful, they were not sufficient. Manual testing of the games was necessary.

	LaserGame	MarbleMania
# classes	42	26
# methods	397	336
# test cases	50	17
# lines of code	1542	1300

Fig. 8. Size indicators for the games used in the study.

While the numbers shown in Fig. 8 indicate that both games are of similar size, the code base of the LaserGame is easier to understand. The authors of MarbleMania placed a great deal of emphasis on the observer pattern and built in many indirections, which impedes understanding the control flow.

3.3 Participants

We recruited 24 participants, mainly through email lists of previous lectures and projects. Of the 24 participants, 3 were bachelor students who had completed their fourth semester, 6 were bachelor students who had completed their sixth semester (nearly graduated), 13 were master student who had at least completed their eighth semester, and 2 were PhD students. The average age was 23 with a standard deviation of 2. For approximately 5 hours of work, each participant received a voucher worth 60 euros for books on programming-related topics. Of the 24 participants, the results of two were dropped which is discussed in the results section (IV).

Prospective participants needed to have experience in using Squeak/Smalltalk and must had passed their fourth semester. By this time students will have typically attended two of our lectures, in which they use Squeak/Smalltalk for project work. Also, these two lectures cover software design and software engineering topics. Thus we could ensure that all participants had theoretical and practical lessons in topics such as code smells, idioms, design patterns, refactoring, and other related topics.

We have balanced the amount of previously gained experience with Squeak/Smalltalk among both conditions (stratified random sampling). Most participants have used Squeak/Smalltalk only during the project work in our lectures. But 6 participants also have been using Squeak/Smalltalk in spare time projects and/or in their student jobs, so that we could assume these participants had noticeably more experience and were more fluent in using the tools.

3.4 Procedure

We always spread the experiment steps over two days, so that participants worked on both tasks on two different but subsequent days. On both days, the procedure comprised two major steps: an introduction to the game and a two-hour time period for improving the respective codebase. On day two participants of the experimental group received an additional introduction to the CoExist tools before working on the actual tasks, during which they could rely on CoExist as an additional recovery support.

Both tasks were always scheduled for the same time of the day in order to assure similar working conditions (hours past after waking up, hours already spent for work or studies, ...). Typically, we scheduled the task assignments after lunch

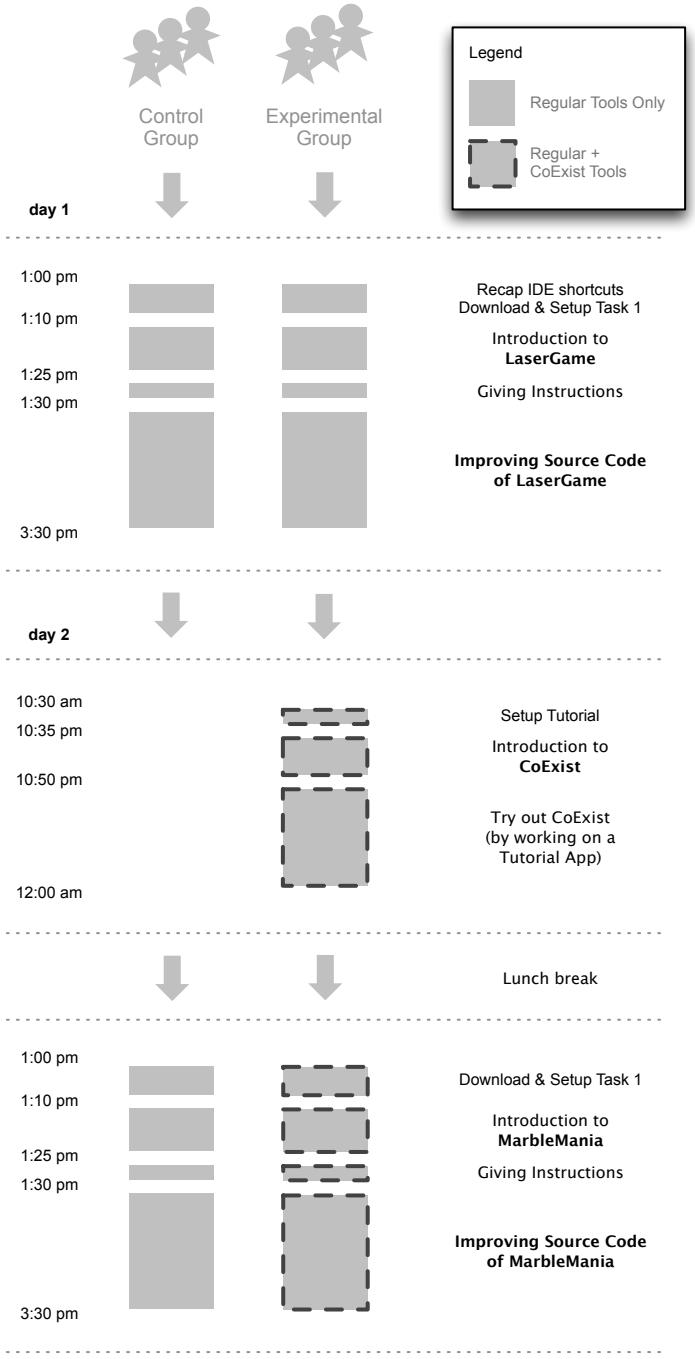

Fig. 9. The experimental procedure for both the control and the experimental group.

so that for day 2, there was time left to run the CoExist tutorial session upfront before lunchtime. (We had to make an exception for three participants, who only had time during the morning or evening hours. As we could not arrange a similar schedule for these participants concerning the CoExist tutorial followed by a large break, these three participants were automatically assigned to the control group).

Fig. 9 illustrates all steps of the experiment. On day 1, participants received a brief recap of IDE shortcuts, which were also written on the whiteboard in the room. The step of *Introduction to <a game>* started with a short explanation of the game play, followed by some time to actually play the game, to understand details, and to get comfortable with it.

3.5 Dependent Measure

To compare the performance of the individual programming sessions, we have operationalized the notion that a programmer can achieve more or less improvements in the given timeframe. We determined performance by *identifying independent increments* among the overall set of made changes, and *quantifying the effort for these increments* by defining sequences of IDE interactions required to reproduce them. This gives a measure of how much actual work was done within the two hours, excluding time that has been spent on activities such as staring into the air or browsing the code base.

1) **Identifying Independent Increments:**

An independent increment is a set of interconnected changes to the code base that represents a meaningful, coherent improvement such as an *ExtractMethod* refactoring, which is comprised of the changes: a) adding a new method and b) replacing statements with a call to the newly created method. Another example for an independent increment is the replacement of code that caches state in an instance variable with code that re-computes the result on every request, or vice versa. Other generic improvements are for example:

- Renaming of an instance variable
- Replace a parameter with a method
- Make use of cascades
- Inline temporary expression
- Replace magic string/number with method

Besides such generic and well-document improvements, an increment can also be specific to a certain application. The following examples are game specific improvements that were identified for the MarbleMania game:

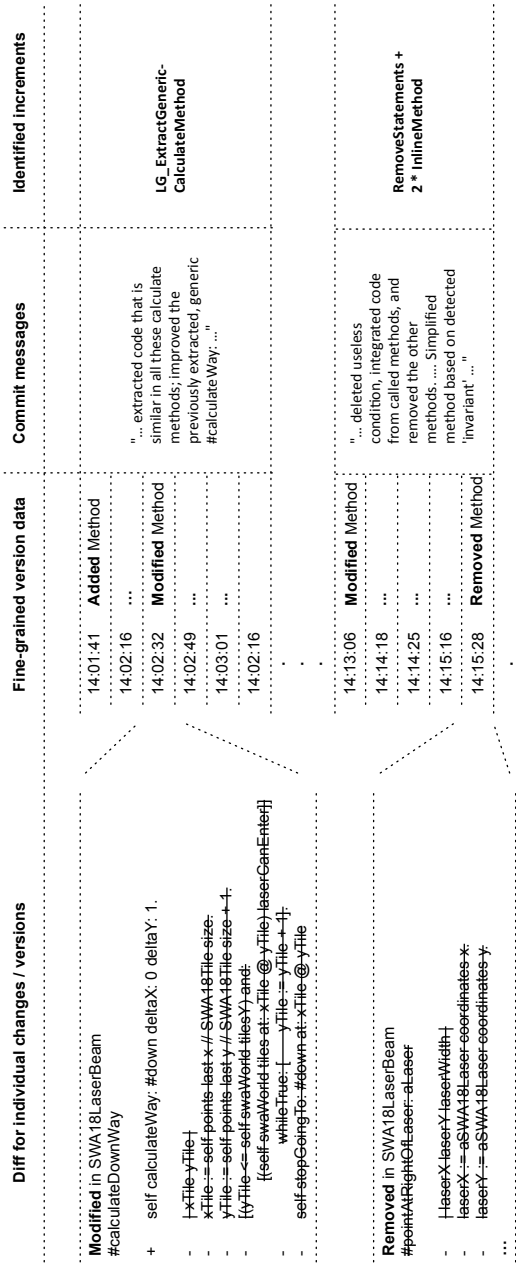

Fig. 10. Excerpt of a spreadsheet with coded version data.

- Replace dictionary that holds information about exchange marbles with instance variables
- Replace “is nil” checks in the Destroyer with null objects (the Destroyer class has the responsibility to “destroy” marbles when, after an exchange, a sequence of three or more marble exists)
- Remove button clicked event handling indirections

For each participant and task, we recorded a fine-grained change history using CoExist’s continuous versioning feature. However, the CoExist tools were neither visible nor accessible to the users, except for the experimental group on day 2. We then analyzed these recorded change histories manually to identify the list of independent increments. For each programming session (per programmer and task), the analysis consisted of two steps to gain a corresponding spreadsheet as illustrated in Fig. 10.

First, we extracted the timestamps of all versions and listed them in a column of a spreadsheet. We then grouped these timestamps according to the commits that subjects made during the task, and put the corresponding commit messages in a second column (illustrated in Fig. 10). The commit messages provide context that helps getting an initial understanding of the changes’ intent.

Second, we hovered over all version items step by step (compare with Fig. 3) to refine our understanding of the made changes, and put names for identified increments in a third column. Such a coded increment can involve only one actual change or consist of many. Sometimes, all the changes made for one commit contribute to one coded improvement. Note that we only coded increments for changes that last until the end of the session. This excludes change sets that were withdrawn later, for example.

2) Quantifying the Effort for Identified Improvements

We measure the effort for every increment by determining the list of IDE interactions that are required to (re-) produce it. Such interactions are, for example: navigating to a method, selecting code and copying it to clipboard, selecting code and replacing it with the content from the clipboard, inserting symbols. Fig. 11 shows two lists of IDE interactions, written down in an executable form (regular Smalltalk code). Executing a script computes a number that represents the effort required to reproduce the described increment.

We determined these scripts by re-implementing every identified increment based on a fresh clean code base, which participants started with. Re-implementing the increments ensured that we had gotten a correct understanding. We always used the direct path to achieve an increment, which might be different than the path made by participants. Thus, we only measured the essential effort and excluded any detours that participants might have made until they eventually knew what they wanted.

```
CvEval >> #renameClass

   self
      navigateTo: #class;
      requestRefactoringDialog;
      insertSymbols: 1;
      checkSuggestionsAndAccept


CvEval >> #lgReplaceCollectionWithMatrix

   self
      navigateTo: #formWidth... in: #Grid;
      selectAndInsert: 5;
      navigateTo: #at: in: #Grid;
      selectAndInsert: 1;
      navigateTo: #at:put in: #Grid;
      selectAndInsert: 1
```

Fig. 11. The first example represents the list of interactions required for the generic RenameClass refactoring, while the second represents an increment that is specific to the LaserGame.

For generic increments such as ExtractMethod or InlineMethod, the required effort can vary: extracting a method with five parameters requires more symbols to be inserted than an extracting a method without any parameters. We accounted for such differences by listing the interactions required for an average case. However, for extreme variations (easy or hard), we used special codes such as ExtractMethodForMagicNumber.

The messages used in these scripts call utility methods that are typically composed of more fine-grained interactions. At the end, all descriptions rely on four elementary interactions, which are: `#positionMouse`, `#pressKey`, `#briefly-CheckCode`, and `#insertSymbols: aNumber`. The methods for these elementary interactions increment a counter variable when they are executed. While the former three increment the counter by one, the latter increments the counter by three for every symbol inserted. So we assume that writing a symbol of an average length is three times the effort of pressing a single key. (While this ratio seemed particularly meaningful to us, we also computed the final numbers with a ratio of two and four. The alternative outcomes, however, show a similar result. In particular, a statistical analysis using ANOVA also reveals a significant interaction effect.)

4 Results and Discussion

Fig. 12 shows the result scores for each participant and task, the accumulated points for the identified increment. Note that while we recruited 24 participants, we only present and further analyze the scores of 22 participants. One of the two participants had to be dropped because after the session we surprisingly found out that he had already been familiar with the MarbleMania game. He had used the source code of this game for his own research. The other result was dropped because the participant delivered a version for task 2 that did not function properly. Further analysis revealed that this problem could not be easily fixed and that the code already stopped working with a change made after half an hour of work. So we decided to drop this data set.

	Task 1 / LaserGame	Task 2 / MarbleMania
Control Group	795	306
	183	62
	783	513
	1031	585
	90	0
	323	460
	1019	278
	394	519
	890	408
	784	480
	611	470
Experimental Group	533	499
	217	479
	1286	1080
	75	420
	726	109
	548	374
	460	338
	195	217
	353	493
	651	1115
	320	771

Fig. 12. Final scores for participants per task.

A 2 × 2 mixed factorial ANOVA was conducted with the task (LaserGame, MarbleMania) as within-groups variable and recovery support (with and without CoExist as additional support) as between-groups variable. Both the Shaphiro-Wilk normality test and Levene's test for homogeneity of variance were not significant ($p > .05$), complying with the assumption of the ANOVA test.

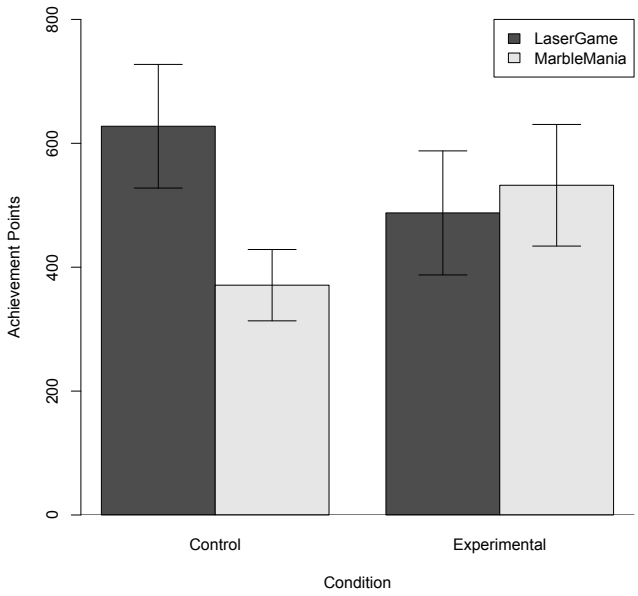

Fig. 13. A bar plot of the study results. Error bars represent the standard error of the mean

The bar plot in Fig. 13 illustrate that the control group scored on average less for the MarbleMania task than for the LaserGame task, while there is a slight increase in the performance of the experimental group. This indicates that improving MarbleMania was the more difficult task, and that the provision of CoExist helped to compensate for the additional difficulty.

Statistical significance tests were conducted from the perspective of null hypothesis significant testing with alpha = .05, and effect sizes were estimated using partial eta-squared, η_p^2. The results show a significant interaction effect between the effects of task and recovery support on the amount of achievement, $F(1, 20) = 5.49, p = .03, \eta_p^2 = .22$.

Simple main effects analysis revealed that participants in the control condition (with traditional tool support for both tasks) achieved significantly more for the LaserGame task than for the MarbleMania task, $F(1, 10) = 9.81, p = .01, \eta_p^2 = 0.5$, but there were no significant differences for participants in the treatment condition (with CoExist tools), $F(1, 10) = .2, p = .66, \eta_p^2 = .02$.

We performed correlation analyses to illuminate whether the amount of programming experience has an influence on the observed effects. However, there was no correlation between achievements and years of professional education & experience (starting with college education), Pearson's $r(20) = .1$, $p = .66$. Fur-

thermore, there was no correlation between gains in achievements (difference between points for MarbleMania and points for LaserGame) and years of professional education & experience, Pearson's $r(20) = .05, p = .83$.

The results suggest that the provision of additional recovery support such as CoExist has a positive effect on programming performance in explorative tasks.

5 Limitations

5.1 Order Effects / Counterbalancing

A possible objection to our study design is the lack of counterbalancing the treatment order, as there might be fatigue or learning effects. However, we think that there are complex dependencies between the order of the treatment and the dependent variable. If some participants had received the introduction and the tutorial to CoExist for task 1, which necessarily includes a description of its potential benefits, this would have likely changed how they approach the second task. In particular, they would have been more risk-taking than usual when not having such additional recovery support. So in order to reduce effects of fatigue, we split the study over two days. Also, the two tasks were significantly different, rendering each of them interesting and challenging in its own way.

5.2 Construct Validity

Care must be taken not to generalize from our treatment and measure. While we were motivated in this work by discussing recovery support in general, we compared only two levels in our study. Because of this, our results provide only little support that more recovery support is generally better with respect to all these other levels. Additional studies are required to better examine and support the general construct.

Also, the control and experimental group did not only differ in the fact, that one group could rely on CoExist in addition to standard tools for task 2. The members of the experimental group also ran through a tutorial that explains and motivates the CoExist tools. The tutorial or the fact of using a new tool might have contributed to the observed effect.

In addition, there are various kinds of social threats to construct validity such as hypothesis guessing or evaluation apprehension that need to be taken into account [15].

5.3 Reliability

We acknowledge the need for further reliability analyses on our measure. Additional studies are required to validate that our construct (the amount of required interactions to reproduce the achieved independent increments) is actually a measure for the amount of work that got done.

We also acknowledge the need for replicating both the coding of change histories, which is the identification of the independent increments, and determining the IDE interactions required for reproduction. Both steps were conducted by only one person, the first author of this article. As the analysis required approximately two to three full working weeks, we did not succeed in convincing another researcher to repeat the analysis.

5.4 Internal Validity

While we can observe a correlation between the treatment and the outcome, there might be factors other than the treatment causing or contributing to this effect. As we used a repeated measurement setup, we ruled out single group threats, but need to consider multiple group threats and social threats.

To the best of our knowledge, participants of the control and experimental group are comparable in so far as they experienced the time between both tasks similarly (selection-history threat), that they matured similarly (selection-maturation threat), and learned similarly from Task 1 (selection-testing threat).

However, there is a selection-mortality threat to the validity of our study, because we needed to drop the results of two participants who were both in the control group. But, on the other hand, we had no need to drop any results from the experimental group.

We also need to consider the selection-regression threat, because the average score of both groups is different. So it might be that one of the two groups scored particularly low or high, so that they can only get better or worse respectively. However, the lines in the interaction plot cross. This is an indicator that, besides other possible factors, the treatment is responsible for the observed differences in task 2. The results of the experimental group got better on average, while the results of the control group got worse on average. So, even if one group had a particularly high performance on task 1, the observed differences can hardly just be an artifact of selection-regression.

We dealt with social threats to internal validity, such as compensatory rivalry or resentful demoralization, by blinding participants to the treatment as much as possible.

5.5 External Validity

As we only recruited students for the study, the results are not necessarily representative for the entire population of programmers. However, we conducted correlation analyses to better understand the effect of experience on task performance and gained differences between tasks. The results show that there is no such correlation in the data of our study.

Our study was artificial in the sense that programmers may rarely spend two hours on improving source code only. It might be more typical that refactoring activities go hand in hand with other coding activities such as implementing new features or fixing bugs.

Furthermore, one might argue that refactoring a previously unknown codebase is also quite untypical. It might be more typical that programmers know a code base and also know their problems that need to get fixed. However, our study design focuses on objectively measuring and comparing programmers' performance.

6 Related Work

We previously presented CoExist and introduced the notion of preserving immediate access to intermediate development states [4]. Informal user studies indicated that programmers can identify a previous version of interest within a few seconds and that they appreciate the tools. Continuous versioning, as the basis of CoExist, closes a gap between the undo/redo feature of editors, which works on a more fine-grained level and handles files independently, and Version Control Systems such as Git, which require manual and explicit control. CoExist further builds on early work such as Orwell [16] and more recent work such as Delta Debugging [17], Continuous Testing [18], Changeboxes [19], SpyWare [20], Replay [21], and Juxtapose [22].

Continuous testing has been evaluated in a controlled experiment on student developers, showing that this approach helped participants to complete the assignment correctly [23]. CoExist improves on this approach by recording the test results and linking them to the corresponding changes, which allows for analyzing test results only when it is convenient. An empirical evaluation of Replay shows that a fine-grained version history and the possibility to replay changes reduce the time required to complete software evolution analysis tasks [21]. In particular, the possibility to replay changes can be considered a meaningful complement to our approach.

Delta Debugging automates the process of testing and refining hypotheses about why a programs fails by re-running an automated test and thereby narrowing down the delta that makes the test fail or pass. The dimension on which to narrow down the delta can be the input set provide to the program [24], but also a set of

changes between two versions [17]. CoExist supports the Delta Debugging approach along the change history well, because it preserves intermediate development states and provides an API to run code on these versions. Also, CoExist records tests results along the history.

Further discussions of related work concerning the technical concepts can be found in our original presentation of the CoExist approach [4].

7 Summary

We have presented an empirical evaluation of the benefits of CoExist over a traditional tool setting on programming performance in explorative tasks. CoExist represents additional recovery support that avoids the need for manually keeping recovery costs low. CoExist continuously versions the source code under development and provides immediate access to intermediate development states and information thereof.

22 participants ran through a lab study. Using a repeated measurement study, they were requested to improve the design of two games on two consecutive days. The experimental group could additionally rely on CoExist for the second task. Fine-grained change histories were recorded in the background, accumulating approximately 88 hours of recorded programming activities. We analyzed the change histories to identify independent increments and determined the required effort for reproducing them. This leads to scores that represent the amount of work achieved within the given time frame. Running an ANOVA tests shows a significant interaction effect, $F(1, 20) = 5.49$, $p = .03$, $\eta_p^2 = .22$, which suggests that additional recovery support such as provided with CoExist positively affects the programming performance in explorative tasks.

References

1. Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, 2004.
2. Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
3. Apache Software Foundation. *Subversion best practices*. 2009. [Online]. Available: http://svn.apache.org/repos/asf/subversion/trunk/doc/ user/svn-best-practices.html
4. Bastian Steinert, Damien Cassou, and Robert Hirschfeld. *Coexist: overcoming aversion to change*. In Proceedings of the 8th symposium on Dynamic languages, DLS '12, pages 107–118, New York, NY, USA, 2012. ACM.
5. Masaki Suwa ,Terry Purcell and John Gero. *Macroscopic analysis of design processes based on a scheme for coding designers' cognitive actions*. Design Studies, vol. 19, no. 4, 1998.

6. Masaki Suwa and Barbara Tversky. *External representations contribute to the dynamic construction of ideas*. In Diagrammatic Representation and Inference, volume 2317. Springer Berlin / Heidelberg, 2002.
7. Zafer Bilda and John S. Gero. *The impact of working memory limitations on the design process during conceptualization*. Design Studies, vol. 28, no. 4, 2007.
8. Jeanne Farrington. *Seven plus or minus two*. Performance Improvement Quarterly, vol. 23, no. 4, 2011.
9. David Kirsh. *Thinking with external representations*. Ai & Society, 25(4):441–454, 2010.
10. Donald A. Schon and Glenn Wiggins. *Kinds of seeing and their functions in designing*. Design Studies, vol. 13, no. 2, 1992.
11. Daniel Kahneman. *Thinking, fast and slow*. Macmillan, 2011.
12. Natalia Juristo and Ana M. Moreno. *Basics of software engineering experimentation*. Springer Publishing Company, Incorporated, 2010.
13. Steven P. Dow, Kate Heddleston, and Scott R. Klemmer. *The efficacy of prototyping under time constraints*. In Conference on Creativity and Cognition, 2009.
14. Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. *Parallel prototyping leads to better design results, more divergence, and increased self-efficacy*. ACM Transactions on Computer-Human Interaction (TOCHI) 17, no. 4 (2010): 18.
15. William R. Shadish, Thomas D. Cook, and Donald Thomas Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. Houghton Mifflin, 2002.
16. Dave Thomas and Kent Johnson. *Orwell—a configuration management system for team programming*. In ACM SIGPLAN Notices, vol. 23, no. 11, pp. 135-141. ACM, 1988.
17. Andreas Zeller. *Yesterday, my program worked. today, it does not. why?* In Software Engineering — ESEC/FSE '99, ser. Lecture Notes in Computer Science, O. Nierstrasz and M. Lemoine, Eds. Springer Berlin Heidelberg, 1999, vol. 1687, pp. 253–267.
18. David Saff and Michael D. Ernst. *Reducing wasted development time via continuous testing*. In ISSRE'03: International Symposium on Software Reliability Engineering, 2003.
19. Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. *Encapsulating and exploiting change with changeboxes*. In Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007, pp. 25-49. ACM, 2007.
20. Romain Robbes and Michele Lanza. *A change-based approach to software evolution*. Electronic Notes in Theoretical Computer Science 166 (2007): 93-109.
21. Lile Hattori, Marco D'Ambros, Michele Lanza, and Mircea Lungu. *Software evolution comprehension: Replay to the rescue*. In Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, pp. 161-170. IEEE, 2011.
22. Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. *Design as exploration: creating interface alternatives through parallel authoring and runtime tuning*. In Proceedings of the 21st annual ACM symposium on User interface software and technology, pp. 91-100. ACM, 2008.
23. David Saff and Michael D. Ernst. *An experimental evaluation of continuous testing during development*. ACM SIGSOFT Software Engineering Notes 29, no. 4 (2004): 76-85.
24. Andreas Zeller. *Isolating cause-effect chains from computer programs*. In Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, pp. 1-10. ACM, 2002.