How to Compare Performance in Program Design Activities: Towards an Empirical Evaluation of CoExist

Bastian Steinert and Robert Hirschfeld

Abstract We present the design of an empirical experiment to compare programmers' performance in program design tasks. The experiment is targeted to empirically examine the benefits of CoExist, a set of extensions to programming environments. CoExist supports programmers in dealing with unexpected and undesired consequences of making changes to their code base. Changing source code involves the risk of making errors. For example, a promising idea to simplify the code can suddenly turn out inappropriate, a situation that, if not prepared, requires programmers to manually withdraw recent changes. Traditionally, programmers have to strictly follow a structured and disciplined approach to reduce the costs of making errors. However, this traditional approach requires planning for upcoming but still uncertain changes in advance, which is time-consuming and also error prone. In addition, it requires significant effort to not forget the regular execution of the required activities, in particular in situations full of uncertainty. In contrast to this, CoExist offers dedicated tool support to recover fast and easily from undesired consequences. We believe that the presence of such tools encourages programmers to make source code changes at the moment they think of them, independent of whether or not the implications of such changes are already apparent. The presented experiment design to compare performance in program design tasks will help to examine this hypothesis.

1 Introduction

Programming involves more than continuously adding new lines of source code to a program. It requires reasoning about already written source code and making changes to it. Change is, for example, necessary to implement newly identified functionality or to fix erroneous behavior. Since programmers know that tomorrow

B. Steinert (✉) • R. Hirschfeld
Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
e-mail: bastian.steinert@hpi.uni-potsdam.de; robert.hirschfeld@hpi.uni-potsdam.de

they will have to make changes to the source code written today, they spend time on structuring it well and making it easy to understand, so that the modifications of tomorrow are easier to accomplish. In addition to the tasks of tomorrow, programmers also restructure their source code to ease the implementation of features of current interest (Beck and Andres 2004; Beck 1996; Fowler 1999).

Changing source code, however, involves risks because programmers can make errors. A promising idea to simplify the code can suddenly turn out inappropriate later in the process. Also, programmers can have flaws in their reasoning or can unintentionally ignore relevant aspects. Making an error represents a risk because it typically requires compensational activities that are time consuming and tedious to carry out.

The recommended way to deal with the risks of change is to anticipate that errors will be made and to continuously perform activities that keep compensation costs low. This includes checking for errors early and often as well as maintaining a safety net of stable development states one can fall back to (Beck and Andres 2004; Apache Software 2009). However, such precautionary activities can only reduce but not avoid the risk of tedious recovery work. Moreover, they can easily be ignored and distract from the actual task at hand.

We have developed an alternative way to deal with the risks involved in changing source code (Steinert et al. 2012). In contrast to asking for manual risk and cost reduction, we propose the provision of tool support such as CoExist that helps programmers deal with undesired consequences of their work. CoExist offers dedicated support to withdraw changes, to recover knowledge from previous development states, or to locate the cause of program failures even late in the process. In this way, CoExist avoids that making an error implies tedious recovery.

We believe that such a tool-based approach is preferable over a manual method based approach, which requires continuously following a set of practices. Research findings on design and cognition suggest that externalizing ideas supports the exploration of the problem and possible solutions. For example, creating prototypes help pursue a line of thought and discover unforeseen implications (Goldschmidt 1991; Lim et al. 2008). Other findings suggest that the creation of external representations inspires new associations (Kirsh 2010; Suwa and Tversky 2002). CoExist enables programmers to make changes as they think of them, because making errors does not imply additional costs.

We have designed an experiment to empirically examine our hypothesis that Co-Exist better supports programmers in program design activities, particularly in situations full of uncertainty. We have decided on a two by two mixed groups factorial design. We repeatedly measure subject performance in two different tasks, thereby having two groups of subjects, one using CoExist in addition to the regular tools for the second task. We measure performance by coding the changes and quantifying the effort for each category of change. In this report, we contribute the design and its justification for an experiment to empirically examine programmers' performance in program design activities.

Given a list of numbers: 3, 6, 1, 9, 10 ..., find all numbers smaller than 5.

```
givenNumbers:= "... a list of numbers ..."
result := OrderedCollection new.
1 to: givenNumbers size do: [:i | | eachNumber |
           eachNumber := givenNumbers at: i.
           eachNumber < 5
                ifTrue: [result add: eachNumber]]
```

```
givenNumbers := "... a list of numbers ..."
result := givenNumbers select: [:each | each < 5]
```

Fig. 1 Two different program snippets (in pseudo code) to fulfill the above stated need

2 Why Programming Involves Change or the Need for Well-Designed Programs

Programmers care about program qualities besides completeness and correctness. They care about a program's layout, its structure, or in which way a certain sub-goal is expressed. To illustrate this point, Fig. 1 shows two program snippets that lead to same observable effect (behavior) but are expressed in very different ways.

In the above example, as is often the case with design, there is hardly a right or wrong solution. Instead, the different solutions in the design space can be more or less appropriate for different purposes. While the first alternative includes more details about how the computation of the desired effect is accomplished, which can be important in certain domains, the second alternative focuses more on what should be achieved. It is also more concise. Programmers care about such design aspects because they know these will affect future programming activities. Appropriate source code design helps in two ways: it eases the implementation of new functionality and it supports maintenance and evolution.

2.1 How Source Code Design Affects Current Implementation Tasks

The amount of programming effort needed to implement a particular part of the problem depends on what has already been implemented. To illustrate this point, we build on the above shown program snippets (Fig. 1). The second solution can be so concise because some programmer has previously defined the program construct `select:`. And due to the availability of this construct, only little code is needed for programming needs similar to selecting a subset of numbers as shown in Fig. 2.

The decomposition of programs into modules helps to manage complexity (Blackwell 2002). It makes it possible to only be specific about details relevant and inherent to a particular concept, and to leave out the details of others. A proper

"Given a list of names, find all names matching 'jones'"

```
givenNames select: [:each | each matches: 'jones']
```

"Given a list of dates, find all dates before today"

```
givenDates select: [:each | each isBefore: Date today]
```

Fig. 2 Using `select:` to achieve similar needs in a concise manner

decomposition eases creating and comprehending the individual parts, and thus supports work on large complex systems (Parnas 1972).

However, how to decompose a program properly is not apparent from the beginning. The qualities of the current decomposition are revealed during the work on the program. The implementation of a particular aspect can be simple and straight- forward or rather complicated. The resulting source code can appear concise and right to the point or it can appear lengthy and disordered. Thus programmers often contemplate the source code and check whether it is easy enough to understand or still unnecessarily complex. They try to find elegant and simple solutions to express the concerns of the problem. Thereby, they can always introduce new modules (or programming constructs), such as selecting, or refining existing ones to better fit the current needs.

2.2 How Source Code Design Affects Program Maintenance and Evolution

The other reason why source code design matters is because programmers will have to (re-) understand formerly written source code and will have to adapt it. It is probably hard to find a piece of code that once written has never been changed afterwards. Programmers have to work on source code written by others as well as on source code they have written for some time. In both cases, they have to gain an in-depth understanding of the source code, either because they have never seen it before or because they cannot remember it in sufficient detail. In any case, the design of the source code can either facilitate or impede gaining a proper understanding of its meaning and effects.

The reason that programmers revisit previously written source code is because programming is a process of learning. It is not the same as assembling a car engine from a set of pre-defined parts, and it is also different from constructing a house according to a blueprint. Typically, every program is the first of its kind and the client does not have any type of a blueprint nor a meaningful description of the problem domain. Quite the contrary, programmers face a rather unstructured and little defined problem domain, which is not surprising considering that clients typically do not have the need to reason about their domain in a level of detail required for computer programs. In addition, the set of requirements often changes faster than the entire software solution can be accomplished.

To deal with these characteristics, software development often follows an incremental and iterative approach to eventually deal with all relevant aspects of the problem domain and to satisfy all needs. In an example development scenario, a client and the programmers sit together to identify the most important features for a small first version. The identified manageable set of features is realized without thinking much about further needs. Programmers may consult the client whenever questions arise during development. After releasing version 1, the client and the programmers talk about the desired functionality for version 2.

But for version 2, the client might want to extend the functionality implemented for version 1, which requires adapting and modifying the existing code to fulfill the extended requirements. Furthermore, the domain concepts implemented in successive program versions often depend on each other. It is likely that implementing functionality for a particular version builds on domain concepts that have already been implemented for former versions. But the existing code might only be partially sufficient and thus also requires adaptation or enhancement.

In addition to this macro level of software development, where clients and programmers collaborate to find out what needs to be built and in which order, programmers also have to deal with complexity and uncertainty on a micro level. Here, programmers are mainly concerned with how to built the defined features. Therefore, they follow a similar iterative and incremental approach. Programmers focus on a particular aspect of the problem, implement it, and thereafter consider another aspect, which possibly requires changing the code written for previous aspects.

On every level, the understanding of the problem domain co-evolves with the implementation (Dorst and Cross 2001). Seeing a first version of a solution improves the understanding of what the problem actually is and how it should be solved. However, as the problem understanding improves, so does the implementation. Thus, programmers will revisit previously written code and refine or adapt it to the improved understanding.

3 Why Changing Programs Involves Risks

While programmers regularly make changes to their programs, changing programs always involves the risk of creating *errors*. We distinguish *errors* from *mistakes*.

> ... a mistake is usually caused by poor judgment or a disregard of [known and understood] rules or principles, while an error implies an unintentional deviation from standards of accuracy or right conduct ... (Lindberg 2008)

Making an error refers to a situation where a programmers believes in the appropriateness of current and planned actions, and only later, after seeing the results, recognizes unexpected and undesired consequences. Making an error represents a risk because it often requires the programmer to accomplish some kind of tedious work to recover from it.

3.1 Risk Experienced, an Illustrative Story

To illustrate how changing programs involves risks, we would like to report the experience of a master's student. The student had been working on a visualization task using the Qt framework. At some point, he recognized that he had added several methods that all work on the same data. He decided to extract a class dedicated to this data structure and these methods. He created a new class called `SemanticLens` as a subclass of a Qt class `QEllipse`.

After moving all methods in this new class and adapting his code to make proper use of the new class, he contemplated his code and became skeptical about the decision to subclass the Qt class. He remembered that subclassing has the drawback of exposing the interface of the superclass to all clients who might make use of it, thereby creating a dependency that can become difficult during maintenance tasks. So, he decided to go for the delegation pattern instead. He was sure that delegation is the right way to go. So, the student changed the superclass of the `SemanticLense` class, added a field and accessor-methods to maintain a reference to a `QEllipse` object and also added initialization code. He changed the methods in `SemanticLense` class and made the required changes in the code using this class.

However, while looking at the result of making all these changes, the student realized that his assumption had been wrong. He could now see that subclassing is preferable over delegation in this situation because having access to the methods of the super-class is actually useful in his program. As a consequence of this insight, the student now faced the laborious task of manually withdrawing all the changes previously made to replace subclassing with delegation. He had to identify the relevant artifacts (files), and for each file to apply the undo command an undefined number of times until reaching the desired state. Such tasks are not only time-consuming but also tedious. Assuming that the changes made for the initial replacement had taken several minutes, this meant that manually withdrawing also took a few minutes. The required recovery work would have been even more laborious if the student had made further changes before recognizing the error. Such situations easily lead to irritation and are those programmers want to avoid.

One might argue that the student behaved incorrectly in this situation. He could have finished implementing the functionality first before considering refactoring the code. However, this could have led to more code needing adaptation later on. One could also argue that the student could have made a local commit (a checkpoint) before starting the refactoring, so that there would have been an easy way back. However, the code was in an immediate state and his work was not yet finished. He also has the very typical habit of thinking about commits only on completion of a task.

One could also argue, that he should have thought more carefully about his ideas before making any changes. Analyzing his idea in more depth might have been enough to raise doubts. However, it is unclear how much deliberation is necessary to avoid such errors. Moreover, too much thinking and questioning can easily lead to counterproductivity.

In contrast to the idea of careful upfront thinking, research findings on design and cognition suggest that externalizing ideas supports the exploration of the problem and possible solutions. For example, creating prototypes help pursue a line of thought and discover unforeseen implications (Goldschmidt 1991; Lim et al. 2008). Other findings suggest that the creation of external representations inspires new associations (Kirsh 2010; Suwa and Tversky 2002). While these results suggest that programmers should support their thinking by doing, this will inevitably imply the constitution of errors.

3.2 The Risks of Change and Methods to Reduce It

The issues in changing source code are broader and more general than illustrated by the story of the master's student balancing the pros and cons of delegation and subclassing. Writing and changing source code always involves the following risks:

- A promising idea unexpectedly turns out inappropriate and the programmer wants to continue exploring a previous idea, but many parts of the source code have already been modified.
- The improvement to one part of the program seems to affect the overall program behavior in unexpected ways, but the programmer has difficulties to find out which of the recent changes is causing the undesired behavior.
- The source code under current improvement turns out to be more complex than the programmer expected and it is unclear how the code was previously working.
- Recent changes turn out to represent multiple independent improvements that should be shared in separate increments.

The above listed issues are all well known. Literature describes them in detail, teachers tell students about them, and every programmer has experienced them (and still does) in some form or another. To reduce the risk of encountering such situations, literature recommends following a structured and disciplined approach and employing certain practices of work, which are, for example:

- To only work on one thing at a time, specifically a task or issue that you understand in detail. Therefore, to break larger task items down into smaller ones. This encourages staying on track and simplifies sharing your improvements with others.
- Make sure you understand the items you work on, if not, consider breaking down items into manageable parts, or consider consciously deciding on a phase of experimenting that should be preceded by committing (and/or branching).
- Write tests and run them often and regularly, at best, after every small change. This helps recognize bugs early in the process, and helps to pin down the cause of the problem to a few recent changes.
- Employ a distributed version control system such as *Git* or *Mercurial* and make frequent use of it by regularly committing small increments, which enables going back to a stable state easily.

The general pattern of these practices follows the contention that programmers should anticipate that they will make errors and thus should perform precautionary activities continuously and regularly in order to keep possible recovery costs low.

However, as the story about the student from above illustrates, the recommended way of relying on best practices does not seem sufficient in all circumstances. There are multiple reasons why this approach can fail in avoiding a need for tedious recovery activities:

- Wrong assessment. As was the case with the student in the story above, programmers can wrongly assess a programming situation. Applying best practices requires interpretation and subjective judgment. While running tests *often* or working on only *one thing* at a time are recommended, these are only guiding rules, and the programmer has to decide what *often* means and what the appropriate granularity of *things* is. Whenever a programmer feels confident about an idea and is unaware of any risks, the possibility remains that the assessment is wrong. In this case, the programmer will be unprepared for errors and will have to recover from them.
- Additional workload. Applying best practices is a continuous effort in addition to the work on the problem domain. As such, it is easy to forget and to ignore, in particular when one is caught up in creativity. Furthermore, mustering the needed discipline and remembering “not to forget” require significant mental effort (Allen 2001), even so practice helps reduce the required amount of attention.
- Upfront thinking. Following the recommended path requires programmers to structure the work ahead of them. This is a consequence of the need for interpretation and value judgment. For example, the practice to work on only one thing at a time requires regular reflection about current and upcoming work and assessing whether it should still be considered as “one thing” (a logical unit of work). Becoming aware of potential future risks requires thinking about the situation without working on it.

These limitations show that programmers will arrive, every now and then, in a situation where they have to face tedious work to recover from a previously made error, like the student in the story above.

4 CoExist: Tools to Encourage Change by Avoiding Risks

We have developed CoExist, an extension to the programming environment Squeak/Smalltalk, to preserve previous development states and to provide immediate access to relevant information (Steinert et al. 2012). CoExist is based on the key insight that the risks are caused by the loss of information during the process of change. With every change, we lose a previous version, unless it is saved explicitly. This version, however, can be of value in future development states, when, for example, an idea turns out inappropriate.

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. To make the user aware of this background

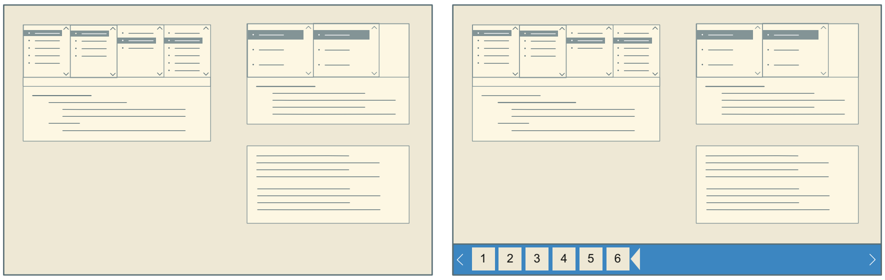


Fig. 3 The user interface of a regular Squeak/Smalltalk programming environment (on the *left*), and the version bar element added in CoExist (on the *right*)

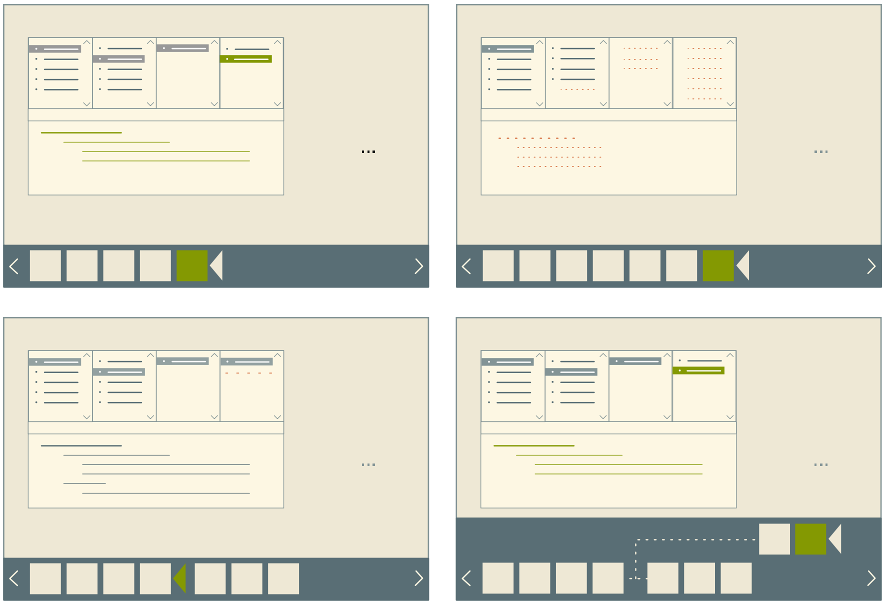
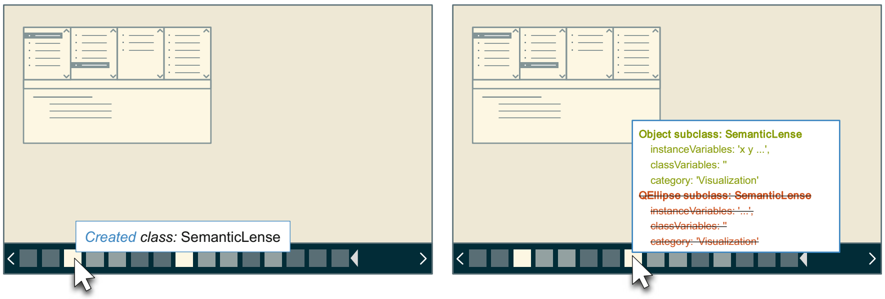


Fig. 4 (From *top left* to *top right*) A programmer modifies source code which implicitly creates items in the version bar. The programmer decides to withdraw several changes by going back to a previous version (*bottom left*), and continues working and creating new changes, which will appear on a new implicitly created branch (*bottom right*)

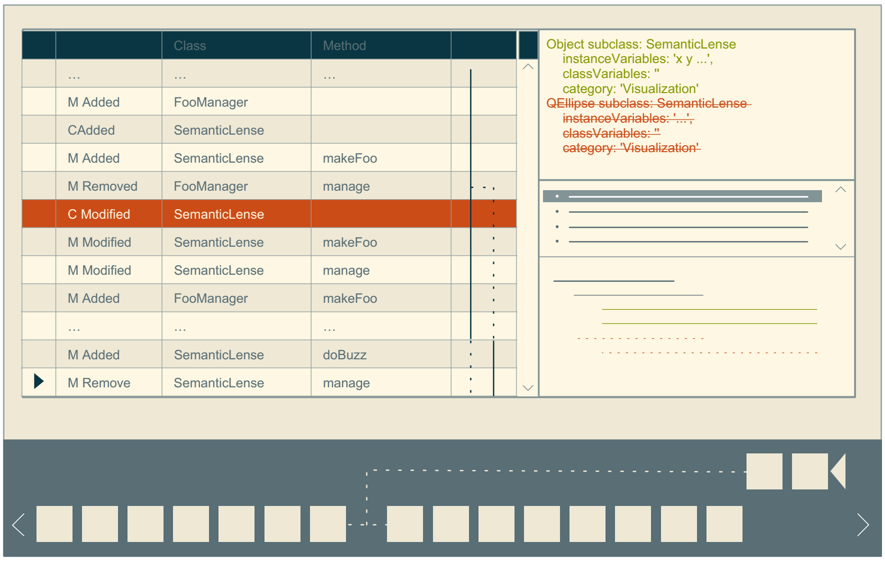
versioning and to allow for selecting previous versions, we have added a version bar (timeline) to the user interface of the programming environment (Fig. 3).

By continuously preserving intermediate development states, CoExist makes it easy for programmers to go back to a previous development state and to start over as shown in Fig. 4. Starting over from a former development state will implicitly create a new branch of versions. This preserves the changes the programmers want to withdraw, as they might be of use later on.


Created class: SemanticLense

```
Object subclass: SemanticLense
    instanceVariables: 'x y ...',
    classVariables: ''
    category: 'Visualization'
~~QEllipse subclass: SemanticLense~~
    ~~instanceVariables: '...',~~
    ~~classVariables: ''~~
    ~~category: 'Visualization'~~
```

Fig. 5 Hovering shows which source code element has been changed (*left*). Holding shift in addition shows the full difference to the previous version (*right*)



		Class	Method
	...	...	...
	M Added	FooManager	
	CAdded	SemanticLense	
	M Added	SemanticLense	makeFoo
	M Removed	FooManager	manage
	C Modified	SemanticLense	
	M Modified	SemanticLense	makeFoo
	M Modified	SemanticLense	manage
	M Added	FooManager	makeFoo
	...	...	...
	M Added	SemanticLense	doBuzz
▶	M Remove	SemanticLense	manage

```
Object subclass: SemanticLense
    instanceVariables: 'x y ...',
    classVariables: ''
    category: 'Visualization'
~~QEllipse subclass: SemanticLense~~
    ~~instanceVariables: '...',~~
    ~~classVariables: ''~~
    ~~category: 'Visualization'~~
```

Fig. 6 The version browser provides a tabular view on change history. Selecting a row shows corresponding diff information in the panes to the right

CoExist provides two mechanisms dedicated to support programmers in identifying previous versions of interest. First, programmers can use the version bar, which will highlight version items that match to the currently selected source code element (Fig. 5). Hovering the items will display additional information such as the kind of modification, the affected elements, or the actual change performed.

Second, programmers can use the version browser to explore information of multiple versions at a glance. The version browser shown in Fig. 6 displays basic version information in a table view, which allows a fast scanning of the history for source code elements of interests.

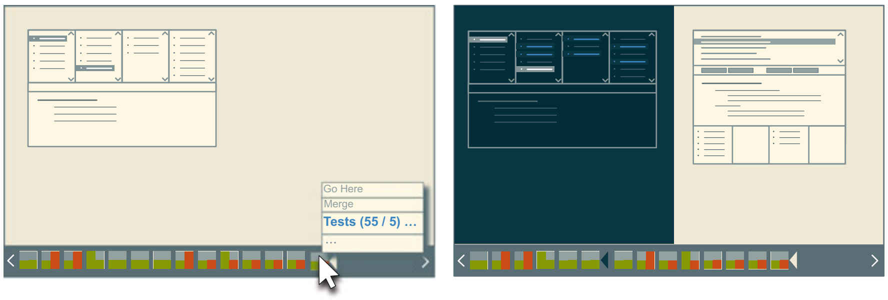


Fig. 7 The items in the version bar are now a visualization of the results of the tests that have been run in the background (*left*). A second inner environment allows the user to explore a previous version next to the current one (*right*)

The versioning facilities of CoExist also allows continuously running analysis on every newly created version. In particular, it supports running 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 (Fig. 7). This makes the effect of each change regarding test quality visible. The user can also run other analysis such as performance measurements. CoExist provides full access to version objects and a programming interface to run code on them. Programmers can thus focus on their task at hand and, when necessary later on, can analyze the impact of each change.

When in the course of change programmers suddenly become curious about how certain parts of the source code looked previously or how certain effects were achieved, they can open a previous version in a separate working environment as shown on the right in Fig. 7. They can browse and explore the source code of a previous version and compare it to the current development version. In addition, they can also run and debug programs in this additional working environment.

With that, CoExist enables efficiently recovering knowledge from the previous version. This 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 because they can rely on dedicated tools that help with whatever their explorations will reveal. They no longer have to follow certain best practices to avoid undesired consequences of changing code.

5 Experiment Design

We hypothesize that programmers making use of CoExist will perform better in program design tasks that involve a strong degree of uncertainty. To gain empirical support for this claim, we have designed a controlled experiment. We have decided for a repeated measure factorial design, in which subjects are instructed to *improve* the source code design of given programs as best as possible in the time frame of 2 h.

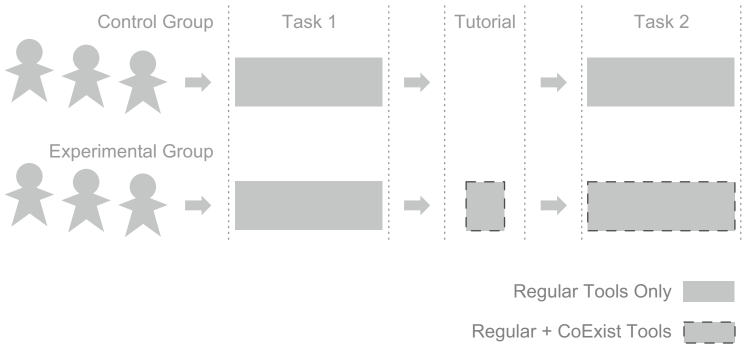


Fig. 8 Our experiment setup to compare performance in program design activities

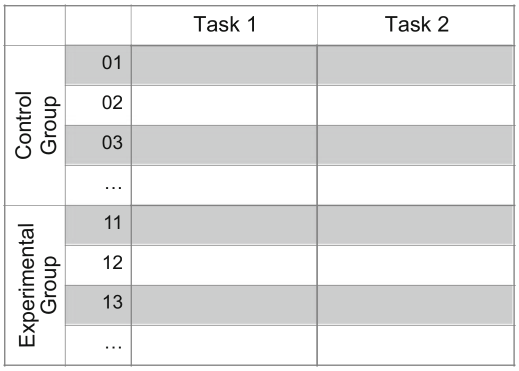
5.1 Repeated Measure Factorial Design

Figure 8 illustrates the setup of our controlled experiment. Subjects are assigned to either of two conditions, the control group or the experimental group. Subjects in the control group use the regular development tools for both tasks. Subjects in the experimental group use regular development tools only for Task 1. After that, they receive a tutorial in CoExist and have time to try it out and experience its benefits and limitations. Thereafter, subjects in the experimental group work on Task 2 and can therefore make use of CoExist in addition to the regular tool support. So while for Task 1 all subjects use regular tools only, subjects in the experimental group may use CoExist for Task 2. After receiving task instructions and material, subjects had two hours for working on the respective task and achieving as much improvement as possible.

At the time of writing, 20 students have already participated in the experiment. They worked on both tasks on two different but subsequent days. Both tasks are scheduled for the same time of the day to ensure similar working conditions (hours past after waking up, hours already spent for work or studies, . . .). Typically, we scheduled the task assignments after lunch so that for day 2, there was time left to run the CoExist tutorial session upfront (before lunch time).

We try to keep subjects unaware of their assignment to the conditions, which worked well for day/task 1. However, on day 2, subjects in the experimental group could guess that they received special treatment because they were introduced to CoExist and were asked to make use of it. Nevertheless, subjects in the control group were unaware of the experimental treatment. They did not know that subjects of the experimental group run through a tutorial and can use CoExist for task 2. Hence, subjects were not entirely blind concerning the treatment, although we tried to be as close as possible to the blind setting.

This setup gives us two measures for every subject which will be prepared in tables such as shown in Fig. 9. Such a data collection allows tests for statistical differences between task 1 and task 2 as well as between the control and the



		Task 1	Task 2
Control Group	01		
	02		
	03		
	…		
Experimental Group	11		
	12		
	13		
	…		

Fig. 9 Data collection form corresponding to the experiment setup

experimental group. It also allows tests for the existence of an interaction effect of the two factors, which will indicate whether or not CoExist has an effect on programmers' performance.

5.2 Task: "Improve"

On each of the two days, subjects work on a different computer programs, but the task is the same. Participants are requested to improve the design of the source code. The two different programs are relatively small computer games like Tetris.

The procedure for each day is as follows. At the beginning of the assignment, subjects are introduced to the game. After introducing the game play subjects have a few minutes to play the game and get familiar with it. Afterwards subjects receive the assignment. The task is 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 2 h. To help understand the intent of the task, we provided sample descriptions of possible improvements, for example:

- Extract methods to shorten and simplify overly long and complicated methods
- Replace conditional branching by polymorphism
- Detect and remove unnecessary conditions or parameters

Subjects were told to imagine that they co-authored the code and are responsible for it, and that they now have time dedicated to improve the code in order to make future development tasks simpler (enhancements or maintenance).

For both tasks, the given program was a relatively simple single player computer games. Figure 10 shows screenshots of the game for Task 1 and Task 2. The game on the left is called LaserGame. The player's goal is to place mirrors in the field so

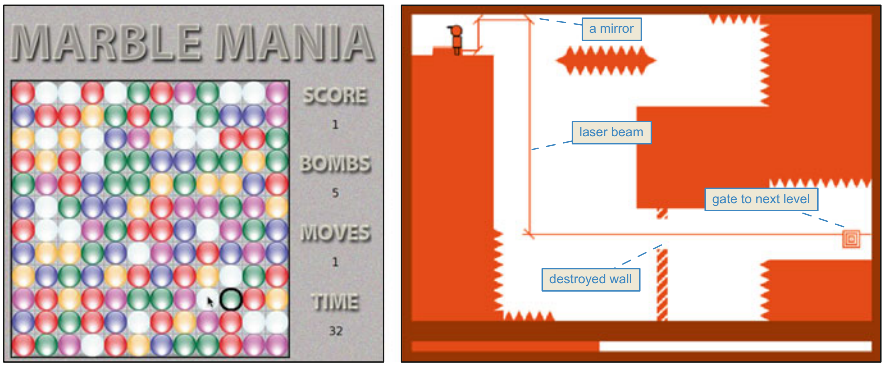


Fig. 10 Screenshots of the games used as experimental unit: MarbleMania (*left*), LaserGame (*right*)

that a laser is redirected properly to destroy the wall that blocks the way to the gate to the next level. The game on the right is called MarbleMania. The user has to switch neighbored marbles to create one or more sequences of at least 3 marbles that have the same color, either by row or by column.

Subjects are also asked to describe their improvements. They should imagine themselves as part of a development team. The description should help other team members to better understand their changes and how they improve the code.

We decided to use these small games as the experimental unit primarily due to practical reasons, but this choice turned out to be useful in unexpected ways. Such games are developed by students in one of our undergraduate courses. Over the years we have gained an interesting repertoire of game, all having different characteristics and qualities. Both of these games function properly and have a simple but still fun game play. So, only little time is required to get familiar with the program's functionality. Both games leave significant room for improvement in terms of the source code, as do most of the games developed in this course. Furthermore, both games come with a set of tests cases, which also have been developed by the respective students. Test cases are particularly useful when existing source code has to be changed because they are a means to automatically check whether selected aspects of the program still work as expected. However, while the offered test cases are useful, they are not sufficient. Manual testing of the games is necessary, as it is often the case in other projects.

Independent of these characteristics, using the games as the subject of work has another benefit. Participants of the study are primarily HPI students. And since they had to take our class, they all have gained similar experiences in developing such games. They have similar knowledge about the technologies, frameworks, and libraries used for developing these kind of games. Consequently, experience is not a factor that varies greatly among the participants.

5.3 Design Justification

For our experiment, we decided to keep the time factor fixed and measure the amount of achieved improvements. Keeping the time fixed is one of two typical ways to examine the effects of tools or methods. The other way is to fix the amount of work to get done by providing a clear unambiguous goal and measure the time needed to achieve this goal (Juristo and Moreno 2010).

A fixed time setup seems preferable for experiments that focus on design tasks, mainly because uncertainty is inherent to design tasks. The setup that measures time to completion requires a clearly defined task without any uncertainty or ambiguity. There has to be clear indicator when the task is finished. Also, the task description should provoke similar thoughts, so that all subjects have the same idea what the goal is and how it should be approached (given the defined conditions). These criteria can hardly be met in an experiment where participants should accomplish a design task.

Related research also shows that fixed time setup is a typical choice. In (Dow et al. 2009, 2010), for example, the authors report on the empirical examination of prototyping techniques. The response variable (dependent measure) has always been some form of quality criteria of the design outcome while participant have had a fixed amount of time to create the best possible design. However, we are unaware of an experiment report in the software engineering field that examines an effect in program design tasks.

Besides the decision for the fixed time setup, we have decided to rely on a repeated measurement experiment design in contrast to other options. A repeated measurement setup is preferable because programmers strongly vary in approaching such tasks. Programmers have different working speeds, which involves code comprehension, code writing (typing speed), but also tool usage. Furthermore, when programmers face the task to spend a fixed (and relatively short) amount of time on improving source code, some programmers will have the tendency to focus on the various small issues in the code, which are probably also easy to fix, while other programmers will have the tendency to try to identify major flaws in the overall program structure of the code and to improve on that while ignoring smaller issues. While there can be significant differences, different contributions can be similarly important for the long term success of a software project. However, the difference in the response variable between programmers can be large in relation to the possible difference caused by the provoked variations. For still being able to discover statistical effects in these circumstances, literature recommends repeated measurement experiment setups, in particular when having limited access to subject candidates (Juristo and Moreno 2010).

5.4 Analysis: Coding Changes

Besides a meaningful setup that allows for comparing performance, the experiment requires a meaningful quantitative response variable (dependent variable). Therefore, we have operationalized the notion that programmers can spend more or less time on contemplating source code and testing their ideas mentally before actually making the changes.

We assume that CoExist encourages participants to make changes as they think of them because they can always recover from undesired consequence easily. We also assume that without CoExist, participants will spend more time on thinking about their ideas before making changes to the code in order to avoid making errors.

In the context of the experiment, we believe that CoExist enables programmers to achieve more improvements in the given time frame. Every thought about a possible improvement, no matter how vague and uncertain it still may be, can either turn out appropriate or inappropriate. If the vague idea turns out to be appropriate, programmers who directly started making changes will clearly save time because they avoid spending time on upfront thinking. Findings in design research suggest that given a design task full of uncertainty, programmers who directly make the changes will save time in case that the idea turns out to be inappropriate. By making the changes, they support their thinking by doing and explore their ideas and their limits more efficiently. CoExist will help to recover fast and to get back to a desired development state.

While we assume that participants using CoExist will make more changes, the pure number of changes made, which correlates with number of versions, is not a meaningful proxy for the amount of achieved improvements. This is for various reasons:

- Changes that lead to new versions strongly vary in the amount of changed code and in the effect they have. While adding leading whitespace is a small change to the code, which likely has no effect on the program execution, removing statements or parameters from a method is a much larger change, which almost always will has a effect on the program execution.
- Participants might want to withdraw changes. In an extreme case, a subject might spend an hour of work on a particular idea to recognize later that the idea cannot work out properly. This will create many versions in the history which cannot be counted as improvements. Moreover, when not using CoExist, participants have to manually withdraw their made changes, which will in turn create additional changes.
- It might also be the case, that a series of changes was only good for inspiration and helped the programmer to develop a better idea of how to improve the elements of current interest. In this case, it would be unfair to double count the made changes, the changes made for the initial idea and also changes made for the final improvement.

Facing these constraints, a meaningful way to quantify the amount of achieved improvements is to code the changes that persist over time, which means to identify groups of related changes and assign them to categories. Such change categories can be either generic or rather specific to the given program.

Generic improvements are for example: to rename an instance variable; to replace a parameter with method; to make use of cascades; to inline a temporary expression; to replace magic string/number with method.

Improvements specific to the MarbleMania Game are, for example: to replace the dictionary holding "exchange state" with instance variables; to replace `isNil` checks in the destroyer with *null objects*; to remove button clicked event handling indirections.

To perform the coding, we have analyzed the data in two steps. In a first step, we have listed the timestamps of all versions (in a column of a spreadsheet) and separated them according to commits, with subjects made during the task. In the second column, we added the respective commit message (illustrated in Table 1). The commit messages help understand the intent of the changes, which gives the required context to understand the small changes to the individual source code elements. In a second step, we identified improvements that we assigned to a category. Thereby, a coded improvement can consist of only one actual change or it can involve many changes. Sometimes, all the changes made for one commit contribute to one coded improvement.

These change categories are assigned numbers that represent the relative effort required to implement them. We sum up these numbers for the coded improvements to finally get a measure of the amount of achieved improvements, which can be used to run the statistical tests.

6 Summary

Programmers spend time on source code design to better support current and future programming activities. While working on their code base, programmers can make errors. Making errors represents a risk because it often requires tedious and time-consuming recovery work. Traditionally, programmers have to anticipate such situations to keep the costs for recovery low. However, anticipating errors is not always possible, as we have illustrated by means of an experience report. We have also described that precautionary activities can easily be ignored and distract from the actual work.

We have presented CoExist as an extension to programming environments. CoExist has been designed to encourage change. It avoids the risk of error making by providing dedicated support to recover from undesired consequences. We believe that such a tool-based approach is preferable because it encourages programmers to support their thinking by doing. We have presented an experiment design to empirically examine this claim. We measure subject performance in two different tasks. In each task, participants have to improve the source code design of a given program. Their changes are recorded. Afterwards, the changes are coded to identify achieved improvements and to compute a quantitative measure of programmers' performance.

Table 1 Spreadsheet Excerpt with coded version data

Version timestamp	Commit message of participants	Coded improvements
. . .		
14:01:41	In LaserBeam extracted code that is similar in all these calculate methods; improved the previously extracted, generic calculateWay: . . . method. (**two things happened** – refactoring + additional improvements) . . . "Refactoring: summarized multiple similar methods into one, with 3 parameters. Original methods call the new one."	LG_ExtractGenericCalculateMethod (ReplaceSimilarStatementsWithCallToExtractedMethod)
14:02:16		
14:02:32		
14:02:49		
14:03:01		
14:03:05		
14:03:11		
14:03:16		
14:06:08		
14:06:42		
14:06:46		
14:13:06	Simplified LaserBeam>>#setStartPoint, deleted useless condition, integrated code from called methods, and removed the other methods. "Simplified method based on detected 'invariant', that self laser direction is always 1@0 at this code location. Removed 2 methods."	RemoveStatements + 2 * InlineMethod
14:14:18		
14:14:25		
14:15:16		
14:15:28		
14:15:38		
14:15:42		
14:17:25		
14:18:39	"Removed 2 unused variables of SWA18Laser. One seemingly was not used at all (point), the other (direction) got useless after previous commit. Removed all usage of the direction-variable (was only used in tests)."	2*RemoveStatement + 2*RemoveUnusedMethod + 2*RemoveUnusedInstVar
14:18:43		
14:18:53		
14:18:53		
14:18:53		
14:19:41		
14:20:33		
14:21:02		

14:25:02	"Extracted method in level parsing." #readNumberArrayFrom:	LG_LevelLoader_ExtractSimilarStatements
14:26:00		
14:26:08		
14:26:26		
14:26:53		
14:27:15		
14:27:20		
14:27:37		
14:27:50		TmpVarRenaming
14:29:20		TmpVarRenaming
14:29:29	"Removed useless method" – belongs to previous context	InlineMethod
14:30:18	"Removed another useless method" – namely #readTimeFrom: – integrated the more meaningful one-liner in the caller	Recategorization
14:30:59		InlineMethod
14:31:11		
. . .		

References

Allen D (2001) Getting things done: the art of stress-free productivity. Penguin, New York

Apache Software Foundation (2009) Subversion best practices

Beck K (1996) Smalltalk best practice patterns. Prentice Hall

Beck K, Andres C (2004) Extreme programming explained: embrace change. Addison-Wesley Longman

Blackwell AF (2002) What is programming. In: 14th workshop of the Psychology of Programming Interest Group. Citeseer, pp 204–218

Dorst K, Cross N (2001) Creativity in the design process: co-evolution of problem- solution. Des Stud 22(5)

Dow SP, Heddleston K, Klemmer SR (2009) The efficacy of prototyping under time constraints. In: Conference on creativity and cognition

Dow SP, Glassco A, Kass J, Schwarz M, Schwartz DL, Klemmer SR (2010) Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. ACM Trans Comput-Hum Interact (TOCHI) 17(4):18

Fowler M (1999) Refactoring: improving the design of existing code. Addison-Wesley Professional, Reading

Goldschmidt G (1991) The dialectics of sketching. Creativity Res J 4(2)

Juristo N, Moreno AM (2010) Basics of software engineering experimentation. Springer

Kirsh D (2010) Thinking with external representations. Ai Soc 25(4):441–454

Lim Y-K, Stolterman E, Tenenberg J (2008) The anatomy of prototypes: prototypes as filters, prototypes as manifestations of design ideas. ACM Trans Comput-Hum Interact (TOCHI) 15(2)

Lindberg CA (2008) Oxford American writer's thesaurus. Oxford University Press, New York

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058

Steinert B, Cassou D, Hirschfeld R (2012) Coexist: overcoming aversion to change. In: Proceedings of the 8th symposium on dynamic languages, DLS'12, New York, ACM, pp 107–118

Suwa M, Tversky B (2002) External representations contribute to the dynamic construction of ideas. In: Diagrammatic representation and inference, vol 2317. Springer Berlin/Heidelberg