Built-in Recovery Support for Explorative Programming: Preserving Immediate Access To Static and Dynamic Information of Intermediate Development States

Dissertation
zur Erlangung des akademischen Grades
"doctor rerum naturalium"

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Bastian Steinert

Betreuer:

Prof. Dr. Robert Hirschfeld
Fachgebiet Software-Architekturen
Hasso-Plattner-Institut
Universität Potsdam

Gutachter:

Richard P. Gabriel, PhD, MFA
IBM Research - Almaden

Prof. Dr. Ralf Lämmel
Fachgruppe Software-Sprachen
Institut für Softwaretechnik
Universität Koblenz-Landau

This work is licensed under a Creative Commons License:
Attribution – NonCommercial – NoDerivatives 4.0 International
To view a copy of this license visit
http://creativecommons.org/licenses/by-nc-nd/4.0/

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2014/7130/
URN urn:nbn:de:kobv:517-opus-71305
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-71305

ABSTRACT

This work introduces concepts and corresponding tool support to enable a complementary approach in dealing with recovery. Programmers need to recover a development state, or a part thereof, when previously made changes reveal undesired implications. However, when the need arises suddenly and unexpectedly, recovery often involves expensive and tedious work. To avoid tedious work, literature recommends keeping away from unexpected recovery demands by following a structured and disciplined approach, which consists of the application of various best practices including working only on one thing at a time, performing small steps, as well as making proper use of versioning and testing tools.

However, the attempt to avoid unexpected recovery is both time-consuming and error-prone. On the one hand, it requires disproportionate effort to minimize the risk of unexpected situations. On the other hand, applying recommended practices selectively, which saves time, can hardly avoid recovery. In addition, the constant need for foresight and self-control has unfavorable implications. It is exhaustive and impedes creative problem solving.

This work proposes to make recovery fast and easy and introduces corresponding support called CoExist. Such dedicated support turns situations of unanticipated recovery from tedious experiences into pleasant ones. It makes recovery fast and easy to accomplish, even if explicit commits are unavailable or tests have been ignored for some time. When mistakes and unexpected insights are no longer associated with tedious corrective actions, programmers are encouraged to change source code as a means to reason about it, as opposed to making changes only after structuring and evaluating them mentally.

This work further reports on an implementation of the proposed tool support in the Squeak/Smalltalk development environment. The development of the tools has been accompanied by regular performance and usability tests. In addition, this work investigates whether the proposed tools affect programmers' performance. In a controlled lab study, 22 participants improved the design of two different applications. Using a repeated measurement setup, the study examined the effect of providing CoExist on programming performance. The result of analyzing 88 hours of programming suggests that built-in re-

covery support as provided with CoExist positively has a positive effect on programming performance in explorative programming tasks.

ZUSAMMENFASSUNG

Diese Arbeit präsentiert Konzepte und die zugehörige Werkzeugunterstützung um einen komplementären Umgang mit Wiederherstellungsbedürfnissen zu ermöglichen. Programmierer haben Bedarf zur Wiederherstellung eines früheren Entwicklungszustandes oder Teils davon, wenn ihre Änderungen ungewünschte Implikationen aufzeigen. Wenn dieser Bedarf plötzlich und unerwartet auftritt, dann ist die notwendige Wiederherstellungsarbeit häufig mühsam und aufwendig. Zur Vermeidung mühsamer Arbeit empfiehlt die Literatur die Vermeidung von unerwarteten Wiederherstellungsbedürfnissen durch einen strukturierten und disziplinierten Programmieransatz, welcher die Verwendung verschiedener bewährter Praktiken vorsieht. Diese Praktiken sind zum Beispiel: nur an einer Sache gleichzeitig zu arbeiten, immer nur kleine Schritte auszuführen, aber auch der sachgemäße Einsatz von Versionskontroll- und Testwerkzeugen.

Jedoch ist der Versuch des Abwendens unerwarteter Wiederherstellungsbedürfnisse sowohl zeitintensiv als auch fehleranfällig. Einerseits erfordert es unverhältnismäßig hohen Aufwand, das Risiko des Eintretens unerwarteter Situationen auf ein Minimum zu reduzieren. Andererseits ist eine zeitsparende selektive Ausführung der empfohlenen Praktiken kaum hinreichend, um Wiederherstellungssituationen zu vermeiden. Zudem bringt die ständige Notwendigkeit an Voraussicht und Selbstkontrolle Nachteile mit sich. Dies ist ermüdend und erschwert das kreative Problemlösen.

Diese Arbeit schlägt vor, Wiederherstellungsaufgaben zu vereinfachen und beschleunigen, und stellt entsprechende Werkzeugunterstützung namens CoExist vor. Solche zielgerichtete Werkzeugunterstützung macht aus unvorhergesehenen mühsamen Wiederherstellungssituationen eine konstruktive Erfahrung. Damit ist Wiederherstellung auch dann leicht und schnell durchzuführen, wenn explizit gespeicherte Zwischenstände fehlen oder die Tests für einige Zeit ignoriert wurden. Wenn Fehler und unerwartete Einsichten nicht länger mit mühsamen Schadensersatz verbunden sind, fühlen sich Programmierer eher dazu ermutig, Quelltext zu ändern, um dabei darüber zu reflektieren, und nehmen nicht erst dann Änderungen vor, wenn sie diese gedanklich strukturiert und evaluiert haben.

Diese Arbeit berichtet weiterhin von einer Implementierung der vorgeschlagenen Werkzeugunterstützung in der Squeak/Smalltalk Entwicklungsumgebung. Regelmäßige Tests von Laufzeitverhalten und Benutzbarkeit begleiteten die Entwicklung. Zudem prüft die Arbeit, ob sich die Verwendung der vorgeschlagenen Werkzeuge auf die Leistung der Programmierer auswirkt. In einem kontrollierten Experiment, verbesserten 22 Teilnehmer den Aufbau von zwei verschiedenen Anwendungen. Unter der Verwendung einer Versuchsanordnung mit wiederholter Messung, ermittelte die Studie die Auswirkung von CoExist auf die Programmierleistung. Das Ergebnis der Analyse von 88 Programmierstunden deutet darauf hin, dass sich eingebaute Werkzeugunterstützung für Wiederherstellung, wie sie mit CoExist bereitgestellt wird, positiv bei der Bearbeitung von unstrukturierten ergebnisoffenen Programmieraufgaben auswirkt.

PUBLICATIONS

JOURNAL PUBLICATIONS

- Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Science of Computer Programming*, 76 (12):1194–1209, 2011

CONFERENCE PUBLICATIONS

- 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. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384591. URL http://doi.acm.org/10.1145/2384577.2384591
- Marcel Taeumel, Bastian Steinert, and Robert Hirschfeld. The vivide programming environment: Connecting run-time information with programmers' system knowledge. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '12, pages 117–126, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1562-3. doi: 10.1145/2384592.2384604. URL http://doi.acm.org/10.1145/2384592.2384604
- Michael Perscheid, Batian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt. Immediacy through interactivity: Online analysis of run-time behavior. In *WCRE'10: Proceedings of the 17th Working Conference on Reverse Engineering*, volume 10, pages 77–86, Beverly, MA, USA, 2010. IEEE Computer Society. doi: 10.1109/WCRE.2010.17
- Bastian Steinert, Michael Haupt, Robert Krahn, and Robert Hirschfeld. Continuous selective testing. In *Agile Processes in Software Engineering and Extreme Programming*, pages 132–146. Springer, 2010

- Bastian Steinert, Marcel Taeumel, Jens Lincke, Tobias Pape, and Robert Hirschfeld. Codetalk conversations about code. In *Creating Connecting and Collaborating through Computing (C5), 2010 Eighth International Conference on*, pages 11–18. IEEE, 2010

- Bastian Steinert, Michael Grunewald, Stefan Richter, Jens Lincke, and Robert Hirschfeld. Multi-user multi-account interaction in groupware supporting single-display collaboration. In *Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009. 5th International Conference on*, pages 1–9. IEEE, 2009

- Bastian Steinert, Michael Perscheid, Martin Beck, Jens Lincke, and Robert Hirschfeld. Debugging into examples: Leveraging tests for program comprehension. In *Testing of Software and Communication Systems*, pages 235–240. Springer, 2009

Workshop Publications

- Benjamin Hosain Wasty, Amir Semmo, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. ContextLua: Dynamic behavioral variations in computer games. In *Proceedings of the 2Nd International Workshop on Context-Oriented Programming*, COP '10, pages 5:1–5:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0531-0. doi: 10.1145/1930021.1930026. URL http://doi.acm.org/10.1145/1930021.1930026

Technical Reports

- Lenoid Berov, Johannes Henning, Toni Mattis, Patrick Rein, Robin Schreiberand Eric Seckler, Bastian Steinert, and Robert Hirschfeld. Vereinfachung der entwicklung von geschäftsanwendungen durch konsolidierung von programmierkonzepten und -technologien. Technical report, Hasso-Plattner-Institute, 2013

Book Chapters

- Bastian Steinert and Robert Hirschfeld. How to compare performance in program design activities: Towards an empirical

evaluation of coexist. In Larry Leifer, Hasso Plattner, and Christoph Meinel, editors, *Design Thinking Research: Building Innovation Eco-Systems*, Understanding Innovation, pages 219–238. Springer International Publishing, 2014. ISBN 978-3-319-01302-2. doi: 10.1007/978-3-319-01303-9_14. URL http://dx.doi.org/10.1007/978-3-319-01303-9_14

- Bastian Steinert, Marcel Taeumel, Damien Cassou, and Robert Hirschfeld. Adopting design practices for programming. In Hasso Plattner, Christoph Meinel, and Larry Leifer, editors, *Design Thinking Research: Measuring Performance in Context*, Understanding Innovation, pages 247–262. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31990-7. doi: 10.1007/978-3-642-31991-4_14. URL http://dx.doi.org/10.1007/978-3-642-31991-4_14

- Bastian Steinert and Robert Hirschfeld. Applying design knowledge to programming. In Hasso Plattner, Christoph Meinel, and Larry Leifer, editors, *Design Thinking Research: Studying Co-creation in Practice*, Understanding Innovation, pages 259–277. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-21642-8. doi: 10.1007/978-3-642-21643-5_15. URL http://dx.doi.org/10.1007/978-3-642-21643-5_15

- Robert Hirschfeld, Bastian Steinert, and Jens Lincke. Agile software development in virtual collaboration environments. In Christoph Meinel, Larry Leifer, and Hasso Plattner, editors, *Design Thinking*, Understanding Innovation, pages 197–218. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-13756-3. doi: 10.1007/978-3-642-13757-0_12. URL http://dx.doi.org/10.1007/978-3-642-13757-0_12

ACKNOWLEDGMENTS

Since I was not alone on my journey, I like to thank the people who accompanied me on my way.

But first, I would like to acknowledge and thank for the financial support of the Hasso Plattner Design Thinking Research Program. Participating in this program allowed me to get a deep undertanding of design and its relation to programming. During the bi-annual workshops I could absorb a great deal of knowledge about various fields related to design and met many fascinating people.

I thank Robert Hirschfeld for his trust in my abilities and giving me the opportunity to be his PhD student. I am thankful for his support, expertise, advice, and criticism concerning all respects we talked about. I am thankful for getting in touch with so many talented and interesting people through him.

I thank Ralf Lämmel for inviting me two times to his group where I always felt comfortable. I am thankful for his critical and detailed feedback on a previous version of this work and for the various conversations that were both exciting and thought-provoking.

I am thankful for every conversation I had with Richard P. Gabriel. In particular in early phases of my work, when I was still looking for the right words and had limited trust in the topic, it was helpful to talk to Richard who often seemed to resonate with my thoughts. I am also thankful for his kind of feedback that felt good but still made me think.

I am grateful to my colleagues for their feedback, support, having snowball fights, and the daily fun having weird conversations at lunch. Thank you Damien, Felix, Jens, Lauritz, Malte, Marcel, Matthias, Micha, Michael, and Tim. I am particularly thankful for every close collaboration on code and papers that involved challenging hard work as well as many ups and downs.

I also appreciate the close collaboration and exchange with the various students. I enjoyed my work during all seminar, master, and bachelor projects.

I thank my parents for their support and complying with my request to not ask any longer when I will be done.

I thank Cathrin and my closest friends Daniel, Felix, Jan, Markus, and Philipp for listening and reassuring words in bad times as well as taking part in my joy in good times.

I am thankful for my children Lena-Sophie and Felix. I enjoy seeing them learn and grow and I am looking forward to the next mountains we will climb together.

CONTENTS

1 INTRODUCTION 1
- 1.1 Designing Programs 1
- 1.2 Making Errors during Program Design 2
- 1.3 Problem Prevention and Its Limitations 3
- 1.4 Thesis Statement 3
- 1.5 Organization 4

2 BACKGROUND 7
- 2.1 Design and Programming 7
- 2.2 Program Design 8
- 2.3 The Need for Well-designed Programs 11
- 2.4 The Risks of Change 13
- 2.5 Best Practices to Prevent Tedious Recovery 16

3 MOTIVATION: THE TRADE-OFF BETWEEN COSTS AND SAFETY 19
- 3.1 Example Case: Unforeseen Recovery Needs 19
- 3.2 Trade-off between Costs and Safety 23
- 3.3 The Need for Built-in Recovery Support 28

4 COEXISTENCE OF PROGRAM VERSIONS 31
- 4.1 Squeak/Smalltalk 31
- 4.2 Concept Overview 34
- 4.3 Continuous Versioning 35
- 4.4 User Interface to Version History 38
- 4.5 Additional Environments 44
- 4.6 Continuous and Back-In-Time Analysis 48
- 4.7 Re-Assembling Changes 52

5 CONCEPT EVALUATION 57
- 5.1 How CoExist Helps in the Example Case 57
- 5.2 Informal User Studies 60
- 5.3 From Problem Prevention to Graceful Recovery 64

6 IMPLEMENTATION 67
- 6.1 Resolving Access to the Active Version 67
- 6.2 Preserving Meta-objects for All Versions 71
- 6.3 Late Class Binding 75
- 6.4 Limitations 78
- 6.5 Performance Evaluation 79

7 DISCUSSION 83
- 7.1 Why Program Design is Difficult 83
- 7.2 Benefits of a Reduced Need for Best Practices 85
- 7.3 Coding as a Means of Learning 88

8 LAB STUDY 93
- 8.1 Method 93

8.2 Results and Discussion 103
8.3 Study Design—Justification and Limitations 107
8.4 Threats to Validity 110
9 RELATED WORK 113
9.1 Versioning 113
9.2 Change Recording for Evolution Analysis 115
9.3 Juxtaposing Versions 116
9.4 Fine-grained Back-in-Time Impact Analysis 117
9.5 Re-Assembling Changes 117
10 CONCLUSION 119
10.1 Contributions 119
10.2 Summary 120
10.3 Future Work 122

BIBLIOGRAPHY 125

LIST OF FIGURES

Figure 1 Conceptual figures of CoExist featuring continuous versioning, running tests and recording the results in the background (left), side by side exploring and editing of multiple versions (left). Hovering shows which source code element has been changed; holding shift in addition shows the full difference to the previous version (right). 5

Figure 2 Three different conceptual models of programs. The left model is less structured and exhibits less order than the other two, thereby rendering development tasks more difficult. 10

Figure 3 Visualization Task Refactoring (from left to right and top to bottom) 20

Figure 4 Screenshot of a running Squeak system having opened various tools. From left to right and top to bottom: a class browser, a browser for senders of selected messages (selectors), a sound application, and a workspace. 32

Figure 5 A comparison of Squeak and Eclipse 32

Figure 6 Development tools working on meta-objects in Squeak. 33

Figure 7 The main concepts of CoExist as an extension to Squeak. 34

Figure 8 Versions representing changes to meta-objects. 36

Figure 9 (From top left to top right) A programmer modifies source code, which implicitly creates items in the version bar. The filled triangle marks the current position in the history - the version that is currently active. When a programmer goes back to a previous version (bottom left), and then continues working, the new changes will appear on a new branch that is implicitly created (bottom right). 39

Figure 10 Hovering shows which source code element has been changed (left). In addition, holding shift shows the total difference to the previous version (right). 39

Figure 11 The integration and interaction of source code browsers and version management tools. 40
Figure 12 Selecting source code elements in developments tools highlights related version items. 40
Figure 13 The version browser provides a tabular view on change history. Selecting a row shows the corresponding differences in the panes on the right. 41
Figure 14 Two conceptual tables showing lists of versions. The table on top presents two summary lines that, when expanded, change the representation of the table to the one on the bottom. In such tables, the numbers in the first column are the version identifiers. Following are the columns for the kind of change that triggered the creation of the version, the class and method that changed, and the time stamp of the change. 42
Figure 15 Composite Pattern for Groups of Versions 43
Figure 16 Working with an additional fully functional working place for a previous version, next to the tools for the current version. 45
Figure 17 Different environments of tools, each working on a different program version. 46
Figure 18 Continuous Analysis Concepts 49
Figure 19 Tests are run in the background for every version. The results are recorded and visualized for every version item. 50
Figure 20 Extracting meaningful increments by first studying the change history and then re-applying selected changes to another branch. 53
Figure 21 Top row: Selecting two changes and re-applying them to the currently active version (triangle). Bottom row: Withdrawing two changes, which results in re-applying subsequent changes to the change before the selection 54
Figure 22 Visualization Task with CoExist, Extracting a New Class 58
Figure 23 Visualization Task with CoExist, From Inheritance to Delegation 58

Figure 24 Visualization Task with CoExist, Recovery: Going Back to the Subclassing Solution 59
Figure 25 Main classes and methods of the CoExist implementation. 68
Figure 26 The same code snippet run in the context of two different versions evaluates to different results. 69
Figure 27 Graph of visual objects (morphs) shown in the conceptual screen above. 69
Figure 28 Accessing the currently active system dictionary by resolving the dynamic variable ActiveCodeBase. 70
Figure 29 Sharing of meta-objects between versions. 72
Figure 30 Copying meta-objects from the working copy to the newly created version 6. 73
Figure 31 LateClassBinding literals for the method `createPerson`. 75
Figure 32 LateClassBinding class in comparison to the Association class. 76
Figure 33 LateClassBinding introduces an additional value send after each push literal byte code. 77
Figure 34 LateClassBinding adds a push method literal and a value send byte code before each super send. 77
Figure 35 Invalid static reference after class modification. 78
Figure 36 Our experiment setup to compare performance in program design activities. 94
Figure 37 Screenshot of the LaserGame used for task 1. 95
Figure 38 Screenshot of the MarbleMania game used for task 2. 95
Figure 39 Size indicators for the games used in the study 96
Figure 40 The experimental procedure for both the control and the experimental group. 98
Figure 41 Excerpt of a spreadsheet with coded version data. 101
Figure 42 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. 102
Figure 43 Final scores for participants by task 104

Figure 44 A bar plot of the study results. Error bars represent the standard error of the mean. 105
Figure 45 An interaction plot of the study results. 106

ACRONYMS

VCS Version Control System

IDE Integrated Development Environment

VM Virtual Machine

JIT just-in-time compiler

LOC lines of code

1 INTRODUCTION

1.1 DESIGNING PROGRAMS

Programmers spend significant time on improving the design of their code base. Thereby, they consider qualities other than completeness and correctness. They are interested in the code's layout, its decomposition into modules, and the ways in which domain concepts are expressed [56, 19, 3]. Programmers strive for simplicity, elegance, and conceptual integrity in their constructions.

A suitable design helps programmers to understand, maintain, and further develop the program:

- Structure helps manage large amounts of concepts [56, 19],
- Modularization eases future modifications [56, 3], and
- Order and simplicity support thinking [2].

These aspects have no effect on the correctness of a program. But they render program design activities important. The achieved design outcome will influence how programmers perceive the program. Design activities thus affect later development steps and their success. Spending effort on improving program design pays off in future development activities.

Program design is particularly important in Agile development approaches such as Extreme Programming [8] or Scrum [65]. Agile teams can quickly deliver value due to short development cycles and incremental explorations. Programmers mainly rely on source code. It is the primary and often the only artifact produced and maintained during the development process. However, when code is the primary development artifact, it becomes imperative that it is easy to understand. Based on a high-quality code base throughout the entire project, development teams can respond quickly to new customer requests and changes.

1.2 MAKING ERRORS DURING PROGRAM DESIGN

While program design is important, it is also particularly difficult. One reason is that source code, which is the subject under design, is complex. A program typically comprises a huge number of source code entities such as methods and classes that are related to each other. Besides the huge number of entities, the relationships between these entities are sometimes unclear and ambiguous. Furthermore, entity names, which are a means of abstraction, are chosen by convention and personal preference. This gives room for misinterpretation and errors. Besides the complexity of source code, program design is also difficult because it is ill-structured. There is no clear problem statement nor is there a clear goal to be achieved. Also, there is typically no objective measure for good or bad design that helps to decide whether one solution is preferable over another. Instead, program design relies on subjective value judgment. The ill-structuredness of design and the complexity of the domain account for the difficulty of program design tasks.

Because designing programs is difficult, it can easily lead to undesired development situations. When programmers improve their programs' design, for example, by removing indirections that seem unnecessary, the following things can happen:

- A promising idea turns out inappropriate. The programmer now wants to continue exploring a previous idea, but has already modified many other parts of the source code.

- The made changes affect the program in undesired ways. But the programmer has difficulties to find the changes that cause the undesired behavior.

- The source code turns out to be more complex than the programmer expected. It is now unclear how the code was working before.

- The made changes cover several independent concerns. Thus, they should now be shared in distinct increments (commits). But this requires careful revisiting and re-assembling.

In situations like that, programmers have to spend effort to get back to a desirable development state. They have to manually withdraw some of the recent changes, recover knowledge from previous development states, or detect and fix a recently introduced bug. Such recovery activities are often tedious and can easily be frustrating.

1.3 PROBLEM PREVENTION AND ITS LIMITATIONS

To avoid tedious recovery efforts, literature recommends a structured and disciplined approach to programming, which consists in the application of various best practices [8, 29, 28]. Best practices include the regular use of testing and versioning tools [69] such as Git [86, 14] or Subversion [28], but also performing small steps and working only on one thing at a time. Regular testing helps discover bugs early. It thus reduces fault localization costs. Regular committing helps going back to a previous state. And, working on one thing at a time eases to commit independent increments. The application of such practices is a form of problem prevention. Best practices are precautionary activities that programmers can conduct to avoid undesired situations in the future.

However, programmers can experience sudden needs of unexpected recovery even though they have been applying best practices as recommended. They might run tests regularly, but still discover some bugs only late in the process, which then requires tedious effort to localize the faults and fix them. Similarly, programmers might only work on one thing at a time and regularly commit small increments, but still experience the need for manually withdrawing recent changes. They might realize only after they have finished a task that their changes made should better be shared in two separate increments, which then requires additional effort to extract meaningful changes from the difference between the current version and the last commit. Also, while a thought might first seem small and only minimally significant, seeing it realized in source code might stimulate new and better ideas, whose realization first require withdrawing already made changes.

1.4 THESIS STATEMENT

This dissertation argues that, first, the application of best practices to prevent recovery has limitations. While it is often easy to tell how tedious recovery work could have been avoided in hindsight, programmers can hardly predict recovery needs, which makes prevention difficult. It will be shown how the application of best practices is time-consuming and requires trade-offs to be practical.

Addressing this problem, this dissertation further argues that, second, recovery can be made fast and easy to accomplish by providing dedicated tool support. It will be shown how built-in recovery sup-

port can preserve immediate access to static and dynamic information of intermediate development states.

This work will show how such tool support has been implemented as an extension to the Squeak/Smalltalk programming environment. Moreover, this dissertation will explain how the level of available recovery support can positively affect cognitive processes during programming work. The results of a controlled experiment suggest that additional recovery support improves programming performance in explorative tasks.

In sum, the main argument of this dissertation is as follows:

THESIS STATEMENT Programmers benefit from dedicated recovery tool support that automatically keeps recovery tasks fast and easy to accomplish by preserving immediate access to static and dynamic information of intermediate development states.

1.5 ORGANIZATION

Chapter 2 provides background information. It introduces the concept of design and relates it to programming. It then describes program design and explains how it is important in software development.

Chapter 3 motivates the additional tool support proposed in this work. A case example illustrates how programmers can suddenly experience the need for recovery effort even though they have followed best practices. After a discussion of this case example, this chapter describes the limitations of the approach to avoid unexpected recovery.

Chapter 4 introduces CoExist, a set of tools dedicated to support various recovery needs [77]. CoExist closes a gap between undo/redo and Version Control Systems (VCSs) [69]. While undo/redo is scoped to individual documents, version control requires explicit and manual control. Fig. 1 illustrates some of the main user interfaces concepts of CoExist.

Chapter 5 shows how CoExist would have helped the programmer in the reported case example. In addition to that, three informal user studies are presented, which accompanied the development of CoExist and helped better understand various usability aspects. It is

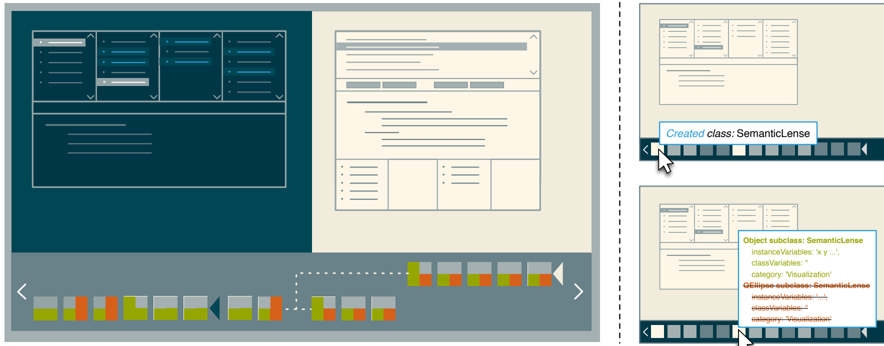


Figure 1: Conceptual figures of CoExist featuring continuous versioning, running tests and recording the results in the background (left), side by side exploring and editing of multiple versions (left). Hovering shows which source code element has been changed; holding shift in addition shows the full difference to the previous version (right).

then explained how recovery support such as provided by CoExist reduces the need for problem prevention.

Chapter 6 demonstrates feasibility of the proposed concepts. CoExist has been implemented in Squeak/Smalltalk [38]. The characteristics of this IDE inspired an implementation that manages versions of meta-objects instead of source code files, which in turn encouraged the development of tools to compare versions of both development and run-time artifacts. After presenting selected aspects of the implementation, its performance is evaluated.

Chapter 7 discusses various implications of the proposed approach. Making recovery easy and fast such as provided by CoExist is generally beneficial: it helps in case of unexpected recovery needs. But it is particularly beneficial for development tasks that are prone to misinterpretation, false assumptions, and other errors. In addition, dedicated recovery support also brings cognitive benefits: it supports understanding, inference, and problem solving during program design tasks.

Chapter 8 describes the setup and the results of a controlled lab study, which has been conducted to examine whether recovery support such as CoExist affects programmers' performance. After describing the study design including material and tasks, the results of the statistical analysis are presented and discussed. The chapter ends with sections on the study's limitation and a justification of the study design.

Chapter 9 presents related work. The presentation is divided among the main features of CoExist, which are: versioning, software

evolution analysis, juxtaposing versions, back-in-time impact analysis, and re-assembling changes.

Chapter 10 summarizes the results of this dissertation and outlines directions for future work.

2 BACKGROUND

This work assumes that programmers care about the quality of their code base and regularly conduct program design activities. Therefore, the first three sections of this chapter are concerned with program design: what it is and what it is good for. Afterwards, it is explained how changing source code can suddenly and unexpectedly lead to tedious recovery work. This is followed by a description of best practices, which are recommended to prevent such tedious recovery work.

2.1 DESIGN AND PROGRAMMING

In [68], Herbert Simon argues that *design* is central to various disciplines (professions):

> Engineers are not the only professional designers. Everyone designs who devise courses of action aimed at changing existing situations into preferred ones. The intellectual activity that produces material artifacts is no different fundamentally from the one that prescribes remedies for a sick patient or the one that devises a new sales plan for a company or a social welfare policy for a state. Design, so constructed, is the core of all professional training; it is the principal mark that distinguishes the professions from the sciences. Schools of engineering, as well as schools of architecture, business, education, law, and medicine, are all centrally concerned with the process of design. [68, p. 130]

According to this description, design crosses the boundaries of individual disciplines. Practitioners of various disciplines perform activities that involve design. The description on Wikipedia supports this generic character of design:

> Design as a noun informally refers to a plan or convention for the construction of an object or a system [...] while "to design" (verb) refers to making this plan. [...] However,

> one can also design by directly constructing an object (as in pottery, engineering, [...], graphic design) [83]

Giving these definitions, it is reasonable to consider the construction of programs as design. While researchers have not yet agreed on a definition of design, the following attempt allows for highlighting the connection to programming. Programming-related concepts are inserted in square brackets and rendered in italics.

> (noun) a specification of an object [*the program*], manifested by an agent [*the developer*], intended to accomplish goals, in a particular environment, using a set of primitive components [*programming language, libraries, ...*], satisfying a set of requirements, subject to constraints [*readability, performance, ...*]; (verb, transitive) to create a design, in an environment (where the designer operates) [59]

In line with the definitions, programming arguably involves design and programmers conduct intellectual activities that are similar to the activities of designers in other fields.

However, in the context of programming, it is meaningful to distinguish between two kinds of design according to whom the design is for. On one hand, programmers design a software for end-users, which involves the overall experience and interaction processes. On the other hand, programmers also design for other programmers including themselves in the future. They continuously improve the code base in preparation for current and future coding activities.

This dissertation focuses on design activities of the second kind, which will be referred to as *program design* in the following.

- **User-experience design** targets end-users and involves the graphical layout and processes that guide users;

- **Program design** targets programmers and involves the consideration of names for program constructs, indentation and code layout, abstractions, separation into modules;

2.2 PROGRAM DESIGN

Developers continuously redesign their programs. A canonical reference about best practices in programming starts by listing some crucial questions programmers face everyday [6]:

- How do you choose names for objects, variables, and methods?
- How do you break up the logic into methods?
- How do you communicate most clearly through your code?

Such questions are important for current and future programming tasks. Continuously considering these and other aspects of program design eases readability and maintainability of the source code. For example, while the following two snippets of Smalltalk source code have identical functionality, the latter version is easier to comprehend and work with.

```
s := " ... "
r := OrderedCollection new.
1 to: s size
do: [:i |
e := s at: i.
e < 5 ifTrue:
[r add: e]]
```

```
givenNumbers := " ... "
result := OrderedCollection new.
1 to: givenNumbers size do: [:i | | eachNumber |
           eachNumber := givenNumbers at: i.
           eachNumber < max
                ifTrue: [result add: eachNumber]]
```

The second version is easier to understand for two reasons. First, names can increase the readability of a program if they clearly express the role of the identified constructs. Second, indentation of lines better conveys the structure of the source code.

Another aspect of program design is the invention of abstractions, which allows for expressing common needs more precisely. The following source code abstracts over iterating elements and filtering according to a predicate.

```
Collection>>select: predicateBlock
    newCollection := OrderedCollection new.
    self do: [:each |
        (predicateBlock value: each)
            ifTrue: [newCollection add: each]].
    ↑ newCollection
```

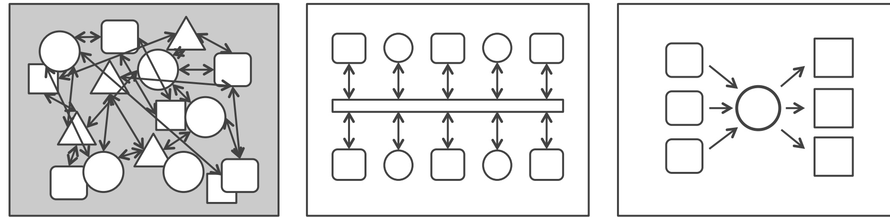


Figure 2: Three different conceptual models of programs. The left model is less structured and exhibits less order than the other two, thereby rendering development tasks more difficult.

While the above definition of the *select:* method is harder to read than the above code iterating over numbers, it allows clients to express their goals in much less code, as the code below shows:

```
givenNumbers select: [:each | each < 5]
```

The new abstraction increases expressiveness of the previous code snippet, makes it easier to read and understand, given that the reader is familiar with the new abstraction. In this example, there is hardly a right or wrong solution. The different solutions in the design space can rather 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 and is more concise.

Program design also refers to the intentional effort of structuring an application's code base. Figure 2 shows different conceptual models of a program, which represent the decomposition of a program and how the parts interact with each other. In structuring the code base, programmers strive for simplicity and consistency, which is useful for the following reasons:

- **Structure helps manage immense amounts of information**, which can easily add up to the size of thousands of books or even an entire library [39],
- **Modularization eases change**, which can reduce the number of elements to be understood and modified in subsequent development steps,
- **Order and simplicity support thinking:** Simplicity seems like a natural desire—it enables one to see clearly. The arrangement of elements partially determines our perception and thoughts. Conceptual models like those presented in Figure 2 implicitly

act as frames that we use to understand the problem and solution spaces. Moreover, simplicity and order make us arguably feel better, which in turn positively affects creative thinking [2].

These aspects do not affect the correctness of a program, but they render program design activities important; their outcomes determine how the program is perceived, analyzed, and processed.

2.3 THE NEED FOR WELL-DESIGNED PROGRAMS

Programmers care about design aspects because they know it might affect the development process. Appropriate source code design helps in two ways: it eases the construction of new building blocks and it also supports maintenance and evolution of existing building blocks.

2.3.1 *Constructing new building blocks*

Programs are typically split up into building blocks such as methods and classes. These building blocks can be used to implement other building blocks, and so on. The decomposition of programs helps to manage complexity [12]. It aids being specific only about details relevant and inherent to a particular concept and to leave out the details of others. A proper decomposition eases the creation and comprehension of the individual parts, and thus supports working on large complex systems [56].

While programmers' current way of thinking about a feature determines how they will express it in source code, the set of building blocks available and previously implemented determines how they think about the problem. These available building blocks define a frame of reasoning about the problem domain. Programmers naturally try to express upcoming features in the terms–the buildings blocks–they have previously defined.

What has previously been implemented also determines, to a certain degree, the amount of programming effort needed to implement current features of interests. For example, the availability of a construct such as the `select:` method determines the amount of code to be written for tasks such as filtering certain elements from a collection. Consequently, the result of current programming efforts can

make future tasks easy and simple to accomplish, requiring only little code.

2.3.2 *Evolving existing building blocks*

The design of programs affects future implementation tasks, because programmers will change the code that is written today in the future. There is hardly a piece of code that, written once, has never been changed afterwards. Programmers will have to work on source code written by others and on source code written earlier by themselves. In both cases, they must 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. And to gain an understanding can be facilitated or impeded by the program's design.

Changing previously written source code is necessary for two reasons. First, the understanding of the problem can co-evolve with the understanding of the solution [20]. When programmers have an initial understanding of the problem, they will come up with ideas and implement them. But the first implementation can reveal limitations, which shows the insufficiency of the initial understanding. So, programmers have to iterate over their program understanding and their solution to eventually complete the task.

A second reason for changing code is that programmers approach problems incrementally. As the task to be fulfilled is typically large and complex, programmer have no choice but to start by focusing on one part of the problem [19] and write source code for this part. However, the parts chosen consecutively will likely overlap in different respects. Consequently, the code needed to cover the functionality of the various parts is likely to overlap as well. But source code that has been for only part of the problem typically lacks some aspects or features that are required for other parts. So, source code needs to be adapted to the additional requirement it should meet.

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, where programmers are mainly concerned with how to build the desired 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 potentially requires changing the code written for previous ones.

2.4 THE RISKS OF CHANGE

While programmers regularly make changes to their programs, changing programs always involves the risk of making *errors*. In this context, the term *error* is used with a meaning different from the term *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...." [51]

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 tedious work to recover from it. The following situations can arise.

2.4.1 *Changes suddenly turn out inappropriate*

In the course of working on a programming task, one particular implementation idea or a part of an idea might suddenly turn out inappropriate. Seeing the idea implemented can reveal unforeseen implications that are undesired. For example, trying to improve several methods by removing an indirection that first seemed unnecessary suddenly reveals the actual purpose of this indirection. In such a situation, programmers want to have the respective changes made undone.

However, there is a high probability that an appropriate commit is not available, which means reverting the code base to the last commit will not only withdraw undesired changes but also changes that should be kept. In this case, programmers have to withdraw the previously made changes by using the undo feature of the text editor. More specifically, it requires manually applying the undo command an unknown and different number of times for each changed file.

2.4.2 *Program knowledge turns out insufficient*

While being in the middle of a large change, programmers might become aware that their understanding of the system, as it was, has been insufficient. For example, they might not understand why the new code does not work as expected, although it seems like a clean refactoring.

To close the knowledge gap, programmers can employ a VCS. Such systems make it typically easy to glance at a previous version of a file. Nevertheless, this approach is impractical if the knowledge is spread across many files or if the programmer needs to execute or debug a previous version.

A workaround is to ask the VCS to replicate all previous files alongside the current working copy. And if programmers require execution or debugging, they will also have to set up a second instance of their environment. Some VCSs such as Git also enable programmers to stash current work (i.e., to save current changes and check out the previous snapshot), understand the previous code, and then bring back the current version to apply the new understanding. However, this approach has the disadvantage of losing a view of the previous code while working on the current one which requires programmers to rely on their memory. Moreover, the concept of staging adds significant complexity to the VCS, which might be more relevant to the user than its potential benefits [**?**].

The above discussion assumes that an appropriate snapshot, with the knowledge of interest, is available. Nevertheless, if no appropriate snapshot is available, programmers have to undo their changes first to manually get back to a previous development state.

2.4.3 *New bugs occur*

Changing source code involves the risk of introducing bugs. Because software systems are huge and complex in relation to what a human mind can keep and process in mind, it is hard to acquire and maintain in-depth knowledge of every single expression in the code base.

When programmers ignore testing for a while, it can require significant effort and time to find, understand, and remedy faulty behavior that they accidentally introduced. Programmers might ignore testing when they follow a particularly interesting idea. Being caught up in

creativity, they simply forgot to run tests. Alternatively, they might ignore testing intentionally, because they first want to prototype an idea to study its feasibility and value.

However, the more changes are made between test runs, the harder its gets to understand and fix newly discovered bugs. It gets harder to remember all the changes that have been made, which are possible causes for the new bugs. In addition, it gets harder to understand the effects of each change in detail.

2.4.4 *Changes cover multiple independent concerns*

Programmers want to commit their improvements to share them with others. However, they might have difficulties separating the made changes into meaningful increments. One commit should contain changes that belong to one improvement [28]. When the implementation of a new feature required a particular refactoring, it is recommended separating the refactoring from the feature implementation and sharing the different sets of changes as independent increments. When authors group their changes into small meaningful chunks, consumers (co-developers) of these commits need to spend less time to understand and evaluate them, because the overall information is already partitioned so that it is easy to process. More specifically, this practice helps co-workers to read the commit log, which is useful to keep informed about the project, but it also eases various integration tasks, when, for example, only a few changes committed to one feature branch should be applied to the master branch.

However, current tool support makes it hard to create small meaningful commits out of a large set of various changes. Versioning tools are unable to show all the single changes made at a semantic level, they can only show the difference between the current development state and previous commits. For example, even though programmers might have modified a certain code element multiple times and for different purposes, tools will only show a single textual difference for this element. Programmers have to remember that the element has been changed multiple times for multiple purposes. In addition, they have to manually reconstruct which changes have been made for which purpose.

Furthermore, when programmers pull apart the difference to the previous commit into multiple new commits, they can hardly check whether the set of grouped changes is complete and works as desired. Only after committing their changes, programmers can checkout the

corresponding revision to see if it represents a correct, complete, and running version, which is meaningful for others. If not, programmers have to manually create new commits to eventually achieve a satisfactory revision history.

2.5 BEST PRACTICES TO PREVENT TEDIOUS RECOVERY

The issues listed above are 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 include, for example:

- Work on one thing at a time, which might be a new feature, a refactoring task, or a bug fix. Avoid mixing the implementation of a new feature with refactoring and bug fixing with refactoring; in general, avoid implementing multiple features at the same time. This avoids losing track and simplifies sharing your improvements with others.

- Small Steps: Make only small changes that you can easily control. Make sure you understand the items of work. Consider breaking down items into manageable parts or consciously deciding for a phase of experimenting, which should be preceded by saving the current development state, for example, when using Git, by committing or stashing recent changes.

- Write tests and run them regularly. To find out if recent changes introduced faulty behavior, programmers can validate the applications behavior. To avoid the laborious effort of manually checking for the desired behavior, it is recommended to write and maintain a suite of automated tests [8, 7]. Such a test suite should ideally check for the correct behavior of most if not all aspects of the application, so that the passing of all tests is a sufficiently good indicator that the application works as desired. If this is the case, the failure of one or more tests reveals that recent changes introduced faulty behavior. Running automated tests helps detect newly introduced bugs early in the process. The set of changes that can be the cause of a new bug remains small, so that the actual failure cause can always be located relatively fast.

- Employ a distributed VCS such as Git or Mercurial and commit meaningful increments. Make regular and frequent use of it by committing small increments. This allows for going back to a stable state more easily.

The general pattern of these practices is that programmers should anticipate that they will make errors and should thus perform prophylactic activities continuously and regularly in order to keep the cost for recovery scenarios low.

SUMMARY

Design is central to various professions including programming. The intellectual activities during program design tasks are similar to those during design tasks in other fields. Program design has the goal to support current and future programming tasks. It is concerned with names, abstractions, or the decomposition into modules, among other. Appropriate abstractions and a proper decomposition helps to manage the complexity of the domain. Currently available abstractions and modules also determine the vocabulary and the concepts for reasoning about the domain and affect the effort for implementing subsequent features. In consequence, a program's design can ease or impede program comprehension and maintenance tasks.

However, changing source code comes along with the possibility of making errors. Changes can turn out inappropriate, program understanding can turn out insufficient, or bugs can creep into the code without notice. Such errors typically requires tedious effort to recover from them. To prevent tedious recovery, literature recommends the application of various best practices: programmers should only work on one thing at a time, make only small steps that they can control, and run tests regularly and frequently. In general, it is recommended to anticipate the possibility of errors and to conduct corresponding precautionary activities.

3 MOTIVATION: THE TRADE-OFF BETWEEN COSTS AND SAFETY

The chapter starts with an example case that illustrates how programmers can still have the need for tedious recovery, even though they have followed the recommended guidelines. After a discussion of this example case, this chapter explains the limitations of the recommended best practices to prevent problems and finally derives the need for built-in recovery support.

3.1 EXAMPLE CASE: UNFORESEEN RECOVERY NEEDS

Despite these practices and recommendations that should help avoid problems, programmers face undesired situations from time to time, as for instance, the following example case illustrates. While intuition might suggest that programmers are the ones to blame, because they obviously ignored recommendations, the discussion will show that it is not obvious that the programmer made an error and intentionally acted against the rules, rendering this case report an example that our intuitive judgment is likely to be biased.

3.1.1 *Experience Report*

Background: At the time of experiencing the situation, the student was 23 years old and in the second semester of a masters curriculum in computer science. (He already held a bachelors degree in computer science). Compared his fellow students, he has been a very experienced programmer due to participation in various open source projects in his spare time, internships during semester breaks, and regular part-time jobs as programmer. He has been a proponent of Agile principles and practices in general and testing and versioning in particular. He has liked Git and has already had 2 years of experience using Git.

The student had been working on a visualization task using the Qt framework. At some point, he recognized that he has been added

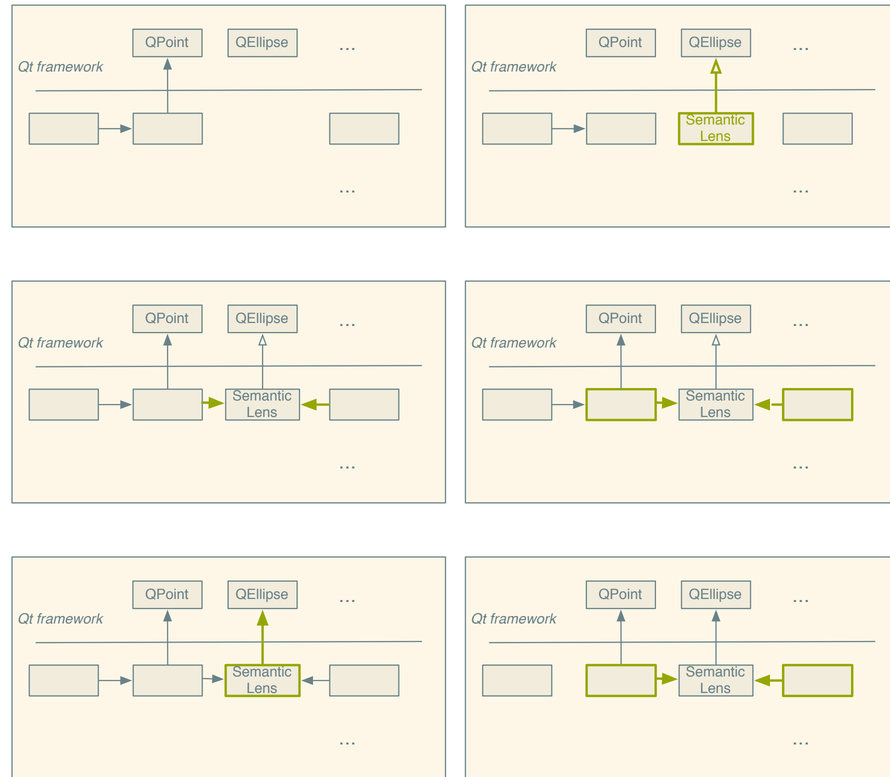


Figure 3: Visualization Task Refactoring (from left to right and top to bottom)

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. Figure 3 illustrates the refactoring.

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 got 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, which 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 SemanticLens class (bottom row of Figure 3, and added a field and accessor methods to maintain a reference to a QEllipse object and also added initialization code. He changed the methods in SemanticLens 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 belief was wrong. He could now see that subclassing is preferable to delegation in this situation because hav-

ing access to the methods of the superclass is actually useful in his program. As a consequence of this insight, the student then faced the laborious task of manually withdrawing all the changes previously made to replace subclassing by delegation. He had to identify the relevant artifacts (files), and for each file, he had 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, manually withdrawing also took a few minutes. The required recovery work would have been even more tedious, if the student had made further changes before recognizing the error.

3.1.2 *Should The Programmer Have Done Things Differently?*

Programmers who read or listen to a story like the one above will likely come up with ideas how the student should have acted to avoid the need for recovery work. But as one can find arguments how the student should have done better, one can also find counter arguments why the advised alternative would not have been a meaningful option. The following two arguments and corresponding counter arguments illustrate opposite points of view.

3.1.2.1 *Pros and Cons of Making Checkpoints*

A REVIEWER: The student could have made a local commit before starting to replace the subclassing mechanism with the delegation mechanism. The commit would have served as checkpoint and there would have been an easy way to recover from the undesired situation.

THE STUDENT: At this time in point the code was in an intermediate state and the task was not yet completed. So, I would have committed a development state that is of no use to other developers. And, using this commit only locally and avoiding sharing it with others would have required additional work to clean up the revision history.

3.1.2.2 *Pros and Cons of Deferring Subtasks*

A REVIEWER: The cause of the problem was mixing two different tasks: implementing a feature and refactoring the code. The student

could have finished implementing the functionality first before changing the code. Finishing the implementation would have helped to understand the code and the options for improvement more easily, so that the student would have seen the benefits of subclassing over delegation upfront.

THE STUDENT: In the moment of seeing and working with the code, changing the recently created class to employ the delegation pattern did not appear as a different task. The idea appeared more like a minor issue, similar to changing a recently entered phrase when writing a text passage. These are small issues that people often fix in the moment of noticing them. If I had stayed with the subclass solution, I would have written more code that I would have needed to change when moving toward the delegation solution. This additional overhead gives reason to make the changes immediately.

3.1.3 *The Role of Experience*

Another interpretation is that sudden recovery needs are caused by a lack of experience. If the student faces a similar situation again, he will probably behave differently. It is likely that he will either stay with the subclassing solution as the alternative was inappropriate the last time, or he will be more careful before making changes to avoid making the same "mistake" again.

However, even though experience can reduce the amount of unexpected recovery efforts, it hardly improves the ability to predict and prevent previously unknown problems. For example, programmers who have considerable experience in developing web shops using a particular web framework will rarely experience situations of unexpected recovery. Over time, such programmers have learned how to accomplish the required tasks. They have increased their repertoire of proven building blocks, which makes them efficient. Relying on proven building blocks implies that the space for design options has been reduced, which then gives little little room for making errors. However, the impact of experience is limited. When tasks or tools change significantly, or when basic assumptions get challenged, the design space widens and the amount of uncertainty increases. The chance for unexpected discoveries increases, and so does the chance for unexpected recovery needs. While previously established and proven building blocks will still be of use, their applicability and usability will be limited.

3.1.4 *Hindsight Bias*

While a story of unexpected recovery needs can easily suggest inadequate behavior, this might not necessarily be the case, as the above juxtaposition of arguments illustrates.

Hindsight bias refers to the tendency to interpret past events differently than they actually were due to the knowledge of the outcome [26, 45]. An often used example is that many people believe they *knew* that a crisis will come, although it was unpredictable for experts in the field. Even though randomness played a significant role, people tend to see cause-effect relationships among a chain of events described in a story. Applied to the experience report of the student, it is tempting to see the absence of certain actions as the cause of the need for recovery, although such an outcome might have been considered for from reasonable when being in the situation.

In sum, best practices help to avoid problems and ignoring them will likely lead to problems. However, employing best practices might not be sufficient to avoid sudden needs of recovery. The subsequent section will discuss the limitation of the preventing unexpected recovery needs.

3.2 TRADE-OFF BETWEEN COSTS AND SAFETY

The discussion of the example case in the previous section demonstrates that the application of best practices involves trade-offs. On the one hand, performing activities such as committing or running tests help reduce the risk of unexpected recovery needs, but, on the other hand, such practices also consume time and do not always appear appropriate.

While the example case focuses on versioning, trade-offs are inherent to the approach of avoiding undesired recovery work by relying on best practices. The following scenarios are further examples how programmers can require tedious recovery work although they regularly employ best practices:

- Programmers might run tests regularly, but still discover some bugs only late in the process, which then requires tedious effort to localize and fix. Bugs can remain undiscovered when programmers run only a subset of tests regularly and run the entire suite of all tests only sporadically. For example, program-

mers might have run the tests for the current unit of interest, but the changes might have had effects on other units.

- Programmers might work on one thing at a time and regularly commit small increments, but still experience the need for manually withdrawing recent changes. Programmers might realize only after they have finished a tasks that their changes should better be shared in two separate increments, which then requires additional effort to extract meaningful changes from the difference between the current version and the last commit.

To prevent tedious recovery efforts more effectively, programmers could employ best practices more rigorously. However, it requires an impractical amount of development time to employ best practices with an intensity that minimizes the risk of unexpected recovery efforts. The following sections discuss the trade-offs for the various problem dimensions.

3.2.1 *Trade-off for Version control*

To always have a proper snapshot to go back to in all circumstances, programmers would have to make a commit after every small change. Frequent commits would be required, because every promising idea can suddenly turn out inappropriate or previously withdrawn solutions can suddenly become interesting again. Consequently, every development state can be of relevance in future development situations, including those intermediate development states before completing the next meaningful increments. However, making commits of all (or most) intermediate development states would require a disproportionate amount of development time, for two reasons: *First*, every commit requires a commit message that later aids finding the relevant version of interest rapidly. But writing such commit messages is a time-consuming activity, which requires analyzing the made changes, summarizing them, and describing their intent. *Second*, the main branch of a project's commit history should include commits that represent meaningful increments. So, frequently making commits ignores this aspects of meaningful increments, and would thus require cleaning up the commit history later on. This involves extracting all relevant changes, sorting irrelevant changes out, and assembling meaningful commits that are clean and function properly. Such clean up activities, which are not well supported by tools, can easily be time-consuming and tedious.

Because of the effort required for each additional commit, programmers have to compromise on the density of their safety-net of commits. They will typically make commits after task completion or after having achieved a meaningful increment. And only in exceptional circumstances, if risks suggests doing so, programmers will make a commit in between only for the purpose of preserving fast access to a particular development state. Consequently: every now and then, there will be a lack of an appropriate commit to go back to or to recovery information from.

3.2.2 *Trade-off for Planning and Structuring*

As the proverb "foresight is better than hindsight" suggests, thinking about the upcoming programming steps before actually making changes to the code can help to avoid unexpected recovery work. For example, when programmers consider removing the usage of the observer pattern [**?**] because the indirections appear unnecessary, they could thoroughly think about the necessary changes and their impact before making those changes to the code base. This might help discover problems and limitations upfront, which in turn avoids tedious recovery work later. *However*, ignoring a single aspect of a problem can render a promising idea inappropriate. Programmers thus would basically need to know all required changes and their impact in order to ensure the absence of sudden needs for recovery. In the example, to exclude all possibility of wrongly assessing the dispensability of the observer pattern, programmers would first need to find out in detail how all involved objects collaborate with each other, which messages are exchanged, and what the corresponding effects are. Programmers would also need to think through how the source code will look without having these indirections in order to be sure, in advance, whether the code would truly be easier to understand. To rule out the possibility for errors and the need for recovery, programmers would have to solve the programming task mentally upfront, before making any changes.

But because solving a programming task mentally first can easily become tedious and time-consuming, programmers will often think only partially about the upcoming work. For example, they might try to pinpoint those aspects of the task that appear particularly risky and reflect only about them. However, programmers can easily fail to consider crucial aspects of the problem.

3.2.3 *Trade-off for Testing*

To ensure low cost for fault localization and repair, programmers need to test the entire application after every small change. They need to do so because changes can have implications that are generally hard to predict. Changes might, for example, unexpectedly affect system parts beyond the borders of the current module of interest. To discover such unforeseen side-effects, programmers need to run the entire test suite, given that the test suite covers 100 % of the functionality. However, running tests can easily consume a considerable amount of development time, from several seconds to minutes to hours, depending on the software system and the test suite. This overhead makes it impractical to run the entire test suite after every small change.

The costs for testing typically lead to trade-offs. Programmers run those tests continuously that directly belong to the unit of interest and run the entire test suite only after having achieved a major increment. Also, programmers run tests not after every small change, like after every statement they changed, but rather run them after having made a few small changes. More generally, the selection of tests as well as how often they are run depends on a personal subjective assessment of risks and needs.

3.2.4 *Meta Trade-off: The Possibility to Forget, and the Costs to Avoid It*

To ensure the appropriate application of best practices, programmers need to constantly reflect on their current and future activities. Constant reflection is required because the application of best practices is a secondary task that programmers have to perform in addition to their primary task, the actual coding work. Such a secondary task is easy to forget, in particular because there are no reminders or triggers when, for example, it is meaningful to consider the possibility of making a commit, or when to run tests, or when to adapt the subset of tests to be run. It is thus easy to ignore such considerations. But to avoid forgetting and thus to avoid ignoring an important situation, programmers have to continuously think about secondary tasks. However, continuously thinking about committing, testing, and restructuring upcoming work easily gets tedious, impedes focusing on the primary task, and makes it hard to get work done.

Because of the costs that constant reflection on programming activities implies, programmers will typically compromise on the consis-

tency and rigour in the application of best practices. When they focus on their ideas and the corresponding changes, they will easily ignore the need for reflection. They will typically only consider committing or running tests, when the task of current interest is completed or when an event interrupts the flow.

3.2.5 *Trade-off: Return on Investment*

Not only that precautionary actions require the investment of valuable development time, but the return on this investment is uncertain. Programmers might run tests often and regularly, but might rarely detect bugs during these test runs, because they rarely introduce any. Similarly, programmers might make commits frequently and carefully write meaningful commit messages to ensure easy and fast recovery, but might rarely have the need to go back to a previous development state, because their ideas turn out useful and sufficient most of the times. Programmers might spend more effort on prevention than they would require to recover from unexpected situations. They might not experience recovery needs at all. More generally, the investment of development time in problem prevention might not pay off, which is in conflict with the need to expend energy on problem prevention.

3.2.6 *Implications*

The trade-off between costs and safety implies that the prevention of recovery problems either requires a disproportionate amount of development time or is error-prone. It is otherwise error-prone because employing best practices selectively can hardly avoid unexpected recovery efforts. To reduce the impact of luck, programmers can assess the risks of the current situation. They can reason whether it is worth running and waiting for the entire test suite or cleaning up and making a commit before moving on. But it is hard to estimate the overall effects of modifications in general and to estimate whether the boundaries of modules are crossed in particular. It is also hard to predict how an idea will turn out. Realizing an idea often exposes more implications and constraints than expected in the beginning. More generally, the assessment of risks is typically based on heuristics rather than on facts, which makes it error-prone. Programmers can thus rarely ensure the early detection of bugs by running only a small selection of tests. Similarly, they can rarely ensure a proper commit to

go back to by only making commits when a task is finished or before approaching a task that appears particularly risky.

3.3 THE NEED FOR BUILT-IN RECOVERY SUPPORT

The trade-offs mentioned above can be resolved by providing tool support that makes recovery tasks easy and fast to accomplish. More specifically, Integrated Development Environment (IDE)s should provide dedicated support for the following recovery scenarios:

WITHDRAWING CHANGES IDEs should support programmers in withdrawing recent changes and starting over from a previous state. IDEs should provide a safe environment where developers can try out ideas without fear of loosing the current stable state of development.

RECOVERING KNOWLEDGE IDEs should support programmers in recovering knowledge from a previous version. This would avoid the need for a precise understanding of every detail before making any changes.

CORRECTING RECENT MISTAKES IDEs should support programmers in finding the change that caused a test to fail. This would allow to focus on the task at hand and to assess test quality only when it is convenient.

RE-ASSEMBLING CHANGES IDEs should support programmers in re-assembling their recent changes into incremental improvements. This would allow programmers to defer the consideration of code sharing and to focus at the task at hand.

SUMMARY

Following best practices can fail to avoid unexpected recovery needs. An example case shows how a programmer, after implementing an idea, suddenly discovered that the changes made did not improve the situation. Despite the fact that programmers can make errors, employing best practices is both time-consuming and exhaustive. For example, running the entire test suite after every small change or thinking through the effect and value of every change clearly renders programming efforts inefficient. Because of that, trade-offs are required, which, however, imply the risk for unexpected problems.

These trade-offs can be avoided by providing tool support that makes the various recovery needs fast and easy to accomplish.

4 COEXISTENCE OF PROGRAM VERSIONS

This chapter introduces CoExist, an approach and a set of development tools that offer *built-in recovery support*. CoExist supports programmers by automatically keeping recovery costs low and is meant as a mechanism that complements undo/redo and VCSs such as Git [86, 14].

After first giving an overview of CoExist's feature set in the first section, the subsequent section explains how the tools would have helped the recovery scenario illustrated previously (Section 3.1). The sections after that describe the various aspects of CoExist's support in detail.

4.1 SQUEAK/SMALLTALK

The Squeak/Smalltalk system [38], Squeak for short, is an open-source implementation of Smalltalk [32, 31]. Squeak provides an environment to open and interact with games, applications, or other tools. The user interface to Squeak, which is based on Morphic, provides directness and liveness [52]. On top of that, Squeak supports the multi-window paradigm. Windows are the typical container of development tools. Windows can overlap and be moved around. Figure 4 shows a screenshot of a Squeak system that has opened a wave editor application and a few development tools.

A LIVE PROGRAMMING SYSTEM The development tools (shown at the top of Figure 4) enable programmers to browse and modify source code. They can work on their applications, but can also browse and change the source code of the Squeak system itself. The source code of the entire system is an inherent part of the same. Changes to the code directly affect the system. In this sense, Squeak is a "live" system. Figure 5a juxtaposes these concepts of Squeak with the main concepts of other development environments such as Eclipse. In case of Squeak (Figure 5b), the virtual machine starts up an image, which is (more or less) a memory dump of the running system. In case of Eclipse, for example, the development environment is a standalone

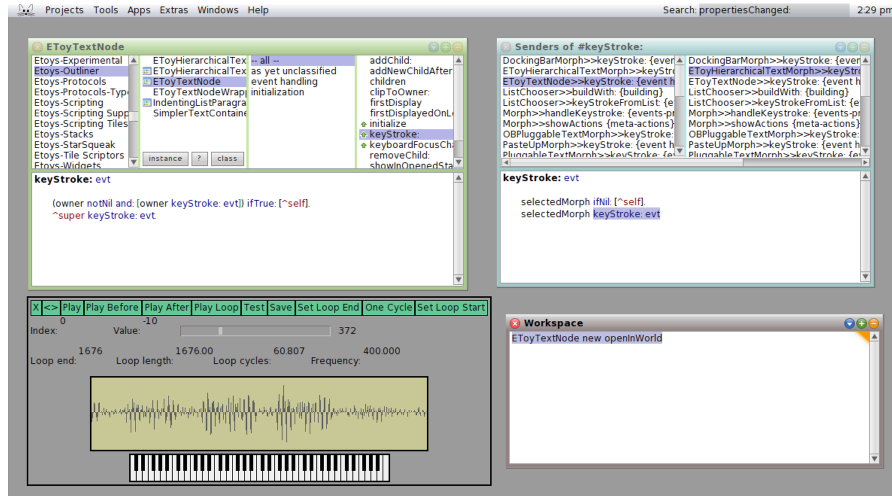
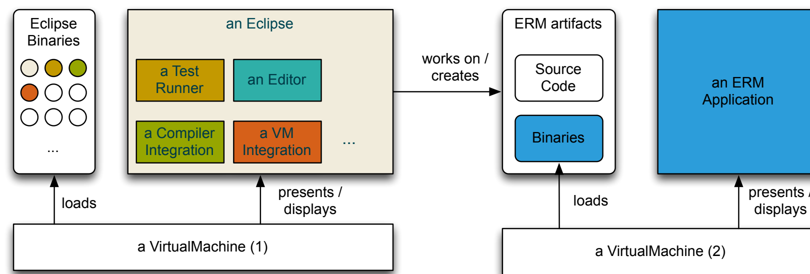
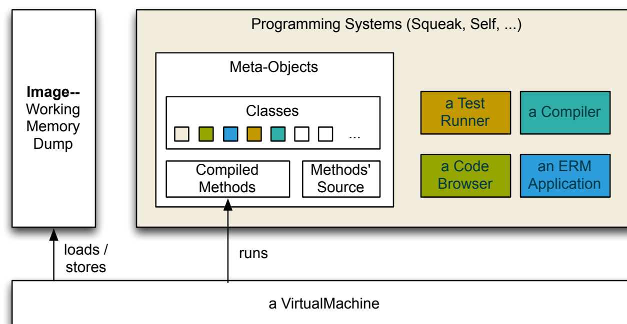


Figure 4: Screenshot of a running Squeak system having opened various tools. From left to right and top to bottom: a class browser, a browser for senders of selected messages (selectors), a sound application, and a workspace.



(a) IDE concepts in Eclipse



(b) IDE concepts in Squeak/Smalltalk

Figure 5: A comparison of Squeak and Eclipse

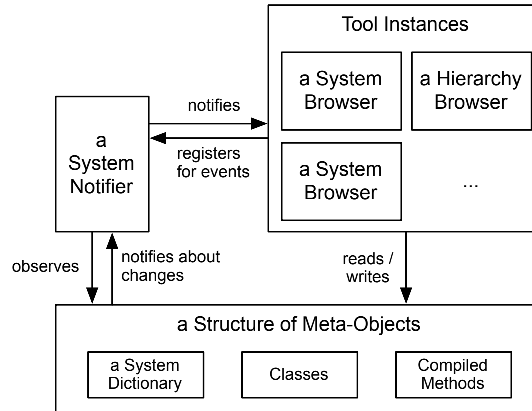


Figure 6: Development tools working on meta-objects in Squeak.

application that manages source code and binary files. The binaries can be used to open the application under development in a different process.

WORKING ON META-OBJECTS As the source code is part of the system in Squeak, the source code is managed through concepts also used for other applications, which are, in case of Squeak, the concepts of object orientation. Source code is thus represented as objects, which are called *meta-objects*. Figure 6 shows the main concepts.

There are two main kinds of meta-objects relevant for this work: meta-objects representing classes and meta-objects representing methods. In addition, there is a system dictionary, which maps class names to *Class objects*. Class objects have access to their definition including their superclass and the list of instance attributes. Class objects include a method dictionary that maps symbols (method selectors) to *CompiledMethod objects*. On one hand, CompiledMethod objects are arrays of byte code that can be interpreted by the Virtual Machine (VM). On the other hand, they also have access to the source code that was used to create them.

SYSTEM CHANGE EVENTS When a programmer changes the definition of class or triggers the compilation of a method, these changes directly affect the system. Changes to meta-objects are observed by a SystemNotifier, which propagates such change events to all development tools, among other, which can then update their views accordingly.

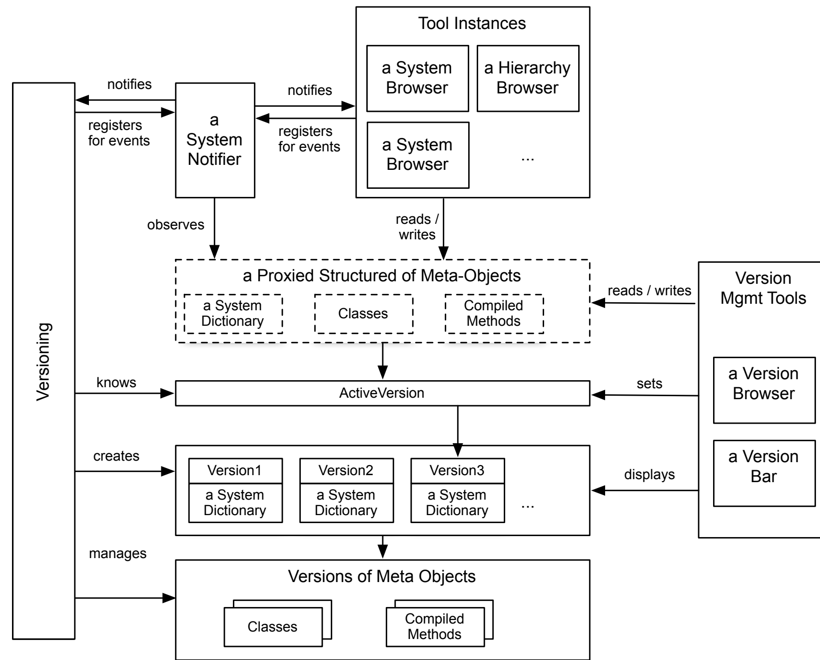


Figure 7: The main concepts of CoExist as an extension to Squeak.

4.2 CONCEPT OVERVIEW

CoExist is based on the insight that the risk for tedious recovery is caused by the loss of immediate access to previous development states. With every change, the previous version is lost, unless it has been saved explicitly. This version, however, can be of value in future development states, when, for example, an idea turns out inappropriate.

For that reason, CoExist preserves fast and easy access to previous development states and information thereof [77]. For every change to the code base, CoExist creates a new version. Programmers can rapidly switch versions or can access multiple versions next to each other. CoExist thus gives the impression that development versions *co-exist*.

Figure 7 illustrates the main concepts that have been added to Squeak by CoExist (compare with Figure 6). The access to meta-objects is now proxied. The proxies use the *ActiveVersion* object to redirect all accesses to the active version of meta-objects. The *versioning* facility listens for system change events, which are sent for changes to meta-objects. For every change event, the versioning facility creates a new version. CoExist provides access to the list of all

versions and offers tools to effectively browse the version history and identify versions of interest.

With that, CoExist contributes the following concepts and tools:

CONTINUOUS VERSIONING to create new versions in the background based on the structure of programs (Section 4.3).

USER INTERFACE CONCEPTS to support browsing and exploring version information as well as identifying a version of interest fast (Section 4.4).

ADDITIONAL ENVIRONMENTS to explore static and dynamic information of previous development states next to the current set of tools (Section 4.5).

CONTINUOUS AND BACK-IN-TIME ANALYSIS for test cases and other computations (Section 4.6).

RE-ASSEMBLING OF CHANGES for sharing independent improvements in separate commits (Section 4.7).

4.3 CONTINUOUS VERSIONING

CoExist continuously creates new versions in the background, which relieves programmers from the need to regularly commit meaningful development states explicitly.

4.3.1 *Structured Continuous Versioning*

NEED Programmers want to rapidly identify previous versions of interest. However, automatic versioning implies the absence of user-written commit messages, which are typically required to identify a previous development state of interest.

APPROACH To overcome this problem, CoExist's versioning approach is based on the structure of programs. This means, CoExist creates new versions when the user changes the program's structure or modifies a structural element. So when, for example, the user adds a new method, CoExist will create a new version. Next to methods, classes are the other main kind of structural elements in object-

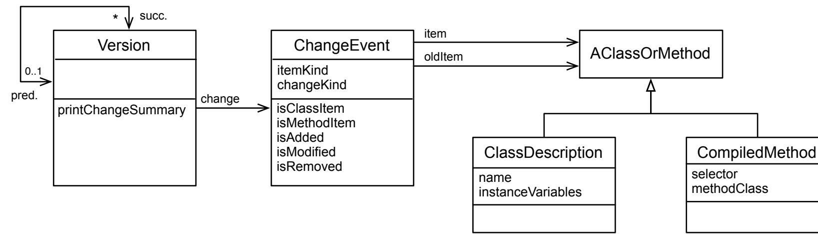


Figure 8: Versions representing changes to meta-objects.

oriented programming. Consequently, CoExist creates new versions for the following set of events classified along two dimensions:

- **Item Kind:** Class, Method
- **Change Kind:** Added, Modified, Removed

The information about the change event that led to the creation of a new version is recorded and linked to the respective version object as depicted in Figure 8. The recorded change event also holds a reference to the program element (item) that has been changed. If the element has been modified, the change event holds two references: one to the old version of the program element and one to the current version.

Recording such change events preserves useful information. For each version, it enables to understand what has been changed despite the lack of written commit messages. For example, version objects have sufficient information to print change summaries like shown below.

```
aVersion printChangeSummary. "->"
    'METHOD Person>>#fullname ADDED at 2:15pm'
```

DISCUSSION The idea of basing the versioning on the program structure is inspired by image-based systems such as Self [87] or Squeak/Smalltalk [38] and by versioning facilities such as Orwell [84] or Envy [57]. In an image-based environment, programmers do not edit files of source code. Source code is rather part of the development environment, which directly stores and manages the structural elements of the program and provides tools for browsing and editing these elements.

Note: While structured continuous versioning is a natural extension to image-based systems, the idea does not depend on specifics

of such systems. It can also be implemented for file-centric environments such as Eclipse or Emacs. In such environments, the unit of editing is a file and each file includes multiple program elements. One option to use structural information for versioning is to track the manipulation of program elements and the moves of the text cursor from one element to another. More specifically, when the programmer modifies an element and then moves the cursor to another one, the environment can make a commit. A similar concept has been implemented in the Engerize IDE [**?**].

4.3.2 *Implicit Branching*

need Programmers want to access a development state previously withdrawn.

approach When a programmer switches to a previous version to start over, subsequent changes will be recorded on a new branch that is implicitly created. With that, the changes that the user has withdrawn (and the corresponding development states) will remain accessible.

discussion The implicit branching mechanism is inspired from and similar to the undo-branches mechanisms of Vim[1] and Emacs[2]. It encourages programmers to try out new ideas from previous versions, as they are now free from considering the potential loss of valuable information.

4.3.3 *No Boundaries*

need Programmers sometimes want to change source code beyond the scope of their main project (or module) under development. The code artifacts that need to be changed might be covered by another (sub-)project. However, when programmers want to go back to a previous development state, for example, all changes should be withdrawn.

1 http://www.vim.org/
2 http://www.gnu.org/s/emacs/

APPROACH CoExist's versioning approach is not restricted to a particular artifact or project. It works on a global scope. Even when the user modifies artifacts of core libraries, CoExist will create a version for every modification and thus allows for withdrawing every change to the entire code base.

DISCUSSION This is different from undo/redo features and version control systems, which are typically scoped. Undo/redo capabilities of editors are scoped to documents. They enable to undo and redo the changes made within a particular artifact. VCSs use *projects* as the unit of versioning. They make it possible, for example, to revert all artifacts of a project to a previously committed state. CoExist's versioning, in contrast, has no restrictions. In trying out an idea, programmers can make changes to every artifact, and if, for example, the idea turns out inappropriate, they can withdraw the changes easily without having to coordinate the histories of single artifacts or projects. In this respect, CoExist complements the undo/redo and traditional version control facilities.

4.4 USER INTERFACE TO VERSION HISTORY

CoExist users sometimes require withdrawing more than a few recent changes. While the offered keyboard commands (shortcuts) are useful to undo and redo a few changes step by step, additional tools are required to efficiently deal with more complex recovery scenarios. Therefore, CoExist provides two tools that offer a graphical user interface to version information: the *version bar* and the *version browser*.

4.4.1 *Version Bar*

Figure 9 shows how Squeak's regular user interfaces has been extended with a version bar. Similar to a timeline, it displays all versions in chronological order. Newly created versions will instantly appear in the version bar. Hovering over the items will display additional information such as the kind of modification, the affected elements, or the actual change performed (Figure 10).

NEED Programmers sometimes have an interest in a previous development state that is related to particular source code element. For example, they might remember that the creation of a particular class

Figure 9: (From top left to top right) A programmer modifies source code, which implicitly creates items in the version bar. The filled triangle marks the current position in the history - the version that is currently active. When a programmer goes back to a previous version (bottom left), and then continues working, the new changes will appear on a new branch that is implicitly created (bottom right).

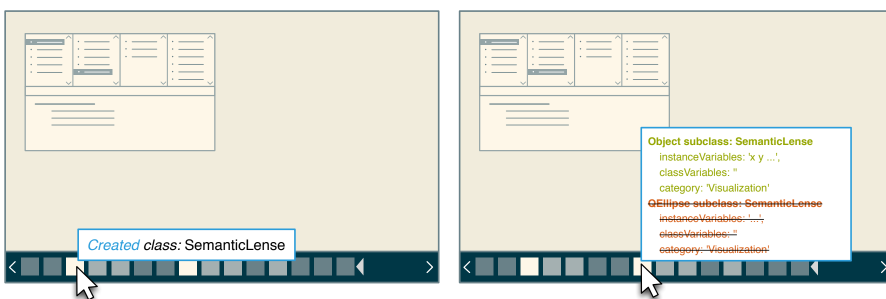


Figure 10: Hovering shows which source code element has been changed (left). In addition, holding shift shows the total difference to the previous version (right).

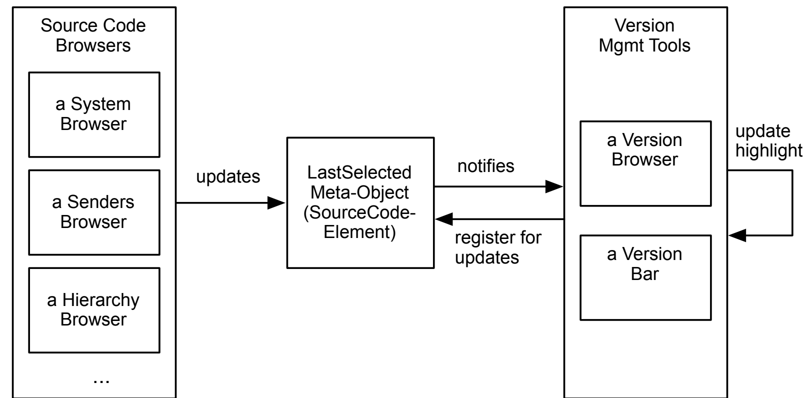


Figure 11: The integration and interaction of source code browsers and version management tools.

Figure 12: Selecting source code elements in developments tools highlights related version items.

happened close to the time were they started a series of changes that they want to withdraw now.

APPROACH: MAKING USE OF CURRENT SELECTION CoExist integrates tools such as the version bar with code browsing tools. As depicted in Figure 11, code browsing tools have been extended to notify version tools about the current selection made by the programmer. With that, the version tools know the meta-object (source code element) that has been selected last. The tools use this information to highlight versions that are related to the currently selected source code element (Figure 12).

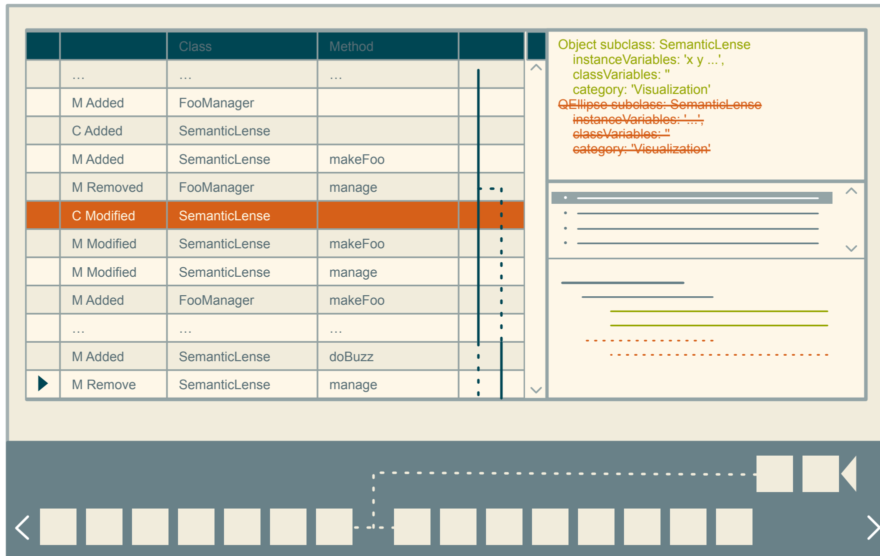


Figure 13: The version browser provides a tabular view on change history. Selecting a row shows the corresponding differences in the panes on the right.

4.4.2 *Version Browser: Efficiently Dealing with Numerous Versions*

NEED Having created a large amount of versions, programmers require support to quickly comprehend the recent history and identify versions of interest.

APPROACH: TABULAR VIEWS The *version browser*, which is illustrated in Figure 13, supports studying records of version information at a glance. It displays information about change events in a table view. As previously explained, CoExist creates new versions when programmers create, modify, or remove elements such as methods and classes. This structured versioning is sufficient to provide structured overviews as presented in the mockup tables of Figure 14. Each line starting with a number represents a version with an associated change. For example, version 1166 introduces the method `Person>>#name`. Provided with such a tabular view, programmers can quickly scan the history by means of the program structure and the used names. Our informal study suggests that the provided information helps identify a version of interest within a few seconds (see Section 5.2).

APPROACH: SEARCH AND FILTER The version browser provides a mechanism for highlighting elements or showing only a subset of all. Programmers can query the system with the name of the method

		Kind	Class	Method	Time
►	HRMan	10 items			
►	HRFin	2 items			

name

		Kind	Class	Method	Time
▼	HRMan	10 items			
	1164	Addition	Person		13:42
	1165	Modification	Person		13:42
	1166	Addition	Person	name	13:42
	1167	Addition	Person	birthdate	13:43
	1168	Removal	Person	birthdate	14:08
►		Renaming	Person	name ➡ firstname	
	1171	Addition	Employee		14:08
	1172	Addition	Employee	arrivalDate	14:08
	1173	Addition	Employee	salary	14:08
▼	HRFin	2 items			
	1174	Addition	Payment		14:09
	1175	Addition	Payment	amount	14:09

Figure 14: Two conceptual tables showing lists of versions. The table on top presents two summary lines that, when expanded, change the representation of the table to the one on the bottom. In such tables, the numbers in the first column are the version identifiers. Following are the columns for the kind of change that triggered the creation of the version, the class and method that changed, and the time stamp of the change.

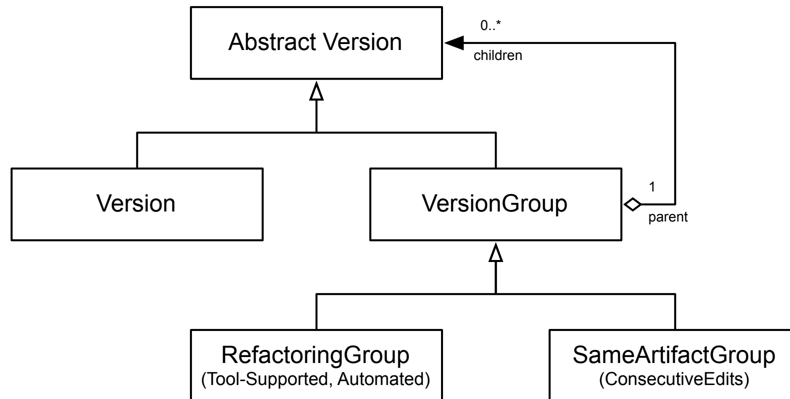


Figure 15: Composite Pattern for Groups of Versions

they remember (or a part of its name). CoExist will highlight the versions that matches, giving the programmer the possibility to focus on their neighborhood instead of scanning the complete list. The mockup tables presented in Figure 14 illustrate this concept: in the table on the bottom, the programmer entered 'name' as a query in the text field on the top right to highlight all changes related to the method `name`.

APPROACH: VERSION GROUPS CoExist's version browser also groups related versions into one summary line. For example, a developer might find it convenient to see the changes at the level of packages instead of a detailed list of all the changes in classes and methods. This is illustrated in Figure 14 where the table at the top presents summary groups for consecutive versions that are in the same packages (the packages are `HRMan` and `HRFin` in this case). Groups can be expanded by clicking on the triangle.

4.4.3 *Composite Versions*

NEED While IDE commands that automate standard code manipulation procedures are triggered by only one command, they will often lead to multiple changes to the source code, which will be reflected by the creation of multiple versions in CoExist. An extract method refactoring [29], for example, will result in a version that represents the addition of the new extracted method and in a subsequent version that represents the method modification replacing the extracted code with a call to the added method.

APPROACH CoExist groups series of tool created changes and presents them as composite versions. The version history illustrated in Figure 14, for example, includes a composite version for a tool-supported renaming of the method `Person>>#name`. The corresponding line entries, the example titled 'Renaming', can be expanded to study the involved changes.

DISCUSSION Composite versions are beneficial in two ways. First, they represent the actual command that has been triggered. This helps remember what has been done and avoids speculation about whether a series of changes have been caused by a single command such as a refactoring. Second, composite versions can be visualized as single entries summarizing the multiple changes, which reduces the overall number of elements to be shown and to be scanned by the user.

To detect that a series of changes has been created through a single IDE command, CoExist integrates with the refactoring engine so that it gets informed about both the beginning and end of automated source code manipulations. CoExist attaches this information to all versions created during the execution of the command, which enables tools such as the version browser to represent the involved versions as a composite entry.

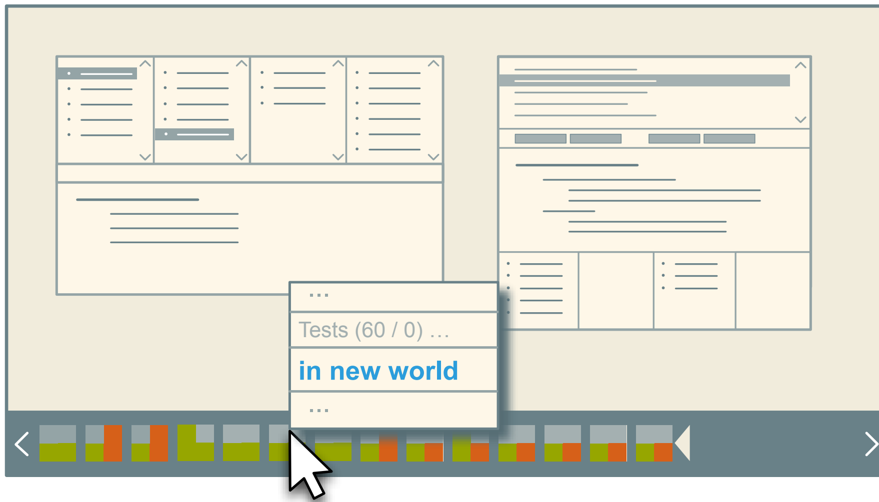
4.5 ADDITIONAL ENVIRONMENTS

To support *knowledge recovery* tasks, CoExist provides a mechanism to work with additional environments.

4.5.1 *Working with Additional Environments*

NEED Programmers want to study the development state of previous versions and compare it to the current one.

APPROACH Programmers can open an additional *working environment* to explore a previous version, as shown in Figure 16. Each working environment has its own active version pointer, as illustrated in Figure 17. With that, each environment accesses the meta-objects of the associated version. Similar to the default environment, additional



(a) Using context menu to open an additional working environment for a previous version.

(b) Browsing code of a previous version (left side), having highlighted (light blue lines) those code elements that differ to the current version (right side).

(c) Debugging the program in two different versions simultaneously supports understanding the differences at run-time.

Figure 16: Working with an additional fully functional working place for a previous version, next to the tools for the current version.

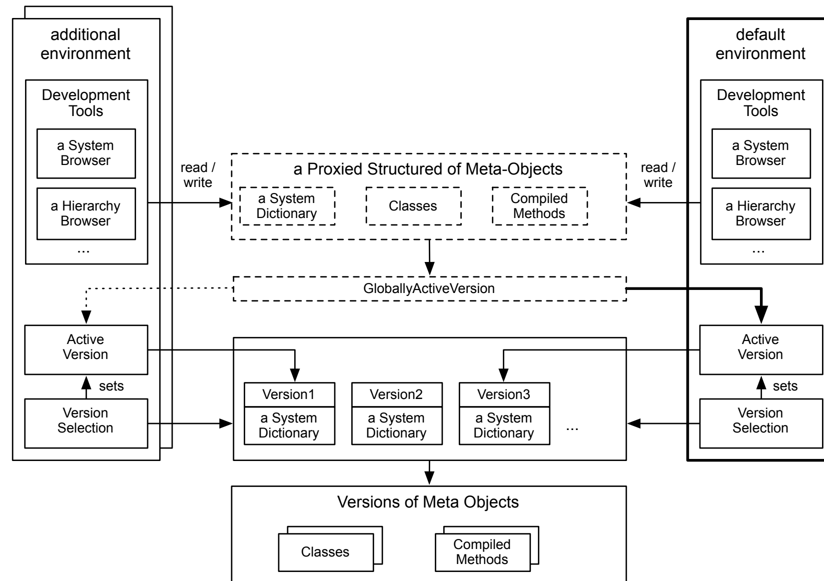


Figure 17: Different environments of tools, each working on a different program version.

environments also allow for switching back and forth between versions.

Every additional environment is a fully functional. Programmers can browse source code, run applications and analyze their run-time behavior using debuggers. The full access to previous development states makes it easy to get an in-depth understanding on how the system parts worked together previously. For example, programmers can debug two versions of a program side by side and compare the run-time behavior (Figure 16c).

To help programmers identify the similarities and difference of two versions, development tools highlight those source code elements that are different in both versions, as illustrated in Figure 16b.

DISCUSSION Providing programmers the possibility to open and work with multiple environments has several benefits:

- It preserves the state of the current environment and its setup, which avoids the need to re-open tools, to switch back to previous views, or to clean up from the recovery task. It avoids the trade-off whether going back to a previous version is worth its effort, which in turn reduces speculation.

- It supports recovering information incrementally. Once the version is identified and opened, the programmer can easily switch back and forth between the two environments and even look at them simultaneously. This avoids the need for considering whether everything that will be needed has been studied carefully and is remembered before switching back to the most current version and continuing work.

- It supports juxtaposing alternative solutions. Multiple environments make it easy to examine the benefits and drawbacks of multiple solutions for a particular problem simultaneously.

- It reduces the costs of task switching. Programmers sometimes need to interrupt a long running task to work on an important bug report, for example. In such situations, programmers can open an additional environment to work on the bug fix, which has the advantage that the current tool setup can remain as is so that continuing the interrupted thread of work will be easier.

4.5.2 *Replication of Views*

NEED When programmers open an additional environment on a previous version, it is likely that they want to recover information about the currently selected source code element.

APPROACH CoExist tries to replicate the current working context for a newly opened environment. It will detect the currently active tool and ask the new environment to open the same tool. Thereby, the new tool instance is requested to present the appropriate version of the content that the original tool instance currently presents, if possible. When, for example, the programmer currently inspects a method definition in a system code browser, CoExist will open another code browser trying to show this method in the previous version of interest. If this method is, however, not available in the selected version, CoExist will try to show its structural parent instead, which is, in this case, the class containing the selected method.

DISCUSSION This idea of replicating the active context is straightforward for many development tools such as code browsers or test runners. The current state of the tool needs to be detected, which mainly consists of the current selection, and then needs to be applied to the new instance. *However*, for some tools, replicating the current

state is conceptually more complex. For example, the possibility to replicate a debugging session to a previous version would be of great use, but it requires further research to understand how to deal with differences in program execution, how to react to them and how to present them.

4.6 CONTINUOUS AND BACK-IN-TIME ANALYSIS

To support analyzing the impact of changes in retrospective, CoExist offers two complementary approaches: *continuous analysis* and *back-in-time analysis*. While both approaches are applicable to a variety of analysis techniques such metrics or performance, the following presentation will focus on testing.

4.6.1 *Continuous Analysis*

NEED After focusing on an idea and making several changes, programmers now want to understand the impact of every change on test results.

APPROACH CoExist incorporates the idea of *continuous testing*[3] [62] by running analysis scripts for every newly created version. In particular, CoExist continuously runs a selected set of tests. CoExist advances the original idea of continuous testing by recording all analysis results along with their corresponding version objects, so that all results are kept for the programmer to be studied when interest suggests doing so.

Figure 18 illustrates the main concepts of this approach. Programmers define analysis scripts that should be run continuously. For every version, script runners are started to run these script in the context of the newly created version. Results will be collected and linked to the corresponding version object.

Concerning test analysis, CoExist visualizes the recorded results in the version bar, as depicted in Figure 19. The visualization reports the test status for every version entry and highlights when a change

3 *Continuous testing* refers to the approach of running a set of automated unit tests continuously in the background. Programmers are notified when the last change has caused tests to fail. This automation avoids the need to manually trigger test execution and to wait for the test results.

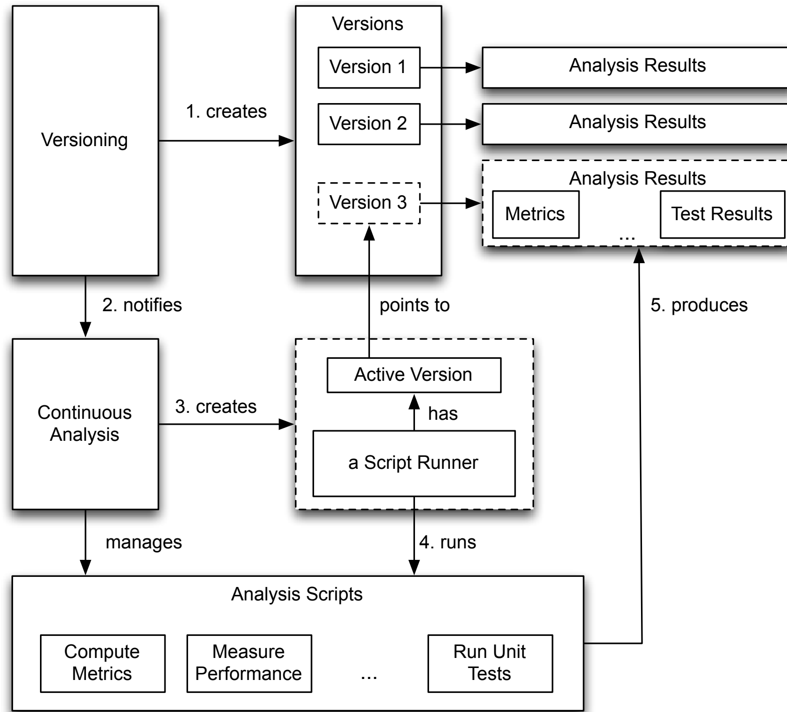
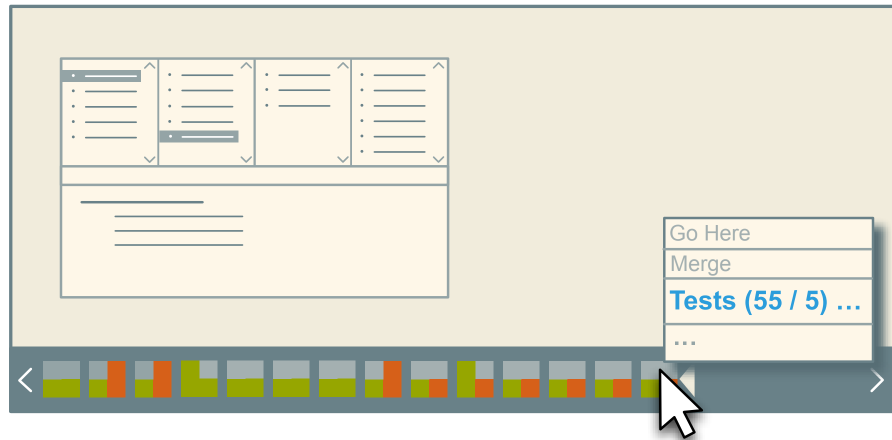


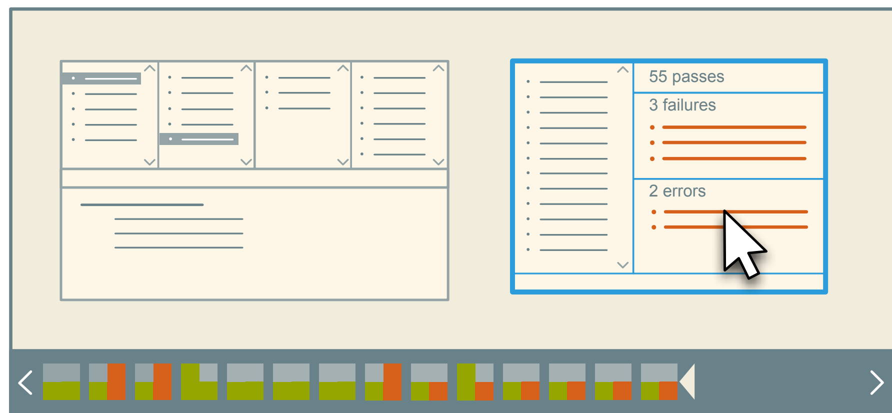
Figure 18: Continuous Analysis Concepts

caused tests to fail or a change made tests pass. By interacting with version items, programmers can study relevant test results and the corresponding change. They can open a test runner to inspect the results in more detail. If the programmer requests the test runner for the most recent version, it will be opened in the current environment. If the programmer requests the test runner for a former version, CoExist will first open an additional environment for this version, and then open the test runner inside, so that the tests can be re-executed in the corresponding version of the code base.

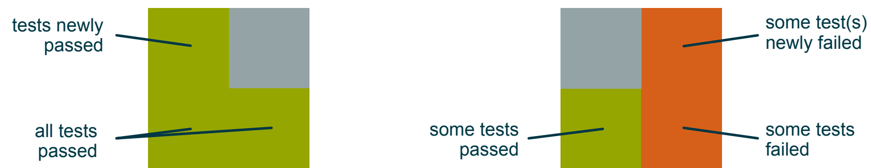
DISCUSSION This tool support avoids the overhead of manually executing tests and, in addition to that, it allows for ignoring test quality without inducing drawbacks since all meta-data on how the system changed is kept and available. This invalidates a main argument for constantly assessing test quality, namely that bugs get harder to locate and repair the more time passes until their discovery. Even when tests fail, programmers can continue making changes. They will remain able to easily identify those changes that caused tests to fail.



(a) Hovering over a version item to get the context menu, which includes test statistics.



(b) After opening the test running via context menu, it directly shows the test result and allows the programmer to further inspect the result, re-execute them, and debug them.



(c) The meaning of test result visualization using two examples.

Figure 19: Tests are run in the background for every version. The results are recorded and visualized for every version item.

4.6.2 *Back-in-time Analysis*

In addition to continuously running tests, CoExist provides an interface for manually running tests on recorded versions.

```
[IntegrationTest buildSuite run]
    valueBasedOn: aVersionOfInterest.
```

need Programmers might want to avoid running all test cases continuously, because it can decrease the computer's overall responsiveness. But they will want to be able to locate introduced bugs quickly.

approach Programmers can request CoExist to run a failing test case on a range of versions to identify the change that first caused the test to fail. To identify the change rapidly, CoExist will use a binary search strategy.

```
self versionControl bisectionAnalysis:
    [IntegrationTest buildSuite run defects notEmpty].
```

This facility for back-in-time analysis assumes that the source code of the test cases to be executed is available in all versions. It raises an error if this is not the case. However, an extended interface accepts a list of source code elements, whose definitions will be applied to each version before running the tests. (Making these changes will implicitly create branches.) This extended interfaces makes it possible to run newly written tests back-in-time, which will be of use, for example, when programmers discover erroneous behavior in the application that is not yet covered by a test case. In this case, the list of source code elements to be applied will include the new test method and newly added utility methods.

discussion CoExist's back-in-time analysis facility builds on previous work. The distributed version control system Git, for example, provides a command line tool called "git-bisect". It requires a shell script that can tell whether a version is *good* or *bad*. Using a given script, this tool finds the commit that introduced the bug (the bad case) by performing binary search on all versions. In comparison to Git, CoExist integrates such features into the IDE and provides dedicated support for running tests. CoExist makes it particularly easy to run or even debug test cases on previous versions. Apart from that, CoExist is conceptually different by having a continuous change record, which is more dense and fine-graned than a typical manu-

ally maintained record. And this continuous change record helps programmers to easily identify the cause of problems, which in turn avoids the need for constant quality control. It provides programmers the opportunity to focus on trying out their ideas.

4.6.3 *Impact Analysis Beyond Tests*

CoExist's facilities for continuous analysis and back-in-time analysis are not restricted to automated test case. While test feedback is often a crucial aspect to software development, there are other important aspects such as complexity metrics and performance. An undesired deviation in performance or in any other metric can be as hard to understand and to fix as a failing test. Since the impact of source code changes is not obvious, programmers will benefit from continuous analysis and back-in-time analysis regarding these aspects. For that reasons, the interfaces of CoExist support the execution of arbitrary computations, which might trigger test runs or performance measurements. The only factor that limits the possibilities, in particular for continuous analysis, is the computational power available.

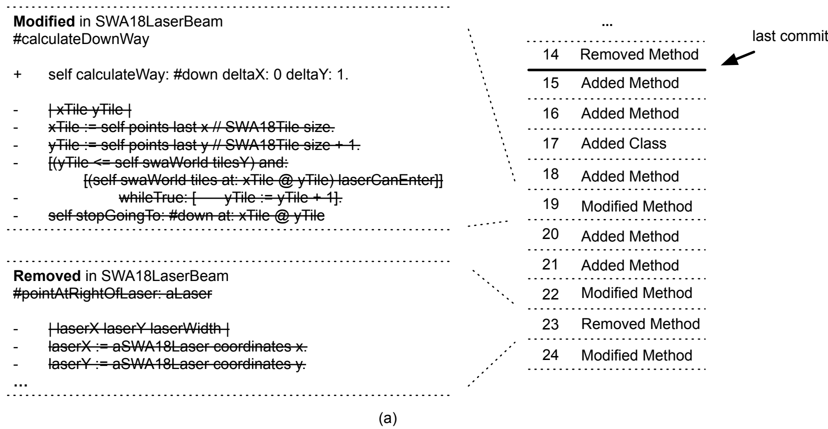
4.7 RE-ASSEMBLING CHANGES

CoExist supports re-assembling changes in two respects: it provides a *fine-grained change history* and support for *re-applying selected changes* on another branch.

4.7.1 *Fine-grained Change History*

NEED Programmers want to get an overview over the recent changes made until the last commit and recall the distinct increments they worked on.

APPROACH CoExist records a fine-grained change history and collects meta-information about every change, which can be presented in tabular views like, for example, in the version browser. Besides a tabular view of meta-information, programmers can inspect the differences for each version (Figure 20a). By inspecting such a record of changes, programmers can recognize the different increments they



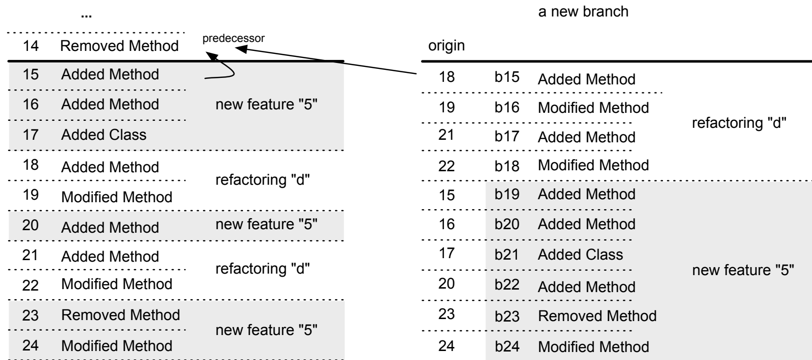


Figure 20: Extracting meaningful increments by first studying the change history and then re-applying selected changes to another branch.

Figure 21: Top row: Selecting two changes and re-applying them to the currently active version (triangle). Bottom row: Withdrawing two changes, which results in re-applying subsequent changes to the change before the selection

have worked on and understand which changes contribute to each of these increments (Figure 20b).

DISCUSSION While VCSs such as Git can show the difference between the current status of the project and the status of the last commit, CoExist has a record of the individual changes (meta-information) that led to this difference. Hence, tools such as the version browser can better help programmers to reconstruct knowledge about the change history.

4.7.2 *Re-applying Selected Changes*

NEED Programmers want to extract distint increments to tests them in isolation and share small self-contained commits with others.

APPROACH CoExist allows programmers to re-apply changes at a different place in the history, which is commonly referred to as *cherry picking*. Re-applying changes creates new versions. When changes

are applied to a version that already has a successor, a new branch will be created implicitly, which avoids changing the current history. Figure 21 illustrates how programmers of CoExist can select and re-apply changes. CoExist also provides a utility command for withdrawing selected changes, which will re-apply all subsequent changes to the change right before the selection.

In the example illustrated in Figure 20c, programmers can first extract all changes that appear to belong to the refactoring task, which will create a new branch. To perform completeness or correctness checks, programmers can open the created development state in an additional environment. In addition to the automatic test runs, they might want to run integration tests. If programmers consider the improvement a meaningful increment, they can share it with other programmers by pushing it to the used source code repository.

discussion Git also provides supports for extracting small commits out of many changes. Using the interactive mode of the "git-add" command, programmers can select those modified files that shall be included in the next commit. Moreover, for every selected file, they can select the modified lines ("hunks") to be inlcuded or exlcuded. However, CoExist improves on this command in two ways. First, instead of selecting lines ordered by their position in a file, programmers of CoExist can re-apply coherent changes of a record that is based on the program structure. Second, using CoExist, the extracted increments can be checked for completeness and correctness before being committed.

summary

CoExist is based on the idea of continuous versioning: Any change made to the system creates a new version, which holds the change made as well as a complete snapshot of the current system. The granularity of versioning is in accordance with the program's structure. With that, each version is implicitly associated with valuable meta-information about its creation, namely the source code entity that has been added, modified, or removed for its creation. This information is presented in history browsers to assist programmers in identifying previous versions of interest. The implicitly gained meta-information avoid the need for explicit commit messages required otherwise.

By opening additional working environments on previous versions, programmers can browse, modify, run, and debug source code in the

same way as in the default environment. This supports recovering knowledge from previous versions without giving up their current working set.

CoExist integrates continuous testing and advances this concept. It continuously runs tests for every created version in the background. Test results are recorded for later inspection. With that, programmers can identify the changes that introduced bugs with only little effort and can thus defer the consideration of test quality. Continuous analysis is not restricted to test cases: programmers can tell CoExist to run other computations such as benchmarks, for example. In addition to the continuous analysis, programmers can also use the versioning facilities for back-in-time analyses.

5 CONCEPT EVALUATION

This chapter discusses the CoExist concepts presented in the previous chapter. The first section describes how CoExist would have supported the student in the reported case study (Section 3.1). The subsequent section presents three informal user studies, which were conducted to help discover usability problems and opportunities through the development course. The last section evaluates how CoExist supports the various recovery needs. It is argued that tools such as CoExist reduce the need for problem prevention.

5.1 HOW COEXIST HELPS IN THE EXAMPLE CASE

Were CoExist available to the programmer in the example case, (Section 3.1) had have available CoExist, recovery would have been fast and easy. Figure 22 shows the situation before the `SemanticLens` class is extracted and how new version items appear for the creation of this new class.

Figure 23 shows how the programmer turns the subclassing solution into delegation and new version items are created correspondingly.

When the programmer then becomes aware that subclassing is actually preferable over delegation, they can use CoExist to get back to the point before subclassing has been turned into delegation. The Figure 24 illustrates one way to get back to the desired development state. First, the programmer makes use of the interaction between code browsing tools and the version bar: Selecting the `SemanticLens` in the code browser highlights all changes that are related to this class, as show in Figure 24a.

Hovering over the highlighted version items exposes the version item that represents the initial class addition and the item that represents the later modification of the `SemanticLens` class (Figure 24b). Holding shift while hovering over the latter item reveals detailed diff information (Figure 24c), which confirm that this version item marks the turning of the inheritance solution into delegation.

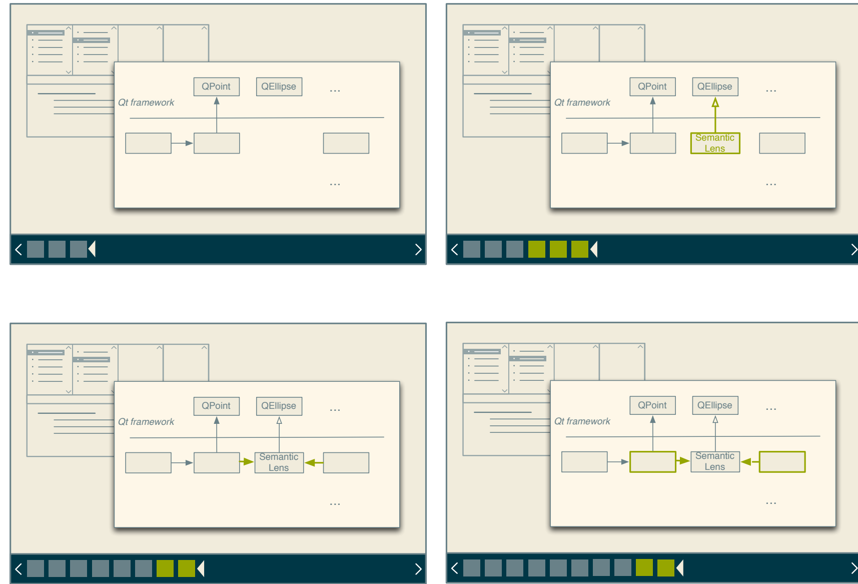


Figure 22: Visualization Task with CoExist, Extracting a New Class

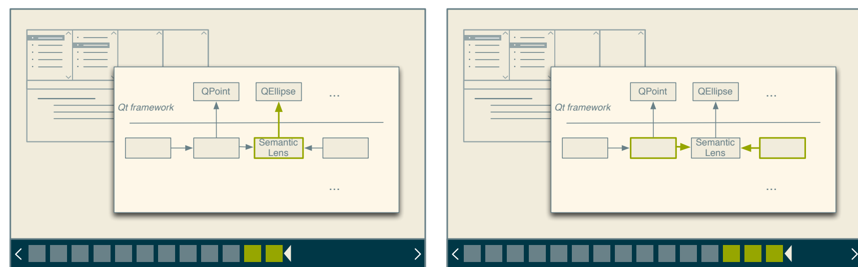
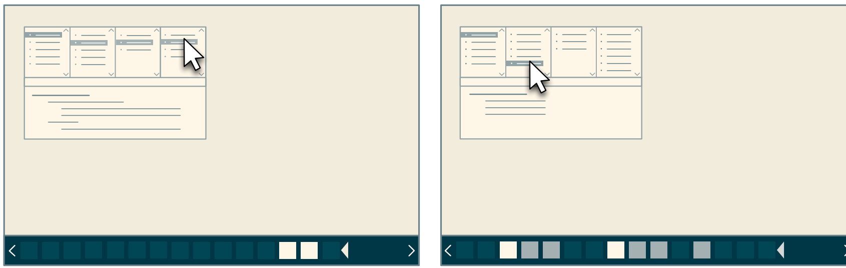
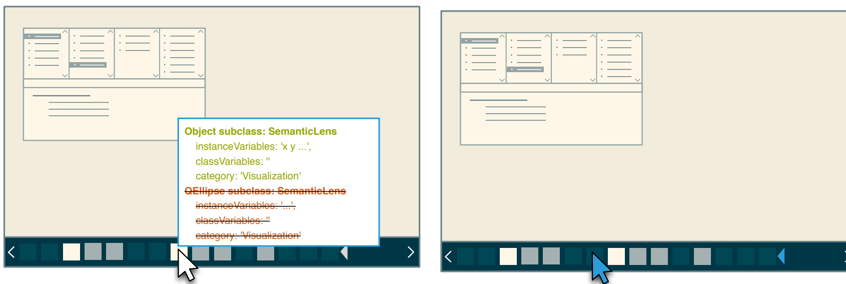


Figure 23: Visualization Task with CoExist, From Inheritance to Delegation



(a) Interaction between code browser and the version var.

(b) Hovering version items to restrict selection.



(c) Shift-hovering to get a detailed diff,

(d) Going back to the selected version and continue developing.

Figure 24: Visualization Task with CoExist, Recovery: Going Back to the Subclassing Solution

The programmer can now select the version item previous to that version to be the current version, which is emphasized with the blue triangle in Figure 24d. This will withdraw all changes that came after this version. The code base is now back to a desired development state and programmers can continue working from there.

5.2 INFORMAL USER STUDIES

During our work on CoExist, we regularly asked members of our research group to use our system. We did informal user studies in three phases of our research project: (1) for an early prototype that supported going back and forward one step at a time, (2) for a midterm prototype that improved speed and provided access to any existing version, and (3) for the final prototype.

5.2.1 *Early Prototype*

STATUS: The early prototype provided only a simple undo/redo mechanism. Programmers could go back and forward by triggering commands. No tools that show the recorded versions or details of them were available.

PURPOSE: The early prototype enabled programmers to try out the idea instead of only talking about it. It is often the case that programmers need see a new tool and try it out in order to understand that a demand, which they previously did not notice, has always been there. The goal of gathering feedback at this early stage was to learn whether other developers would confirm the need for such tool support, once they have seen and used it.

PROCEDURE: Three students were asked to try out the prototype. Participants received a Squeak image Two that included the prototype and a small game application. After a short demonstration of the prototype, participants were instructed to change and extend the game in an explorative manner and make use of the global undo/redo commands. The proposed approach was exemplified by suggesting removing code artifacts, running the game, observing the effect, and withdrawing the changes afterwards. While removing code will often bring up the debugger, it can help understand how the removed functionality is expected by other parts of the systems.

results: Two of the three participants were enthusiastic about the prototype and confirmed its potential value. While they could not retrieve details from their memory, they had the feeling that such tool support would have been useful in various development scenarios they had experienced previously. However, one participant did not share the enthusiasm and was skeptical about the value of the prototype. In particular, he doubted the idea of removing code to see what the effects will be. These results might indicate an inappropriate study setting or that the tools. Furthermore, the student disliked that the effect of the global undo/redo was not sufficiently transparent and observable, though he was unsure about his observation.

5.2.2 *Midterm Prototype*

status: Whereas the early prototype offered simple undo/redo commands, the midterm prototype provided a version bar to identify and go back to any version of interest. The midterm prototype also provided the possibility to open additional working environments. Finally, this prototype significantly improved responsiveness by preserving direct access to previous versions of meta-objects instead of only versioning source code.

purpose: This study should further substantiate our assumption that programmers require additional recovery support. One goal was to examine whether programmers experience recovery needs when they perform explorative programming tasks. Another goal was to study how well the current prototype supports programmers in their recovery needs. For example, we were wondering whether the current design of the version bar is sufficient to rapidly identify previous versions of interest.

procedure: One student, one PhD student, and one post-doc were asked to spend one to two hours improving the design of a 2D puzzle game named MarbleMania, implemented by undergraduate students in the context of a previous lecture. The game worked properly and was covered by 39 unit tests (all of them passing) but implementation showed room for many improvements. At the beginning of the experiment we introduced the game play and gave a conceptual overview of the implementation. Then we asked our subjects to study the source code and to freely refactor the pieces that they felt needed improvement. We encouraged the subjects to implement their ideas as they came to mind without evaluating them

mentally. One evaluator sat next to each subject to take notes and to answer questions about the proposed prototypes. We also used screen recording for later analysis.

RESULTS: All three participants experienced the need to start over from a previous development stat. Two participants went back to previous version two times and one participant, one time. For example, participants tried out initial ideas to improve the program design and by doing so learned about the limitations of these ideas. This knowledge helped participants develop better ideas, which they realized after withdrawing the previously made changes. This observation supports that programmers can benefit from tool support that makes recovery easy.

This study also highlighted the need for juxtaposing. Two subjects opened a previous version in a separate environment to study aspects of this version. One of them wanted to study a test execution in a previous version. To do this, he opened an additional environment, found the test, opened a debugger, and stepped to the place of interest. This led to the development of the replication feature that makes it simple to re-create the currently active view for a different version (with just one click).

In addition to that, we discovered that the version bar is not sufficient for rapidly finding a version of interest, because it lacks an overview of version information. This motivated the creation of the version browser.

Furthermore, the subjects felt positive about the tools. Even some weeks after the studies, subjects sometimes dropped in and reported a situation where the proposed tools would have helped for their own project.

5.2.3 *Current Prototype*

STATUS: Compared to the midterm prototype, the current prototype includes the version browser (Figure 13), and provides features such as search, highlighting, and grouping.

PURPOSE: This study focused on the usability of the version browser. The goal was to find out whether programmers can quickly identify a previous development state, when they suddenly en-

counter the need. In contrast to the version bar, the version browser presents various meta-information for multiple versions at a glance. This should help programmers to scan the history quickly for relevant pieces of information. We wanted to test this idea.

procedure: The need for going back was induced artificially. We asked two students to perform a pre-defined refactoring task on the MarbleMania game. More specifically, they had to eliminate an unnecessary observer indirection. After approximately half an hour of work, participants were interrupted and asked to go back to the version where they started. However, the correct version was not the first in the list. In the provided image, the history already contained a number of changes. The time stamps of those changes were adjusted as well, so that time information is no good indicator to identify the version of interest.

results: The first subject remembered the source code elements he changed first when he started the refactoring. Because the version browser shows the names of each modified element, the subject was able to find the target version within a few seconds. The second subject produced significantly more changes during the evaluation, which made the task of finding the target version more difficult. As a result, the subject took around 20 seconds to identify the target version. During that period of time, the subject first identified a range of candidate versions and then inspected the details of the associated changes. However, both participants were able to identify the version of interest with only limited time and effort.

5.2.4 *Discussion*

These informal user studies helped understand the benefits and problems of both our approach and implementation. We understood the need for responsiveness to make the tools attractive. We also discovered the need for the version browser. Indeed, the information presented by the version bar was insufficient to identify a previous version of interest.

5.3 FROM PROBLEM PREVENTION TO GRACEFUL RECOVERY

The purpose of employing best practices is the avoidance of tedious recovery work. However, CoExist enables programmers to easily deal with such recovery situations:

- When programmers suddenly realize that their current idea is inappropriate and it would be better to pursue another idea, they can easily go back to a previous development state and continue working from there on. Programmers can easily go back and thus withdraw recent changes even when the former state of interest has not been committed explicitly. When programmers later realize that the first idea is actually preferable, they can go back to the state before they decided to discard this idea. Thus, programmer can explore an idea by trying it out. They can make the corresponding changes and thereby learn about the implications. If the results are not sufficient, it requires only little effort to start over.

- When programmers want to start over but want to keep some of their recent changes, they can explore their change history in detail and transfer (copy) all changes of interest to another branch. CoExist supports programmer in "picking the cherries" of their work. *Also*, programmers might want to mix tasks and, for example, want to perform a refactoring in the middle of another task because it seems beneficial. Programmers can do so without hesitation, because they can easily re-assemble the various changes later.

- When programmers suddenly realize that their understanding of the code base has been insufficient, and that they now miss information that has already vanished due to recent changes, they can easily recover the required knowledge. They can quickly open previous versions of the code base in order to study the source code as well as the program behavior at former development states. Programmers can recover information while still having direct access to the current development state and the current tool setup, so that they can continue working immediately after they have found the missing pieces of information. Such tool support reduces the need to be careful and to ensure a sufficient understanding of the code base upfront. So, if programmers feel it is good to do so, they can start working on the task to see how far they will make it, and if need be, they can easily have a look at previous versions.

- When bugs creep into the code and remain unnoticed for a long time, CoExist helps programmers quickly identify those changes that caused the faulty behavior. Being equipped with such tool support, programmers can ignore quality assessment without having serious disadvantages.

- When programmers successfully complete an improvement and are ready to commit it, but then suddenly realize that their recent changes actually comprises several independent increments that should be shared separately, CoExist will support them in re-assembling those changes. They can explore the change history, select the changes that belong to one increment, for example, to a refactoring that simplified a feature implementation, move them to a new branch, check if the created development state is complete and functional, commit the diff, and move on to the next changes. Programmers thus have to take no care of working only on one increment or that they consider committing at the right point in time.

While a lack of tools such as CoExist requires means of problem prevention to avoid tedious recovery work, the availability of such tool support reduces the need and effort for means to prevent recovery scenarios.

SUMMARY

CoExist would have helped the student in the reported case study. For example, through interaction with the code browser and the version bar, the student would rapidly identify the version right before he started to change from subclassing to delegation. Informal user studies further suggest that the provided tools generally enable programmers to rapidly identify previous versions of interest. The need for responsiveness is fulfilled. Furthermore, the discussion shows how CoExist makes various recovery needs easy and fast to accomplish. It thus reduces the need for best practices to prevent problems.

6 IMPLEMENTATION

This chapter highlights selected aspects of CoExist's implementation. The explanations provide technical details on the concepts presented earlier in chapter 4. The class diagram in Figure 25 presents an overview of the most important classes and methods that contribute to CoExist. The classes on the left (bottom) were newly added to the system, the classes on the right (top) are standard classes that were extended.

The first section describes how a dynamic variable is used to provide access to different versions in the same environment. The subsequent section explains the versioning of meta-objects to provide immediate access and the sharing among versions to reduce performance overhead. Sharing of meta-objects requires special treatment of

reference between them. The corresponding changes made to the image, the compiler, and the virtual machine are explained. After this, attention is drawn to the the current limitations followed by a brief evaluation of performance characteristics.

6.1 RESOLVING ACCESS TO THE ACTIVE VERSION

Figure 26 shows how the same code snippet is executed in two different versions and evaluates to different results. While the left workspace is opened in an environment pointing to version 6, the right workspace is embedded in the root environment pointing to version 14. The class `Person` used in this example contains different sets of methods in the two versions. Hence, sending the message `selectors` to this class, which returns the set of selectors for which the class has corresponding methods, evaluates to different results.

Figure 27 shows a simplified graph of the visual objects (morphs) that are involved in the example scenario. The `World` morph is the root. It directly contains the workspace morph at the right and the additional working environment at the left, which directly contains the left workspace morph. As the state of the objects in the diagram

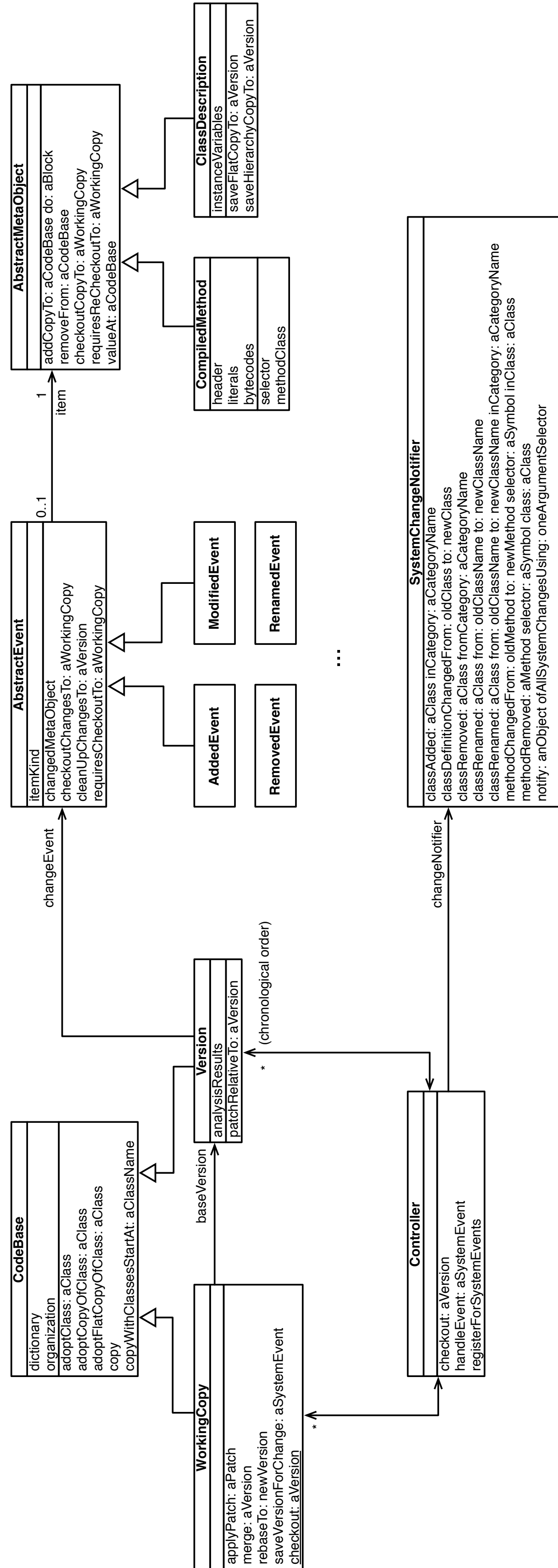


Figure 25: Main classes and methods of the CoExist implementation.

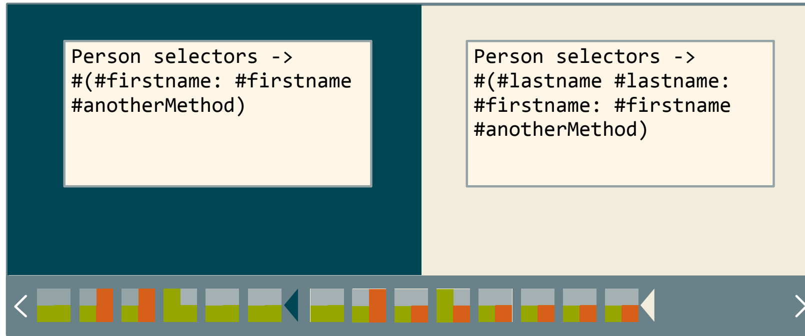


Figure 26: The same code snippet run in the context of two different versions evaluates to different results.

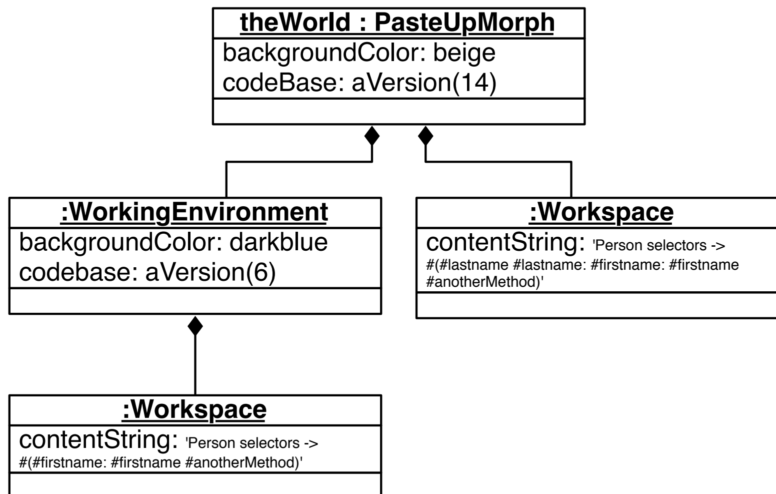


Figure 27: Graph of visual objects (morphs) shown in the conceptual screen above.

illustrates, the `World` morph is associated with version 14 and the additional working environment is associated with version 6. The code base associated with a morph defines the execution context for this particular morph and all its submorphs unless being redefined.

Whenever the programmer interacts with the system, code is executed. This code runs in the context of a particular morph, for example, in the context of the `World` morph. Squeak runs a main loop. In each cycle, it processes events such as keyboard and mouse events. Afterwards it redraws the scene graph.

The code listing shows two main entry points into the execution of the Squeak machinery. These entry points (`doOneCycle` and

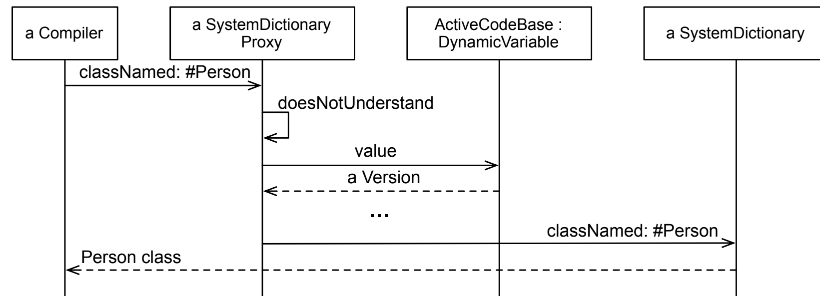


Figure 28: Accessing the currently active system dictionary by resolving the dynamic variable ActiveCodeBase.

`handleEvent`) have been modified so that the code base of the corresponding morph is assigned to the dynamic variable *ActiveCodeBase*. When, for example, the programmer triggers code execution in a workspace, this workspace will handle the corresponding system event because it has the focus. Afterward, all code executed in the context of the dynamic variable has access to the assigned code base.

```
Morph >> #codeBase
    ↑ self valueOfProperty: #codeBase
        ifAbsent: [self owner
            ifNotNilDo: [:ea | ea codeBase]]]

PasteUpMorph >> #doOneCycle
    ActiveCodeBase
        value: self psCodeBase
        during: [worldState doOneCycleFor: self]

Morph >> #handleEvent
    ActiveCodeBase
        value: self psCodeBase
        during: [↑ anEvent sentTo: self]
```

Diagram 28 and the code listing below illustrate how the dynamic variable `ActiveCodeBase` is used to resolve the currently active system dictionary. Before the code snippets shown in Figure 26 are executed, they are compiled. The compiler resolves symbols such as `Person`. Therefore, it sends a corresponding request to the system dictionary. Since the global system dictionary has been replaced with a proxy, this proxy can handle the dispatch using the `ActiveCodeBase` variable.

```
SystemDictionaryProxy >> #doesNotUnderstand: aMessage
    | currentVersion |
    currentVersion := ActiveCodeBase value.
```

```
↑ aMessage sendTo: currentVersion dictionary.
```

6.2 PRESERVING META-OBJECTS FOR ALL VERSIONS

For CoExist to be useful, it has to provide both *immediate* and *full access* to any version of interest. *Immediate access* implies that it should be easy to get the desired information, and that the information is provided fast. According to recommendations such as in [67, Ch. 10], CoExist needs to fulfill user requests that are common in less than *two seconds*. If programmers have to wait too long to get access to a version they might refrain from applying the proposed approach. *Full access* to any version refers to the idea that programmers should be able to explore, modify, run, and debug any program version. The goal for immediate and full access is important for programmers but also for CoExist itself, because CoExist runs unit tests on all versions independently of what the programmer is currently working on.

These requirements suggest an implementation strategy that holds available all program artifacts of all versions in favor of a strategy that requires re-computation or recompilation. So, instead of having to apply a sequence of changes to a baseline in order to restore a previous development state, it is beneficial to preserve the meta-objects such as classes and methods for each development state.

6.2.1 *Sharing Meta-Objects Among Versions*

To limit memory consumption, unmodified meta-objects are shared among versions. When a class has not been modified in a series of successive versions, the respective class meta-object is shared among those versions. This means, the respective dictionaries point to the same class object. Similarly, methods are shared among versions of classes.

Figure 29 shows that each snapshot contains a dictionary which associates all class meta-objects to their names. Each class meta-object references all its compiled method meta-objects. In the shown example, the programmer added `#bar` method to the class `Person` in version 6. The method addition changes the class `Person`, because its method dictionary now contains an additional entry. For that reason, there exists two different versions of the `Person` class. In contrast, the class `App` can be shared between both version 5 and 6, because it has

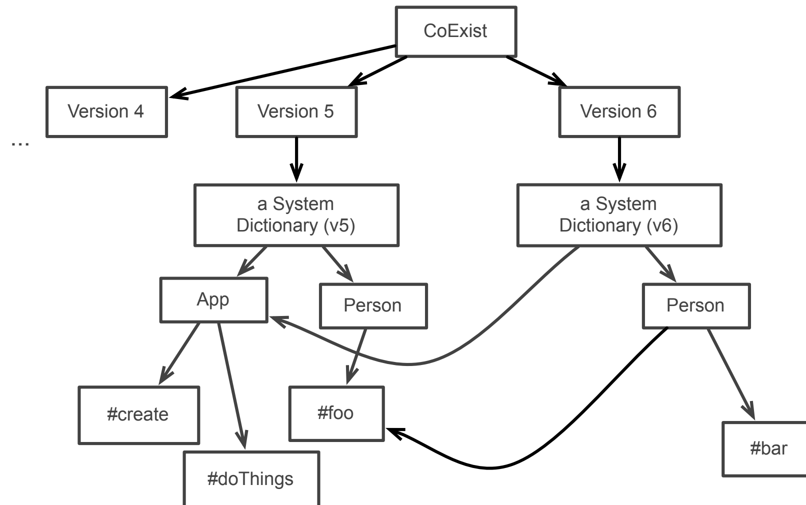


Figure 29: Sharing of meta-objects between versions.

not been changed. Similarly, the method `foo` can be shared among both versions of the `Person` class.

Note: This strategy used for managing versions of meta-objects is related to purely functional data structures (persistent data structures) [55].

6.2.2 *Copy-After-Write*

Meta-objects are copied when programmers make changes to them. When the programmer modifies a class definition, for example, by adding an instance variable, the class is copied behind the scenes, so that now an additional version of the class exists to which changes can be applied while still preserving the previous class version. In addition to the class, all other meta-objects affected by this change are copied as well. For example, modifying a class requires copying the system dictionary, so that one dictionary version includes the new modified version of the class, and the other still includes the previous version of the class.

Relying on Squeak's notification mechanism required a copy-*after*-write rather than a copy-*on*-write mechanism. Squeak's notification mechanism sends out notifications only after modifications have already been done and not before they are done. For that reason, programmers cannot directly work on the data structures of meta-objects under versioning. Instead, the tools present a working copy to the

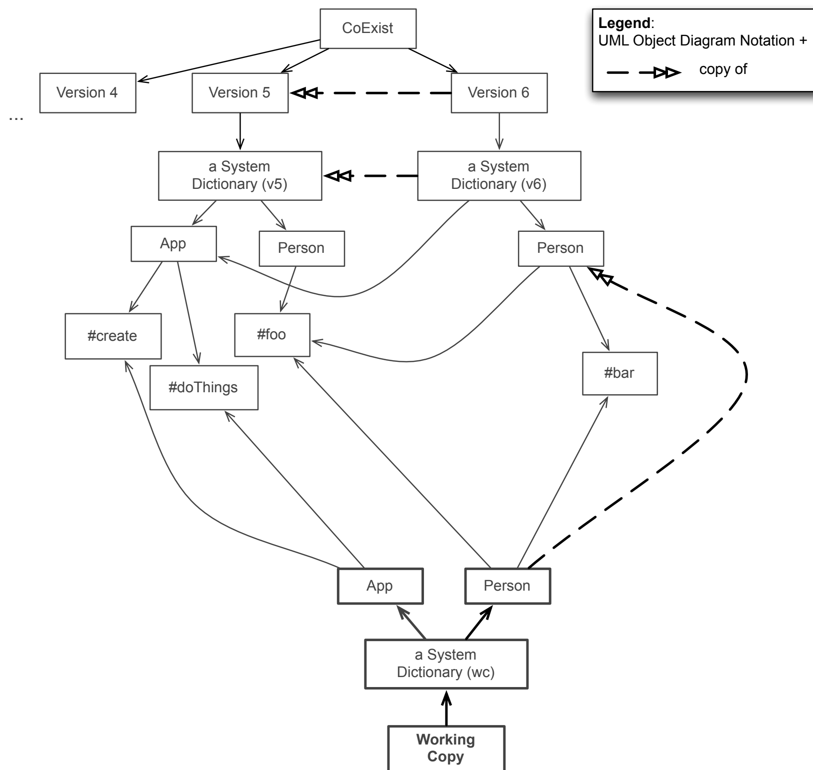


Figure 30: Copying meta-objects from the working copy to the newly created version 6.

programmers. After every modification to the working copy, a new version object is created and all modified meta-objects are copied to this version. This procedure is illustrated in Figure 30.

DISCUSSION Maintaining a working copy of meta-objects can be avoided by changing Squeak's notification mechanism. If change events were sent before the actual change happens, a copy-on-write mechanism could be used.

6.2.3 *Handling References Between Shared Meta-Objects*

The chosen implementation strategy, to copy modified meta-objects and share unmodified meta-objects among versions, requires the consideration of references between meta-objects in the graph of all meta-objects. For example, when a class is modified, the class and the system dictionary are copied. However, the modified class is referenced by other meta-objects that were not directly modified. A class can have subclasses, which in turn have a pointer to their superclass. In addition, a class' superclass references its subclasses for performance reasons. This issue of cross references concerns all meta-objects that are mutable and referenced by other meta-objects, which thus concerns all meta-objects except method objects. Compiled methods are not concerned because they are immutable. (New method objects are created when the code is modified and saved.)

There exist various kinds of references between meta-objects, which have been handled differently:

- The *system organization* associates classes with categories and thus has references to all class objects. CoExist handles these reference by also creating a copy of the system organization whenever a class is copied.
- The *subclass relationship* has been re-implemented. Every class still holds a set of references to their subclasses. But instead of returning this set, it is refreshed on demand by looking up the names in the currently active system dictionary.
- The *superclass* relationship cannot easily be re-implemented in the image, because the superclass reference is used for executing byte codes in the virtual machine. Therefore, whenever a class is copied, CoExist also create copies of all classes of the subclass hierarchy.

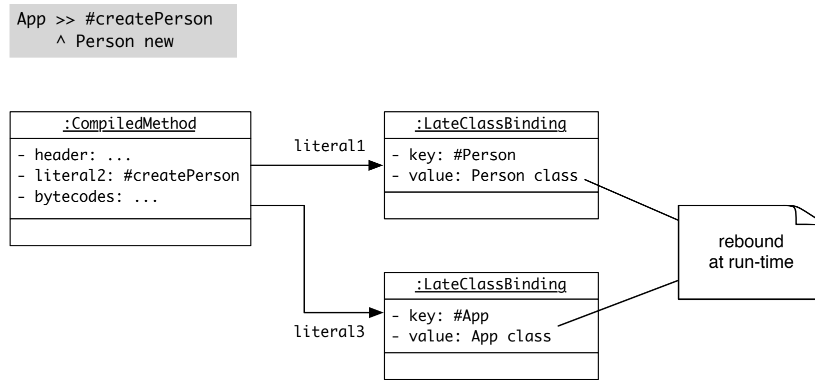


Figure 31: LateClassBinding literals for the method `createPerson`.

- Method objects can also have references to classes. The nature of these references has been changed to promote sharing of meta-objects. The change includes adaptations to the compiler and the virtual machine, which are describe in the next section.

6.3 LATE CLASS BINDING

CoExist uses late class binding to promote sharing of meta-objects among versions. This concept changes how methods reference classes. Traditionally, when a method's source code includes a symbol that refers to a class, the compiled method object will have a literal that points to the respective class object. In addition, every method has a reference to its owner class. These "static" references to class objects reduce the possibilities for sharing meta-objects. When, for example, a method is modified, it needs to be copied as well as its owner class. However, all other methods also would have to be copied so that the pointer from the methods to their new owner class can be adjusted properly. Furthermore, all methods that reference the owner class would have to copied as well, which in turn requires a copy of their owner classes and so on.

So, instead of using regular literals, which directly point to the respective class object, methods now use *late bound class literals*, which mainly consist of the name of the referenced class. These are bound to actual class objects only at run-time, when the virtual machine executes byte code and therefore accesses the late bound class literals. The actual binding of names to classes depends on the currently active version, which is defined by a dynamic variable [90] implemented using process-/thread-local storage.

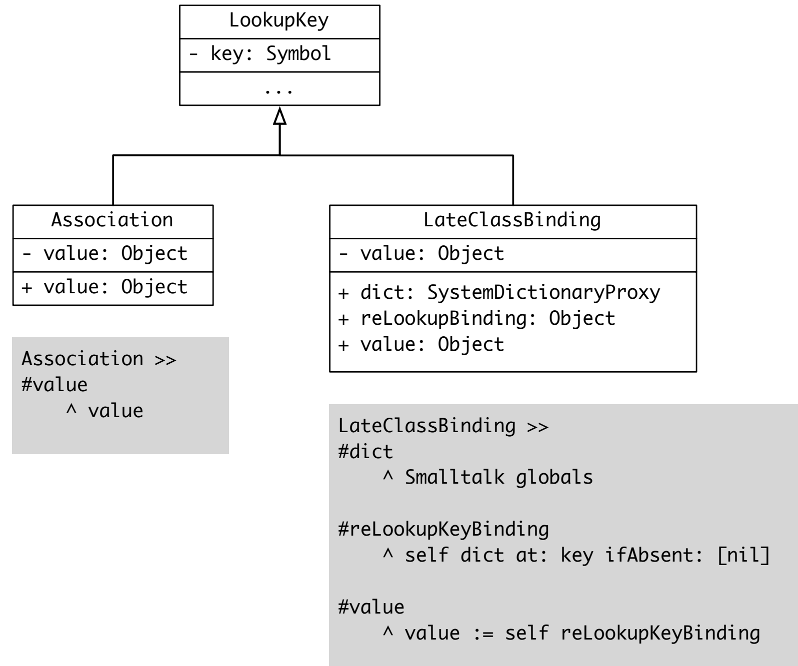


Figure 32: LateClassBinding class in comparison to the Association class.

The introduction of late class binding required modifications to Squeak's compiler. Instead of adding regular literals to the compiled method, instances of the new class *LateClassBinding* are added to the method during compilation (see Figure 31). LateClassBinding objects are used for referring to both other classes and the method's owner class. *LateClassBinding* objects have two attributes, similar to regular literals, which are named *key* and *value*. However, in contrast to regular literal objects, the `value` method of *LateClassBinding* performs a re-binding of the value attribute using the system dictionary of the currently active version (see Figure 31).

To trigger the execution of the re-binding code, the compiler inserts additional byte codes. After every "push literal" byte code, an additional "value send" byte code is inserted (Figure 33), which binds the literal's value field to the appropriate class version, which will be read during the execution of of subsequent bytecodes.

The execution of super sends has been modified to ensure proper rebinding of the literal that points to the method's owner class. This reference is used to lookup the corresponding superclass. To trigger the re-binding, the new byte code "push method class literal" has been introduced (see Figure 34). This byte code is inserted into the compiled method before every "supersend" byte code along with an

example source code

```
App >> createPerson
    ^ Person new
```

generated byte codes

traditional

```
pushLit: Person
send: new
returnTop
```

with late class binding

```
pushLit: Person
send: value
send: new
returnTop
```

Figure 33: LateClassBinding introduces an additional value send after each push literal byte code.

example source code

```
App >> asString
    ^ super asString
```

generated byte codes

traditional

```
self
superSend: asString
returnTop
```

with late class binding

```
self
pushLit: method class
send: value
superSend: asString
returnTop
```

Figure 34: LateClassBinding adds a push method literal and a value send byte code before each super send.

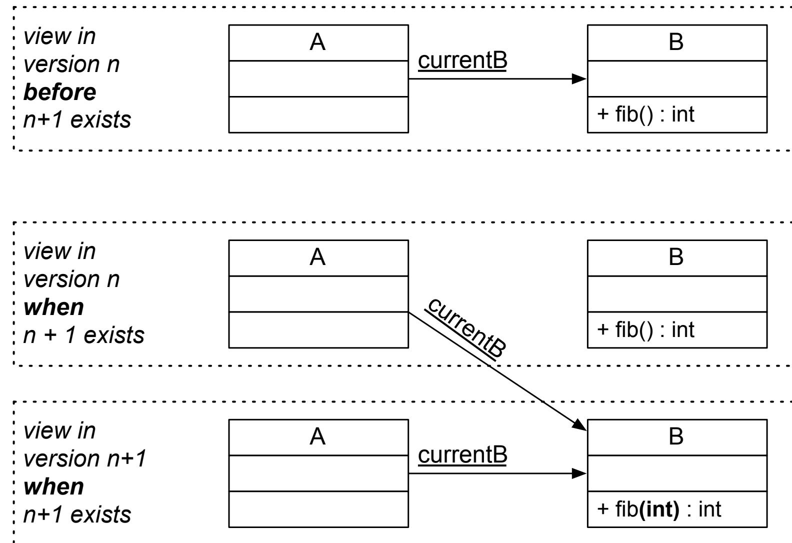


Figure 35: Invalid static reference after class modification.

additional "value send" byte code, to trigger the execution of the value method.

The described changes required corresponding adjustments to the virtual machine that interprets the byte code. The changes were made to the Cog VM [53]. The Cog VM consist of a regular interpreter and multiple generations of just-in-time compilers (JITs). For this work, changes were made to the regular interpreter called StackInterpreter and the first generation JIT called SimpleStackBasedCogit.

6.4 LIMITATIONS

The current prototype is sufficient for many applications, but it still has some limitations:

- Some classes of the image need to be excluded from late class binding, and thus, changes to them cannot be versioned. This includes in particular all classes that are needed for implementing and running the late class binding mechanism and the core of the versioning mechanism.
- Versioning is limited to source code changes and does not include changes to any other kind of object state in the image (except source code). For example, application state is not un-

der version control. Hence, switching between versions only changes the source code, and not the state of running applications. This means that switching between versions requires a restart of the application under development. A possible improvement could involve snapshotting the state of running applications alongside the source code and meta-objects. With this snapshotting in place, going to different version would immediately bring back the application in the state it was when the version was created. This improvement could be implemented using the same mechanism as the one used in worlds [91], a language construct to support scoped-states for each object.

- The current implementation also lacks support for direct references to class objects, for example, when a class variable of class A holds a reference to class B (the meta-object). Figure 35 illustrates this scenario. If a user makes changes to class B, and then goes back to a previous version, the class variable of class A will still point to the most recent version of class B. The class modification will rebind all references from version *n* to version *n+1*. However, this will also affect the view of version *n*, because such state changes between versions cannot be handled properly.

6.5 Performance Evaluation

User interface guidelines suggest a limit of two to four seconds for frequent programmer operations [67, Ch. 10]. Furthermore, the results of our informal user studies suggest that performance characteristics are an important factor for the adoption of the proposed tool support. In the following, we report on several performance evaluations.

Setup: As the system under evaluation, we use the Seaside[1] Web application framework, because it extends many parts of the Squeak environment, which challenges the current implementation strategy for sharing meta-objects.[2] We artificially created 243 versions by loading 5 consecutive releases (from 3.0.0 to 3.0.4). The entire system including Seaside contains 3,312 classes and 68,950 methods. We used the 4.2 release of Squeak/Smalltalk. All measurements were performed on an Apple MacBook Pro 2.93 GHz Intel Core 2 Duo with 4 GB of RAM.

1 http://www.seaside.st

2 For every change in a class meta-object, each subclass' meta-object needs to be copied to create the new version. The higher in the hierarchy a class is, the more subclasses need to be copied.

PERFORMANCE OF CHECKING OUT: Checking out is the action of installing a snapshot into the IDE by copying all meta-objects (as explained in Section 6.2). Checking out requires 1.6 seconds on average, which is below the threshold of 2 seconds.

PERFORMANCE OF LOADING: Loading is the action of updating a working copy to a different version. Updating a working copy can be implemented either by checking out (1.6 seconds) or by applying a set of changes to the current working copy. We use the latter option as an optimization when the target version has few changes with respect to the current one. Using this option, CoExist requires 188 ms to withdraw 30 changes, approximately corresponding to 30-60 minutes of work (according to our experience from the studies). *These results suggest that our implementation of CoExist meets the desired response time.*

MEMORY CONSUMPTION: CoExist consumes memory to maintain snapshots of source code and meta-objects. The 243 created Seaside versions roughly amount to a day of work (at a rate of 30 versions per hour). CoExist requires 68 MB of memory to maintain snapshots of these 243 versions of Seaside, indicating that the size of each version is less than 300 KB on average. *These results suggest that CoExist uses a reasonable amount of memory.*

DEVELOPMENT SLOW-DOWN: In a Squeak environment, applications are executed in the same process as the IDE and thus, IDE performance impacts applications. We measured the overhead that the presence of CoExist in the IDE introduces by timing 10 executions of the 617 Seaside unit-tests: on average, the execution takes 270 ms with CoExist installed and 217 ms without. Using CoExist thus makes executing programs around 1.24 times slower, mostly due to the binding of the class names that must be done at run-time with CoExist. *These results suggests that static class binding (compile/load-time) does not yield a significant performance improvement compared to class binding at run-time. These results also suggests that having CoExist always running does not significantly slowdown the execution of programs.*

This first evaluation of CoExist is promising. The results show that subjects appreciated the tools and even missed them after the experiment. Furthermore, CoExist is fast enough for frequent user operations and only consumes a reasonable amount of memory.

SUMMARY

CoExist is implemented by versioning meta-objects, which preserves immediate access to previous development states. To reduce overhead, meta-objects are shared between versions. However, sharing required modifications to the treatment of references. For example, changes to the compiler and VM were made to enable late binding of classes, so that the appropriate version of a class is resolved at runtime when the late bound reference is accessed. Performance evalutions suggest that, on one hand, versioning meta-objects provides sufficiently fast access to previous versions so that programmers can use it frequently, and, on the other hand, the memory overhead required for that is acceptable.

7 DISCUSSION

This chapter discusses the dissertation topic in a broader context. The first section explains why low recovery costs is particularly useful for working on program design tasks. The subsequent sections illuminate cognitive implications of sufficient recovery support or the lack thereof.

7.1 Why program design is difficult

The provision of dedicated recovery support is particularly beneficial for program design tasks, because design tasks in general are inherently difficult and thus prone to false assumptions, misinterpretations, and errors of other kinds. Researchers have studied the attributes of design tasks and describe, among others, that design involves a co-evolution of problem and solution, requires the consideration of various interrelated dimensions of the problem, and requires subjective interpretation [48, 17]. While these attributes characterize design in general, they do also apply to program design in particular. They generally help to understand the demanding nature of design task.

7.1.1 *Multiple dimensions and their interdependencies.*

Design can be complex because it requires the consideration of various dimensions that frame the situation. These dimensions are connected with each other: Making a change in one dimension has an impact on a fair number of others. This raises conflicts when the effect on other dimensions is undesirable.

Program design also involves many interdependent dimensions. Functional requirements often need to be considered in context of non-functional constraints such as performance, fault-tolerance, security, and overall projects constraints such as time and budget. In addition, programmers consider aspects such as modularity and read-

ability to support subsequent tasks. These dimensions are often in conflict: Source code optimized for performance is often harder to understand, and improving both readability and performance takes time and thus money, which might also be spend on the implementation of additional features; also, improving the readability and conciseness of one aspect of the program might impede the readability and conciseness of another one, and avoiding this drawback might require a larger refactoring.

7.1.2 *Co-evolution of problem and solution.*

Design is difficult due to a high degree of uncertainty—"It seems that creative design is not a matter of first fixing the problem and then searching for a satisfactory solution concept. Creative design seems more to be a matter of developing and refining together both the formulation of a problem and ideas for a solution ... ." [20]. Study results show that designers tend to look for patterns—a partial structure—that helps to better understand the problem and related information. This partial structure can aid the development of a solution idea. Trying out this idea supports understanding its implications and refining the problem. Iterating over problem and solution in this way allows to incrementally improve the understanding of both [20].

Program design is an iterative activity as well. When programmers are unsatisfied with the source code, they explore it and look for alternative structures. They look for patterns that potentially help maintaining the code, for example, the Visitor pattern [**?**]. After refactoring to this pattern, developers might realize that the new code still has problems. But even if they are unsatisfied after a first iteration, they improved their understanding of both the problem and the solution.

7.1.3 *Subjectivity and Measurement.*

Design is complex because it requires subjective judgment. The desired goals are typically described in a vague manner and the efforts to achieve them are mainly limited by time and budget. This vagueness requires designers to interpret the situation and move forward according to inner values and beliefs. Furthermore, the open-ended problem and solution space render metrics such as right or wrong inadequate. Every solution can only be more or less appropriate with respect to certain dimensions. While some dimensions allow

for quantitative measurements, many require qualitative considerations as well as careful interpretation and judgment with respect to importance.

Program design requires subjective judgment as well: They have to decide, for example, whether some more decoupling would be beneficial, and whether additional refactorings are worth the effort; They might also wonder whether it is worth to create an abstraction if used once or twice; Sometimes it might be meaningful to invent a domain-specific language; They have to decide what is simple enough and what name is sufficiently meaningful for a variable.

7.2 BENEFITS OF A REDUCED NEED FOR BEST PRACTICES

CoExist helps programmers deal with various recovery situations. It makes recovery fast and easy to accomplish. Because of that, programmers have reduced need for employing best practices to avoid recovery.

7.2.1 *Cognitive Benefit: Reduced Mental Workload*

Manual problem prevention by constantly performing best practices requires conscious analytic reasoning. It involves reflecting about recent changes, imagining upcoming changes, assessing risks, and making decisions on what actions to be performed next.

This kind of cognitive work has to be conducted in parallel to the main programming work. On one hand, programmers are concerned with advancing the code base by implementing new features and improving the program design. On the other hand, they are concerned with observing their programming activities, and conducting various practices if their assessment of the current situation suggests doing so.

However, research in psychology and neuroscience suggests that, for example, attention and effort should be considered resources that are limited [45, 44, 43, 88]. Among other cognitive capacities, the mental working memory for conscious analytic reasoning is limited. If there is an overload, some tasks will fall short. For example, one study illustrates that people have deficits in perceiving information during mental tasks [46]. Another study suggests that driving a car and conducting a phone call in parallel is difficult (and thus danger-

ous) if the route is particularly challenging (e.g. a winding road) and the phone call also requires concentration and reasoning [79].

Consequently, a constant need for problem prevention activities increases the chance for cognitive overload during programming tasks, in particular during program design tasks. First of all, program design task are particularly challenging as discussed above. Hence, the cognitive load is typically high. But in addition, such tasks inherently have considerable room for errors. Because of the increased risk for mistakes, the ease of recovery is particularly important. Thus, programmers have to be more careful and perform problem prevention to a stronger degree.

Cognitive overload during programming tasks, in part because of the need for manual problem prevention, implies that some tasks will fall short. It is thus possible that programmers ignore best practices while being focused on advancing their code base. Reasoning about the code base and the desired improvement can be so demanding that programmers "forget" to consider testing or committing or else.

Another possibility is that reasoning about problems and their prevention dominates the course of action. If programmers think too hard about conducting best practices and problem prevention, there might be insufficient cognitive capacity left for reasoning effectively about the actual programming task. Also, a programming task might so hard in relation to the cognitive capacity available that any other dominant thought would disrupt performance significantly. Such effects have been shown, for example, in a study on performance pressure [9]. "Results demonstrated that only individuals high in working memory capacity were harmed by performance pressure, and, furthermore, these skill decrements were limited to math problems with the highest demands on working memory capacity. These findings suggest that performance pressure harms individuals most qualified to succeed by consuming the working memory capacity that they rely on for their superior performance." [9]. The findings suggests that "Too much concern about how well one is doing in a task sometimes disrupts performance by loading short-term memory with pointless anxious thoughts." [45, Chapter 3]

Providing sufficient recovery support avoids the need for manual prevention and thus reduces the workload during programming tasks. It allows for a *temporal separation of concerns*. Programmers can focus on advancing their code base, and if need be, they can later focus recovering from problems with only little effort.

7.2.2 *Cognitive Benefit: Reduced Need for Self-Control*

While running tests might also increase the confidence in recent changes, the main purpose of regularly conducting best practices is the avoidance of problems. Thereby, the return on investment is unclear, as discussed previously. Consequently, programmers have little motivation to regularly conduct these activities. They typically have more interest in solving their current programming problems and advancing the code base. To still conduct precautionary activities, even if they are not of current interest, requires self-control (or willpower).

However, studies found that the ability to control one's behavior gets depleted by the use of self-control [4, 5]. Researchers proposed self-control as a limited resource similar to a muscle: exertion leads to decreased performance. An alternative model to poorer self-control at time 2 after exerting self-control earlier at time 1 is that people have "reduced motivation to exert control, reduced attention to cues signaling a need for control, increased motivation to act on impulse, and increased attention to reward-relevant cues" [40].

These finding suggests that programmers' self-control ability decreases over the day, given the described assumption that constantly following best practices requires self-control. This implies that their ability to constantly follow guidelines gets depleted. It further implies that less of this resource remains available for other tasks.

It is thus beneficial for programmers to work with built-in recovery tools that reduce or even avoid the need for following best practices.

7.2.3 *Cognitive Benefit: (Temporal) Separation of Creative and Judgmental Concerns*

Furthermore, built-in recovery support is particularly beneficial during explorative tasks, which involve uncertainty and require creative thinking. Psychology distinguishes two modes of thinking [70, 24]. While creativity along with intuition is attributed to the fast thinking mode, the analytic approach along with suspicion is attributed to the slow thinking mode [24]. The operations of the slow thinking mode are expensive and exhaustive. For this reason, this mode is only used when need be [45]. Studies suggests that whether operations of the slow thinking mode are involved depends mainly on two aspects: *processing fluency* and *affect* [13, 27, 85]. *Processing fluency* is described as the "subjective experience of ease with which people process informa-

tion" [1] or as "cognitive ease" [45]. *Affect* refers to emotional state, the mood, which people are in.

This separation of modes implies that when people are in one mode, people are better in tasks that involve operations associated with that particular mode, and less good in tasks that involve operations associated with the other mode [45].

These findings suggest that explorative programming tasks, which require creativity and intuition, do not go well together with problem prevention activities, which require suspicion and conscious analytic reasoning. When writer constantly judge their ideas and sentence, they will hardly create much ideas and sentences. Similar to such a writer's block, programmers can also experience a coder's block [54], when they judge their ideas too early too critically.

Providing sufficient tool support for recovery scenarios avoids the constant need for problem prevention activities. It allows deferring judgmental activities and thus avoids mix of the different modes of thinking. A temporal separation of creative and judgmental concerns is likely to increase efficiency for both kinds of tasks.

7.3 CODING AS A MEANS OF LEARNING

The presence of dedicated recovery tools such as CoExist allows programmers to immediately realize ideas instead of conducting careful reasoning about it upfront. Programmers can make corresponding changes without hesitation, because they can easily recover if such a need suddenly arises.

This approach of immediately realizing ideas in order to test them is beneficial for complex tasks. Changing source code according to current thoughts is a form of externalizing these thoughts, similar to talking an idea through or writing it down. Such activities have cognitive benefits. They help to better understand the problem and its constraints, as well as to discover new and interesting options for solving it. The value of externalizing thoughts during design work is supported by research on activities such as sketching and prototyping, which are considered crucial in fields such as architecture or product design

7.3.1 *Cognitive Benefits: Support Thinking by Doing*

The externalization of thoughts facilitates inference, understanding, and problem solving [81, 80]. More specifically, the creation of external representations serves, among others, the following purposes:

REDUCED WORKING MEMORY LOAD. An external representation contributes as an external memory for elements of thought. "This frees working memory to perform mental calculations on the elements rather than both keeping elements in mind and operating on them", as explained by Suwa and others [80]. 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. Study results suggest that no more than 3 or 4 chunks can be held in mind. (The complexity of chunks can strongly vary depending on previous knowledge). Given too many chunks at once, a human being experiences cognitive overload, which impedes learning and problem solving [11, 25]. Studies have also shown, that cognitive activity rate will slow down [11]. The mind will work slower.

RE-INTERPRETATION AND UNEXPECTED DISCOVERY. External representations allow for re-interpretation. During the creation of an external representation, abstract concepts and thoughts are associated with specific tokens: the idea is depicted in a particular way, or described using specific words, or implemented by changing and extending the source code in a particular way. However, when designers revisit these external tokens, they can *see* them *as* something else [33]. They associate abstract concepts with these tokens that are different than the original ones. An external representation brings to mind information from long term memory that might otherwise not be retrieved [80]. The particular arrangement of external tokens can also lead to the discovery of unexpected relations and features [64].

7.3.2 *External Representations in other Design Fields*

Designers can rely on mental simulation to a certain degree [15]; in particular, experts are able to do a fair amount of *what-if* reasoning [63]. However, the use of mental simulation can easily become too complex for one's mental abilities [25]. For that reason, designers of various fields rely on sketching and creating physical prototypes

to externalizes theirs ideas [41]. The externalization of thoughts is considered key to efficient and effective outcomes.

Sketching is a method that designers use for exploration in a design situation, for example, in architecture, engineering design, or product design [16, Section 3.5.]. Sketching often reveals unexpected consequences and thus helps to keep the exploration going. Sketching supports pursuing a line of thought, to refine an idea, and also to study an idea's implications [33]. It enables designers to have a "reflective conversation with the [design] situation" [64].

Prototyping supports design activities in ways similar to sketching. A prototype is a manifestation or an embodiment of an idea (independent of the medium) [37]. Prototypes are often created to evaluate certain aspects of an idea such as look-and-feel, feasibility (implementation), or usage purposes. But despite evaluation, prototypes also have a generative role: They help designers explore a design space [49]. They are sources for inspiration and aid reflection on design activities. Several studies emphasize the value of prototyping for design tasks. For example, study results suggest that, under given time constraints, the creation of multiple prototypes for one design problem will yield more valuable insights and lead to better results [22, 34], in particular for novices. The result of another study shows that prototyping reduces fixation on a certain idea [93] and encourages developing and engaging other ideas.

7.3.3 *Source Code as External Representation*

Programmers can benefit from exploring their ideas by realizing them. Programmers should be encouraged to change the source code according to current thoughts, observe the result and study its implication, better understand the constraints but also get inspired, change the source code again, and so on. This approach will support programmers in reasoning about program design. Programmers will associate many thoughts with different pieces of source code, such as idioms, patterns, or "bad smells" [29]. They have also developed a feeling for simplicity and beauty concerning source code, and they "know" what kinds of code they like and what they want to avoid. Seeing an idea realized in source code reveals unanticipated implications and facilitates the discovery of unintended features.

SUMMARY

Program design tasks are particularly difficult. Programmers thus notably benefit from recovery support during such tasks when errors are likely. But even more, the level of available recovery support has considerable cognitive implications. Low recovery costs reduce the need for problem prevention, which in turn reduces the mental workload and the need for self-control. It also allows for a separation of creative and analytic concerns during the course of programming. In addition, when the costs for recovery are sufficiently low, programmers are encouraged to try out their ideas as they come to mind and can use coding as a means to learn and reflect about their ideas.

8 LAB STUDY

The cognitive implications described in the previous chapter gave reason to hypothesize that recovery support such as CoExist influences programming behavior. Because of that, a controlled lab study was conducted to examine whether CoExist affects programming performance.

8.1 METHOD

Using a repeated measurement setup, 22 participants were requested to improve the design of two games on two consecutive days.

8.1.1 *Study Design*

Figure 36 illustrates the experimental setup. Participants were 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.

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 makes it possible to test for an interaction effect of the two fac-

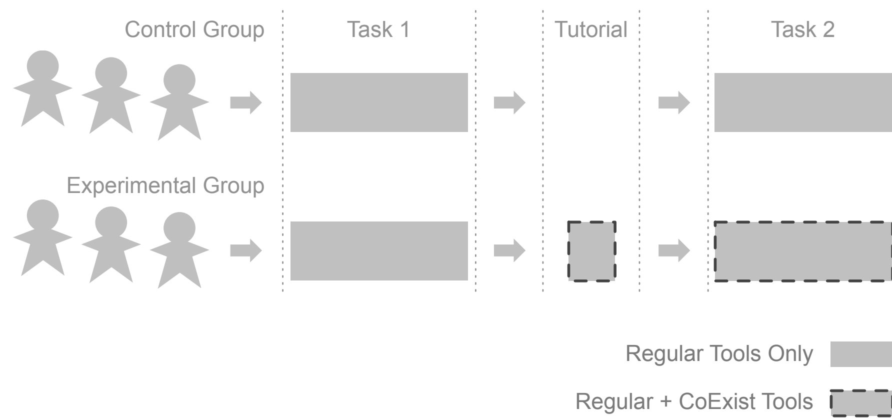


Figure 36: Our experiment setup to compare performance in program design activities.

tors, which is the indicator of whether CoExist affects programming performance.

8.1.2 *Materials and Task*

On both days the task was improving the source code of relatively small computer games. More specifically, participants were requested to study the source code, detect design flaws in general and issues of unnecessary complexity in particular, and 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 with 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 thus help imaginary team members better understand them. (Most participants followed this instruction by regularly writing commit messages).

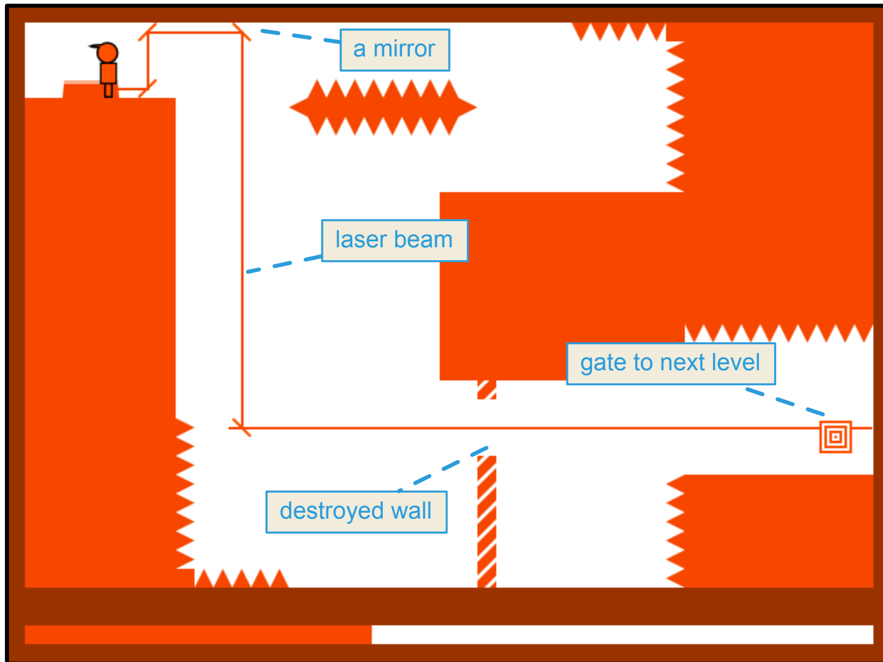


Figure 37: Screenshot of the LaserGame used for task 1.

Figure 38: Screenshot of the MarbleMania game used for task 2.

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

Figure 39: Size indicators for the games used in the study

On day 1, participants worked on a game called *LaserGame*, and on day 2 they worked on a game called *MarbleMania*. Screenshots of both games are shown in Figure 39. 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 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.

While the numbers shown in Figure 39 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.

8.1.3 *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 the 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 (8.2).

Prospective participants needed to have experience in using Squeak/Smalltalk and must have passed their fourth semester. By this time students will have typically attended two 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 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.

8.1.4 *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 2, 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 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 to run the CoExist tutorial session before lunch. (We had to make an exception for three participants, who only had time during the morning or evening hours. Because 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).

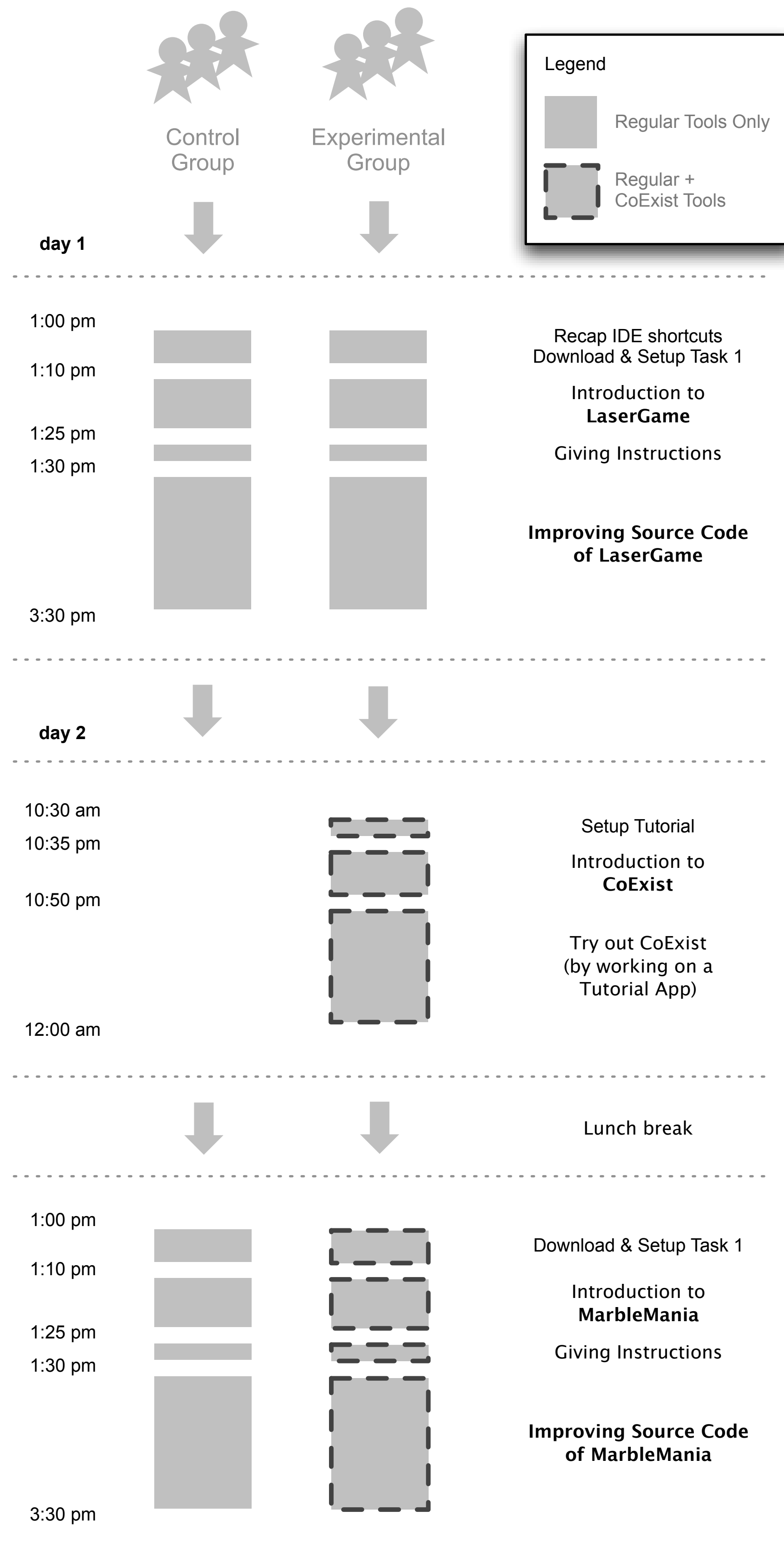


Figure 40: The experimental procedure for both the control and the experimental group.

Figure 40 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.

8.1.5 *Dependent Measure*

We measured 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 measure is a proxy for how much actual work was done within the two hours, excluding time spent on activities such as staring into the air or browsing the code base.

8.1.5.1 *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 consists 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 recomputes 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:

- Replace dictionary that holds information about exchanged 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 not 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 Figure 41.

First, we extracted the time stamps of all versions and listed them in a column of a spreadsheet. We then grouped these time stamps according to the commits that subjects made during the task, and put the corresponding commit messages in a second column (illustrated in Figure 41). 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 Figure 10) to refine our understanding, 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 belong 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.

8.1.5.2 *Quantifying the Effort for Identified/Code Increments/Improvements*

We measured 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. Figure 42 shows two lists of IDE interactions, written down in an executable form (regular Smalltalk

Diff for individual changes / versions	Fine-grained version data		Commit messages	Identified increments
Modified in SWA18LaserBeam #calculateDownWay + self calculateWay: #down deltaX: 0 deltaY: 1. - ~~| xTile yTile |~~ - ~~xTile := self points last x // SWA18Tile size.~~ - ~~yTile := self points last y // SWA18Tile size + 1.~~ - ~~[(yTile <= self swaWorld tilesY) and:~~ ~~[(self swaWorld tiles at: xTile @ yTile) laserCanEnter]]~~ - ~~whileTrue: [yTile := yTile + 1].~~ - ~~self stopGoingTo: #down at: xTile @ yTile~~	14:01:41	**Added** Method	"... extracted code that is similar in all these calculate methods; improved the previously extracted, generic #calculateWay: ..."	**LG_ExtractGeneric-CalculateMethod**
	14:02:16	...		
	14:02:32	**Modified** Method		
	14:02:49	...		
	14:03:01	...		
	14:02:16			
	. . .			
Removed in SWA18LaserBeam ~~#pointAtRightOfLaser: aLaser~~ - ~~| laserX laserY laserWidth |~~ - ~~laserX := aSWA18Laser coordinates x.~~ - ~~laserY := aSWA18Laser coordinates y.~~ ...	14:13:06	**Modified** Method	"... deleted useless condition, integrated code from called methods, and removed the other methods. .... Simplified method based on detected 'invariant' ... "	**RemoveStatements + 2 * InlineMethod**
	14:14:18	...		
	14:14:25	...		
	14:15:16	...		
	14:15:28	**Removed** Method		
	. . .			

Figure 41: Excerpt of a spreadsheet with coded version data.

```
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
```

Figure 42: 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.

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.

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 extracting a method without any parameter. 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`, `#brieflyCheckCode`, 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.)

8.2 results and discussion

Figure 43 shows the result scores for each participant and task, the accumulated points for the identified increment. The plots in Figures 44 and 45 illustrate the averaged score of each group for each task.

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 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.

The results show a difference in the performance between the control and the experimental group for the MarbleMania tasks. Furthermore, the control group performed on average less well for the second task, which indicates that improving the design of MarbleMania was the more difficult tasks. In contrast to this increase for the control group, the experimental group scored on average higher for the second tasks compared to their results of the first task. The fact that the experimental group improves while the control group degrades is an indicator that the provision of CoExist helped to compensate for the additional difficulty.

A 2 x 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-

	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

Figure 43: Final scores for participants by task

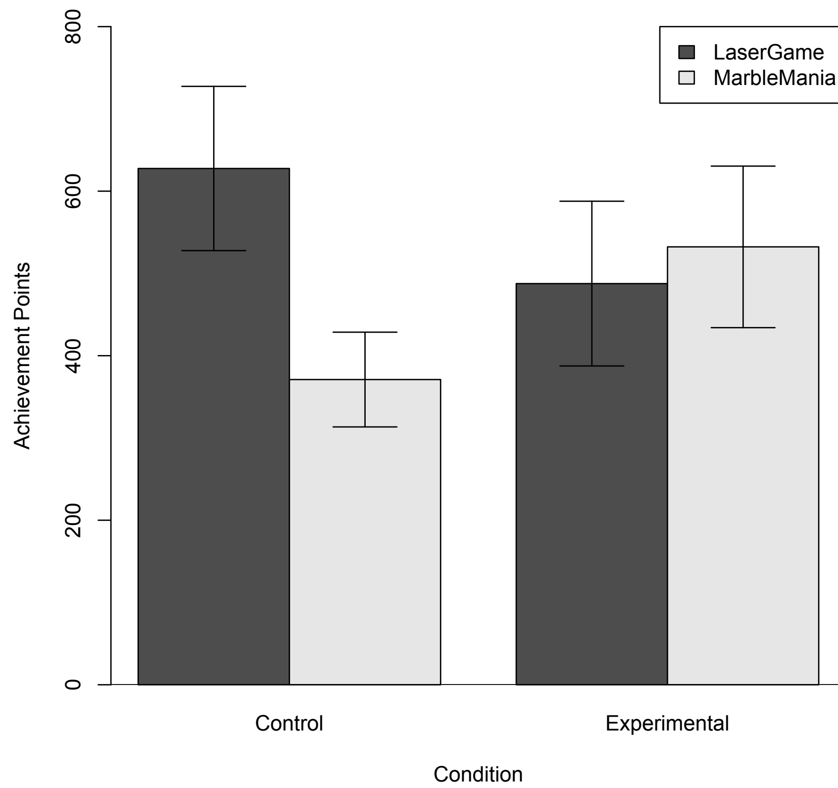


Figure 44: A bar plot of the study results. Error bars represent the standard error of the mean.

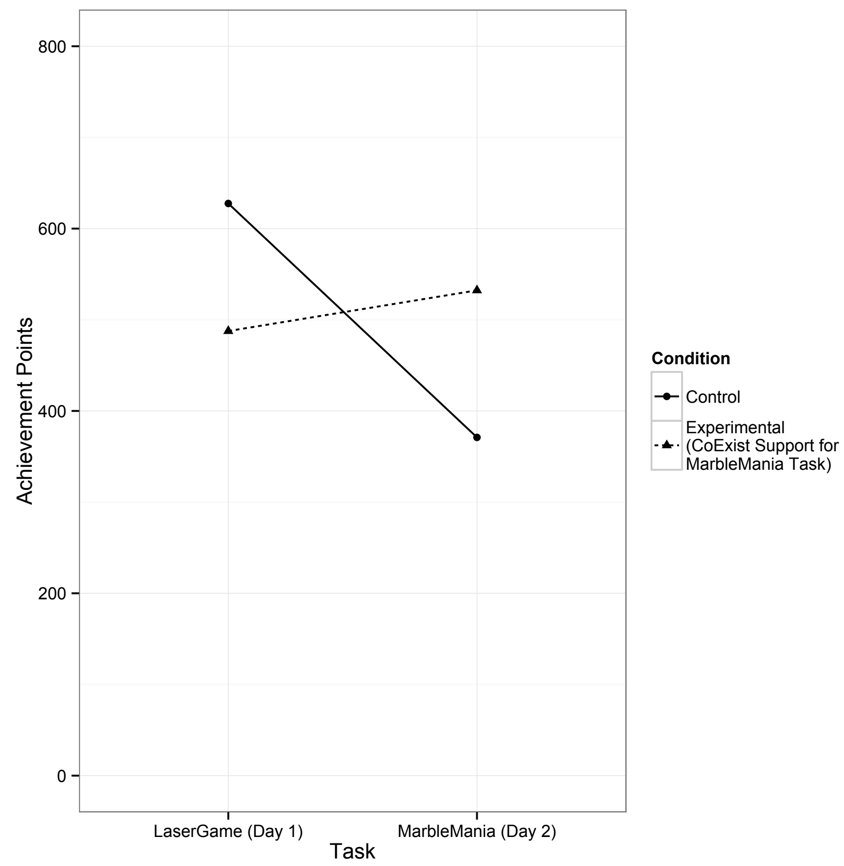


Figure 45: An interaction plot of the study results.

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.

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, ${\eta_p}^2$.

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 $r(20) = .1$, $p = .66$. Furthermore, 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.

8.3 Study Design—Justification and Limitations

8.3.1 *Measuring Achievements for a Fixed Time Span*

For our experiment, we decided to keep the time 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 well-defined goal and measure the time needed to achieve this goal [42].

We decide for a fixed time setup, because CoExist was designed to help programmers in explorative tasks. Such tasks inherently in-

volve ambiguity and uncertainty, contradicting the requirements for a setup that measures time to completion. A time to completion setup requires a clearly defined task without any uncertainty or ambiguity. There has to be a clear indicator when the task is finished. Also, the task description should provoke similar thoughts, so that all subjects have a similar idea how to achieve the goal. These criteria can hardly be met in an experiment where participants accomplish a design task.

Research in the general field of design supports that a fixed time setup is a meaningful choice for measuring performance in design tasks. In [22, 21], for example, the authors report on the empirical examination of prototyping techniques in product design and ad-design. In these experiments, performance has been compared by evaluating various quality criteria of the design outcome, and participants 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.

8.3.2 *Using a Repeated Measurement Setup*

We decided to rely on a repeated measurement study design because programmers strongly vary in approaching such tasks. Programmers have a difference in working speed which involves code comprehension, code writing (typing speed), but also tool usage. Moreover, different programming personalities will identify and work on different kinds of issues in the source code. While some programmers will have the tendency to focus on the various smaller independent issues, others will have the tendency to find out major flaws in the overall program structure of the code and to improve on that while ignoring smaller issues. While the various kinds of contributions can be similarly important for the long term success of a software project, the differences will lead to strong variations in the response variable, in relation to the possible difference caused by the provoked variations. In such circumstances, literature recommends repeated measurement setups to ensure the ability to discover statistical effects, in particular when having access to a limited number of participant candidates [42].

8.3.3 *Measuring Interactions for Independent Increments*

To approximate the amount of work achieved within the given time frames, we decided to identify the number independent increments

and determining the IDE interactions required for reproduction. We decided for this unusual approach because alternative measures have severe limitations, and we did not want to fall back on measuring time due to the above mentioned reasons.

We could have used measures such as the number of created versions, or number of changed lines of code or similar kinds of metrics. However, such constructs are of limited value as discussed below.

8.3.3.1 *Limitations of Counting Created Versions*

- Changes that lead to new versions strongly vary in the amount of changed code. While adding leading white space is a small change to the code, removing statements or parameters from a method is a much larger change.
- Participants might want to withdraw changes, and in particular when they could not rely on CoExist, they need to manually withdraw made changes, which will likely create additional changes.
- It might also be the case, that a series of changes were good for inspiration and helped the programmer to develop a better idea 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.

8.3.3.2 *Limitations of Counting Changed LOC*

Changed lines of code (LOC) or similar constructs typically have the issue that they do not necessarily correlate with the amount of work. Consider for example the following increments: moving all code from the instance side of a class to the class side (analog to static in Java like languages), or vice versa, including methods and instance variables; removing an unnecessary superclass by moving code into its only subclass. Such refactorings can easily result in a large number of changed LOC, which do not correlate with amount of required work compared to other refactorings such as ExtractMethod.

8.4 THREATS TO VALIDITY

The presented study has a number of threats to validity, which are discussed below.

8.4.1 *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.

8.4.2 *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 limited 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 [66].

8.4.3 *Reliability*

I 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.

I 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.

8.4.4 *Internal Validity*

While the results show a correlation between the treatment and the outcome, there might be factors other than the treatment causing or contributing to this effect. While the use of a repeated measurement setup rules out single group threats, there is need to consider multiple group threats and social threats.

To the best of my 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.

The selection-regression threat has to be considered as well, 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.

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

8.4.5 *External Validity*

Because participants were students, 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.

The study was artificial in the sense that programmers may rarely have a fixed time span of two hours they can spend on improving the source code. 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 code base 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.

SUMMARY

A study was conducted to empirically examine the effect of CoExist on programming performance. Participants run 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. Change histories were analyzed to identify increments and determined the required effort for reproducing them. An ANOVA test shows a significant interaction effect, $F(1, 20) = 5.49$, $p = .03$, $\eta_p{}^2 = .22$, which suggests that built-in recovery support such as CoExist increases programming performance in explorative tasks.

9 RELATED WORK

This chapter discusses the proposed concepts with respect to related approaches and highlights similarities and differences.

9.1 VERSIONING

The undo/redo feature of text editors is very convenient for changing recently entered text. However, undo/redo works on the level of characters, which makes going back to a less recent version of a file rather tedious. Mac OS X starting with release 10.7 provides the feature to regularly save files without explicit request. It offers visual feedback to support finding a previous version.[1] This auto-save feature also allows for juxtaposing the current version with previous ones. Nevertheless, such undo/redo concepts handle files independently of each other, while programming typically involves the manipulation of multiple artifacts. Thus, withdrawing changes requires the developer to manually apply the undo feature an unknown and different amount of time for every modified file.

Version Control Systems (VCSs), which are sometimes referred to or part of configuration management systems, manage the files that belong to a project and support going back to previous snapshots of the project. Developers can employ a VCS to support withdrawing changes. Either developers snapshot manually from time to time, or they let tools automatically snapshot at regular intervals, for example, after each save operation. However, both approaches exhibit limitations. Using the former approach, developers are required to foresee the future, which remains hard. Developers have to continuously assess the likelihood that a future situation might require discarding changes and going back to the current state. Unfortunately, this is hard to assess and there will be situations where a developer forgot to snapshot at the most appropriate time. Furthermore, this approach makes it hard to focus on the design task, because a control loop keeps reminding the developer to consider snapshotting now. To overcome these problems, another approach is to snapshot at reg-

1 http://www.apple.com/macosx/whats-new/auto-save.html

ular intervals.[2] However, if performed in an unstructured way, this approach easily results in a huge amount data that is hard to browse, because it lacks meaningful meta-information. In contrast, CoExist captures meta-information such as the kind of change that happened. This information guides programmers in finding the snapshot they are looking for, even in the presence of many of them.

Orwell [84] is an early VCS / CM that provides both source and object sharing for Smalltalk systems. Compared to other VCS that employ files as the unit of versioning, Orwell manages versions of classes as well as editions of methods and classes among other. Orwell requires class ownership, but provides an owner with a complete development history. The programmer can also work on multiple versions (or releases) of one class. Nevertheless, according to our understanding, Orwell's versioning scheme is scoped to classes. This means the user cannot easily withdraw changes that span multiple classes and application parts. The Envy system [57], which is based on Orwell, provides *baselining* as an additional concept. This enables programmers to create named snapshots of all artifacts involved in a project. In contrast, CoExist continuously creates snapshots of the entire system, so the possibilities for exploring without hesitation are not restricted to scopes such as classes.

Changeboxes [18] is an approach to capture the history of a system and permit the existence of simultaneous versions in a single virtual machine and development environment. Because a Changebox encapsulates a particular change in the history of a system it is possible to use Changeboxes to go back to a particular state of this system. Nevertheless, since Changeboxes has not been designed for recovery scenarios, going back to previous development states is tedious: it requires finding a Changebox in a tree with no meta-information, creating a branch out of it, giving it a name and a color, closing all code browsers, and opening new ones. Our approach makes going back simpler by automating most of the process and presenting all versions of a system with visual meta-information. Moreover, the Changebox prototype implementation shows an important slowdown when used on a project with close to 2,000 methods: in this setup, the system becomes around 4.9 times slower on average. Contrary to the Changebox prototype, CoExist versions the entire workspace (around 69,000 methods). Still, our approach exhibits only insignificant runtime overhead (a slowdown of only 1.2 on average).

2 http://stackoverflow.com/questions/688546/continuous-version-control

9.2 CHANGE RECORDING FOR EVOLUTION ANALYSIS

The concept of preserving change information between explicit commits has been first introduced with SpyWare [60]. The authors recognized that relying only on traditional, file-based VCS limits the possibilities of software evolution analysis. They proposed recording fine-grained change information in IDEs (such as method modified, instance variable added), as well as semantic borders drawn by tool supported refactorings. Such dense change information, compared to plain diffs, additionally supports reverse engineering activities. It helps programmers better understand the current state of a system, or why it has been designed or changed in a certain way. In particular, it supports reasoning about development sessions [61] of colleagues and to reconstruct what they have done and how. The implemented prototype called SpyWare enables programmers to browse the change history ordered by time, and to sort and re-arrange change information. It features support for interactive visualizations to track changes and to reason about statistics. In addition to browsing change history, SpyWare also allows for generating the source code of intermediate development states (by applying or reverting change operations).

The SpyWare approach has been continued, resulting, for example, in tools such as Replay [35] and ChEOPS [23]. Using Replay, programmers can replay change operations, which have been previously recorded in another development session, to better understand the evolution of the code base. A controlled experiment shows a statistically significant improvement in time required to complete software evolution analysis tasks. ChEOPS is an IDE prototype implemented to fulfill the needs of Change-Oriented Software Engineering, which is itself an extension of the SpyWare approach. ChEOPS extends SpyWare by managing changes in different levels of granularity or supporting the declaration of intents.

There is an overlap in the needs for software evolution analysis and explore-first programming: both require a detailed record of change information. But compared to our approach, the above mentioned line of work on software evolution analysis has a stronger focus on the recording of changes, their management, and the visual preparation. For example, the corresponding tools allow for tracking changes on a statement level, but also for combining fine-grained changes into composite ones, and for declaring the intent of change operations [23]. These and other aspects seem to be meaningful complements to our work. However, instead of recording change operations, CoExist preserves the artifacts of intermediate development states including both source code and run-time artifacts, thereby avoiding the need for re-generation. Furthermore, in addition to preserving intermediate de-

velopment states, which is similar to the above mentioned work, CoExist also provides features such as implicit branching, runnings tests in background, or browsing, running, and debugging previous versions in additional inner environments next to the current one. Such dedicated tool support is essential for explore-first programming.

Another approach that is close to our work is VPraxis, a language which models the history of source code through commands like "create class Person", and "add field 'name"'.[3] Finding a version of interest can then be done via queries such as "which commit last changed this method?". VPraxis can be attached to a development environment to monitor code changes while programmers are doing them. Except for the unit test results and the refactoring-level grouping, VPraxis could propose tables such as the one in Figure 14. Nevertheless, VPraxis lacks support dedicated to needs such as withdrawing changes to start over, juxtaposing program versions, or back-in-time impact analysis. Moreover, VPraxis is not integrated in a development environment in such a way that multiple versions can be browsed and edited independently.

9.3 JUXTAPOSING VERSIONS

Orion is an interactive prototyping tool that allows programmers to compare the impact of multiple changes on a software system [47]. Orion uses models as an abstraction over source code and model transformations as an abstraction over changes. Orion has been implemented to support the manipulation of very large models with more than 600,000 entities, such as classes and methods. As a result of using abstractions instead of source code, Orion cannot be used to execute a previous version of a software system or even study its source code. Moreover, even if Orion's implementation shares some similarities with ours, its goal is inherently different: Orion has been designed for re-engineering changes (such as the removal of dependencies) whereas CoExist has been designed for all kinds of source code changes. As a result, CoExist is fully integrated in the development environment and versions are created automatically whereas, with Orion, changes have to be explicitly expressed as a model transformation.

Juxtapose [34] is a tool that facilitates the creation and comparison of alternatives through a dedicated code editor. Juxtapose also allows the execution of multiple alternatives in parallel. Nevertheless, Juxtapose is not associated with the history of a software system and thus

3 http://harmony.googlecode.com

cannot propose going back or re-assembling changes into incremental improvements.

9.4 FINE-GRAINED BACK-IN-TIME IMPACT ANALYSIS

There are various approaches that use tests to support fault localization. For example, the continuous testing approach proposes to run tests automatically after each change [62]. Because this approach executes tests on the current development state, the execution of a test suite is aborted after each change. Nevertheless, it immediately presents the results to the programmer so that he can fix problems as they appear. CoExist improves on that by recording test results and linking them to the corresponding changes, which allows for analyzing test results only when it is convenient and desired. In addition, CoExist also supports running tests on previous versions.

The Git VCS provides the "git-bisect" command, which uses binary search on a sequence of commits to find the first commit that introduced a bug. CoExist advances this concept by integrating it into the working environment and, more importantly, by preserving access to all intermediate versions between two explicit commits. Furthermore, CoExist can also run unit tests in the background during development, and it provides support for running newly defined tests on all previous versions.

A continuous integration server can recognize commits to a VCS. For every new commit, it builds the code under development and executes tests afterwards [30]. By providing a history of test results for every commit, these servers support back-in-time impact analysis. Unfortunately, such tools can provide fine-grained analysis only if the commits are themselves fine-grained, which requires time and discipline: When commits contain many changes, the programmers have to study all these changes to locate faults.

9.5 RE-ASSEMBLING CHANGES

Best practices include making small commits (sometimes called *atomic* or *logical*) so that every commit only affects one aspect of the system. Followings this recommendation facilitates activities such as going back, fault localization, and reviewing [28]. Using the interactive mode of the "git-add" command can help programmers extracting small commits out of many changes. This command enables pro-

grammers to define the boundaries of a commit by selecting a subset of the modified lines in each modified file. CoExist improves on this command by proposing change-level selection instead of line-level selection, ordering the changes by time stamp, and providing means to run anlyses for the defined increments.

10 CONCLUSION

When unexpected errors require tedious and time-consuming effort to recover, programmers tend to avoid them. They either try to avoid making errors in general or try to be prepared for them. Best practices include running tests often and frequently, making small steps, and performing only one task at a time.

However, when the costs for unexpected recovery are low, programmers have no reason to worry about making errors. With tools such as CoExist, recovery tasks become fast and easy to accomplish. Having such tool support reduces the need for best practices to prevent tedious recovery work.

10.1 CONTRIBUTIONS

Following this line of thought, the main argument of this dissertation is:

THESIS STATEMENT Programmers benefit from dedicated recovery tool support that automatically keeps recovery tasks fast and easy to accomplish by preserving immediate access to static and dynamic information of intermediate development states.

In explaining and supporting the above argument, this work makes the following contributions:

- It shows that effort required for preventing unexpected recovery are either overly time-consuming or error-prone. This work makes the trade-offs between costs and benefits explicit.
- It shows that the costs for unexpected recovery can be kept low by providing dedicated tool support.
- It provides a reference implementation for the proposed concepts. The described implementation shows how immediate and full access to previous state can be achieved by versioning

meta-objects and that the performance overhead can be limited by sharing meta-objects between versions.

- It presents a lab study that examines the effect of additional recovery tools on programming behavior. The results indicate a positive impact. Further research is required to support and validate these results. The presented study is a first step of measuring performance differences in ill-defined design tasks.

10.2 SUMMARY

This work focuses on program design tasks. Program design is concerned with names, abstractions, and the decomposition into modules, among other. Such aspects affect current and future programming tasks. They can, for example, ease or impede program comprehension and maintenance tasks. While program design activities are important, they involve the risk of making errors: an idea can turn out inappropriate, the programmer's understanding of how the code works can be wrong, or bugs can creep into the code without notice. Recovering from such situations is often tedious and time-consuming. Therefore, literature recommends anticipating errors and preventing unexpected recovery.

However, following best practices can fail to avoid unexpected recovery needs. An example case shows how a programmer experiences a sudden need to withdraw recent changes, although he followed best practices. A discussion of this example shows that it is hard to avoid unexpected recovery needs by following best practices. Conducting the required precautionary activities is time-consuming. Because of that, the application of best practices requires trade-offs to be practical. But making trade-offs also implies the risk of unexpected problems.

These trade-offs can be avoided by providing tool support dedicated to various recovery needs. CoExist is a set of tools that has been developed in the course of this work. Implicitly gained meta-information avoids the need for explicit commit messages and assists users in identifying previous versions of interest. CoExist allows for simultaneously opening working environments for previous versions. CoExist runs analysis computations such as tests cases for every newly created version. Users can also use the versioning facilities for back-in-time analyses.

CoExist reduces the need to prevent recovery because it makes recovery fast and easy to accomplish problems. The presented tools would have helped the student in the reported case example to withdraw recent changes and start over from a previous development state. Easy and fast recovery is made possible by continuously versioning the source code based on its structure. The automatically recorded meta-information about each change helps users to rapidly identify a previous versions of interest. With that, recovery becomes inexpensive and programmers have reduced need to rely on best practices.

CoExist is implemented by versioning meta-objects. With that, it provides full and immediate access to previous development states. To reduce overhead, meta-objects are shared among versions. Sharing classes and compiled methods is enabled by making classes late bound, which required changes to the compiler and VM. A performance evalution indicates that access to previous versions is sufficiently fast for frequent use and that the required memory overhead is acceptable.

Recovery support such as CoExist has considerable effects on cognition during explorative programming tasks. The reduced need for problem prevention decreases the mental workload and the need for self-control. Low recovery costs further enable programmers to separate creative from analytical concerns. Even more, it encourages programmers to make changes as a means to learn and reflect about ideas.

In a controlled lab study, 22 participants improved the design of two different applications. Using a repeated measurement setup, the study examined the effect of CoExist as additional tool support on programming performance. The results of analyzing 88 hours of programming suggest that built-in recovery support such as provided with CoExist increases programming performance in explorative programming tasks.

CoExist fills a gap between conventional undo/redo of text editors and VCS. Like undo/redo, no explicit control is required; it runs continuously in the background. Like VCS, the scope of the history is not restricted to invididual files; CoExist versions the entire development state, even independent of project boundaries. Furthermore, while the developed concepts and tools are related to previous concepts and tools, they are notably different in their support for recovery needs.

10.3 FUTURE WORK

CoExist is a first step in providing a rich set of tools supporting an explore-first programming approach. The following paragraphs outline directions how the presented work can be advanced.

10.3.1 *Continuous Versioning*

In a "live system" such as a typical Smalltalk environment, applications keep running while their implementation is changed. The Smalltalk environment makes this possible by automatically migrating the application state after structural changes to source code. A possible improvement could involve snapshotting the state of the running application alongside its source code and meta-objects. With this snapshotting in place, going to different version would immediately bring back the application to the state it was when the version was created.

10.3.2 *Juxtaposing Versions*

Juxtaposing multiple versions currently consists in getting a main window presenting the current version and one sub-window for each different version to juxtapose. This setup works fine when the programmer wants to browse the source code of multiple versions at once. Nevertheless, this setup could be improved in cases where a programmer wants to see the changes for a particular program element (such as a class or a method). For these cases, future work could provide programming tools dedicated to presenting multiple versions of one program element. For example, a code browser could present synchronized views of the same element in different versions.

Beyond comparing static information, tools could provide comparisons of dynamic data such as object states and call trees. These data could be gathered during program and unit-test execution and then presented to the programmer [58]. These tools would help the programmer understand the impact of a set of changes on the execution. In the spirit of what Bret Victor proposes with dynamic pictures [89], this idea could be expanded with comparison of application output. For example, in the context of a game, tools could show the differences between two versions of an animation.

10.3.3 *Fine-grained Back-in-Time Impact Analysis*

Beyond unit test results, CoExist's infrastructure for running computations for each version could be extended for use cases beyond unit tests.

PERFORMANCE FEEDBACK. While run-time performance can be important, programmers typically ignore performance during an initial implementation and consider performance only when it becomes critical. Nevertheless, when the application is already optimized for speed, programmers must pay attention that new changes do not degrade performance. As with bug squashing, automated continuous performance analysis could help a programmer find the change that slowed-down the application.

QUALITY FEEDBACK. Quality-management platforms, such as Sonar,[1] help developers manage code quality by combining metrics and visualizations. These platforms are automatically triggered upon commits. Our approach could bring these platforms to a next level by providing change-level instead of commit-level feedback. The metrics and visualizations could then be integrated within the development environment to provide on-demand feedback about the quality improvement (or decrease) of the not-yet-committed changes.

10.3.4 *Re-assembling Changes into Incremental Improvements*

CoExist enables programmers to select and re-apply individual changes to any version. This feature allows for extracting incremental improvements that are spread over a set of many changes. Nevertheless, CoExist does not detect dependencies between changes which makes the feature less than ideal to use. For example, CoExist could automatically apply the class-creation change when the programmer wants to apply a method-creation change to a version that does not contain the corresponding class. This kind of support can further be improved by detecting behavioral dependencies. For example, a method change often depends on other method changes to make the application compile and behave properly. Besides static analysis, CoExist could leverage the results of the continuously ran unit tests to detect such dependencies.

1 http://www.sonarsource.org/

BIBLIOGRAPHY

[1] Adam L. Alter and Daniel M. Oppenheimer. Uniting the tribes of fluency to form a metacognitive nation. *Personality and Social Psychology Review*, 13(3):219–235, 2009.

[2] F Gregory Ashby, Alice M Isen, et al. A neuropsychological theory of positive affect and its influence on cognition. *Psychological review*, 106(3):529, 1999.

[3] Carliss Y. Baldwin and Kim B. Clark. *Design rules: The power of modularity*, volume 1. The MIT Press, 2000.

[4] Roy F. Baumeister, Ellen Bratslavsky, Mark Muraven, and Dianne M. Tice. Ego depletion: Is the active self a limited resource? *Journal of personality and social psychology*, 74(5):1252, 1998.

[5] Roy F. Baumeister, Kathleen D. Vohs, and Dianne M. Tice. The strength model of self-control. *Current directions in psychological science*, 16(6):351–355, 2007.

[6] Kent Beck. *Smalltalk Best Practice Patterns. Volume 1: Coding*. Prentice Hall, Englewood Cliffs, NJ, 1997.

[7] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[8] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, 2004. ISBN 978-0321278654.

[9] Sian L. Beilock and Thomas H. Carr. When high-powered people fail working memory and "choking under pressure" in math. *Psychological Science*, 16(2):101–105, 2005.

[10] Lenoid Berov, Johannes Henning, Toni Mattis, Patrick Rein, Robin Schreiberand Eric Seckler, Bastian Steinert, and Robert Hirschfeld. Vereinfachung der entwicklung von geschäftsanwendungen durch konsolidierung von programmierkonzepten und -technologien. Technical report, Hasso-Plattner-Institute, 2013.

[11] Zafer Bilda and John S. Gero. The impact of working memory limitations on the design process during conceptualization. *Design Studies*, 28(4), 2007. doi: 10.1016/j.destud.2007.02.005.

[12] Alan F. Blackwell. What is programming. In *14th workshop of the Psychology of Programming Interest Group*, pages 204–218. Citeseer, 2002.

[13] Annette Bolte, Thomas Goschke, and Julius Kuhl. Emotion and intuition effects of positive and negative mood on implicit judgments of semantic coherence. *Psychological Science*, 14(5):416–421, 2003.

[14] Scott Chacon. *Pro git*. Apress, 2009.

[15] Bo T. Christensen and Christian D. Schunn. The role and impact of mental simulation in design. *Applied cognitive psychology*, 23 (3), 2009.

[16] Nigel Cross. Design cognition: Results from protocol and other empirical studies of design activity. *Design knowing and learning: Cognition in design education*, 2001.

[17] Peter DeGrace and Leslie Hulet Stahl. *Wicked problems, righteous solutions: a catalogue of modern software engineering paradigms*. Yourdon Press, 1990. ISBN 978-0135901267.

[18] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with Changeboxes. In *ICDL'07: International Conference on Dynamic Languages*, 2007. ISBN 978-1-60558-084-5. doi: 10.1145/1352678.1352681.

[19] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[20] Kees Dorst and Nigel Cross. Creativity in the design process: co-evolution of problem-solution. *Design Studies*, 22(5), 2001.

[21] S.P. Dow, A. Glassco, J. Kass, M. Schwarz, D.L. Schwartz, and S.R. Klemmer. Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 17(4):18, 2010.

[22] Steven P. Dow, Kate Heddleston, and Scott R. Klemmer. The efficacy of prototyping under time constraints. In *Conference on Creativity and Cognition*, 2009.

[23] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented software engineering. In *Proceedings of the 2007 International Conference on Dynamic languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 3–24. ACM, 2007.

[24] Jonathan St BT. Evans. Dual-processing accounts of reasoning, judgment, and social cognition. *Annu. Rev. Psychol.*, 59:255–278, 2008.

[25] Jeanne Farrington. Seven plus or minus two. *Performance Improvement Quarterly*, 23(4), 2011. doi: 10.1002/piq.20099.

[26] Baruch Fischhoff. Hindsight is not equal to foresight: The effect of outcome knowledge on judgment under uncertainty. *Journal of Experimental Psychology: Human perception and performance*, 1(3): 288, 1975.

[27] Joseph P Forgas and Rebekah East. On being happy and gullible: Mood effects on skepticism and the detection of deception. *Journal of Experimental Social Psychology*, 44(5):1362–1367, 2008.

[28] Apache Software Foundation. Subversion best practices, 2009. URL http://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html.

[29] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[30] Martin Fowler. Continuous integration, 2006. URL http://www.martinfowler.com/articles/continuousIntegration.html.

[31] Adele Goldberg. *SMALLTALK-80: the interactive programming environment*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[32] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[33] Gabriela Goldschmidt. The dialectics of sketching. *Creativity Research Journal*, 4(2), 1991.

[34] 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 *Symposium on User interface software and technology*, 2008.

[35] 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*, pages 161–170. IEEE Computer Society, 2011.

[36] Robert Hirschfeld, Bastian Steinert, and Jens Lincke. Agile software development in virtual collaboration environments. In Christoph Meinel, Larry Leifer, and Hasso Plattner, editors, *Design Thinking*, Understanding Innovation, pages 197–218. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-13756-3.

doi: 10.1007/978-3-642-13757-0_12. URL http://dx.doi.org/10.1007/978-3-642-13757-0_12.

[37] Stephanie Houde and Charles Hill. What do prototypes prototype? *Handbook of Human-Computer Interaction*, 2, 1997.

[38] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *OOPSLA'97: International conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1997. doi: 10.1145/263700.263754.

[39] Viewpoints Research Institute. STEPS toward expressive programming systems, 2010 progress report submitted to the national science foundation. Technical report, Viewpoints Research Institute, 2010.

[40] Michael Inzlicht and Brandon J Schmeichel. What is ego depletion? toward a mechanistic revision of the resource model of self-control. *Perspectives on Psychological Science*, 7(5):450–463, 2012.

[41] John Christopher Jones. *Design methods*. John Wiley & Sons, 2 edition, 1992. ISBN 978-0471284963.

[42] Natalia Juristo and Ana M. Moreno. *Basics of software engineering experimentation*. Springer Publishing Company, Incorporated, 2010.

[43] Marcel A. Just, Patricia A. Carpenter, and Akira Miyake. Neuroindices of cognitive workload: Neuroimaging, pupillometric and event-related potential studies of brain work. *Theoretical Issues in Ergonomics Science*, 4(1-2):56–88, 2003.

[44] Marcel Adam Just and Patricia A Carpenter. A capacity theory of comprehension: Individual differences in working memory. *Psychological review*, 99:122–149, 1992.

[45] Daniel Kahneman. *Thinking, Fast and Slow*. Penguin Books Limited, 2011. ISBN 9780141918921. URL http://books.google.de/books?id=oV1tXT3HigoC.

[46] Daniel Kahneman, Jackson Beatty, and Irwin Pollack. Perceptual deficit during a mental task. *Science*, 1967.

[47] Jannik Laval, Simon Denier, Stéphane Ducasse, and Jean-Remy Falleri. Supporting simultaneous versions for software evolution assessment. *Science of Computer Programming*, 2010.

[48] Bryan Lawson. *How designers think: the design process demystified*. Architectural press, 2006. ISBN 978-0750660778.

[49] Youn-Kyung Lim, Erik Stolterman, and Josh Tenenberg. The anatomy of prototypes: Prototypes as filters, prototypes as manifestations of design ideas. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 15(2), 2008.

[50] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Science of Computer Programming*, 76 (12):1194–1209, 2011.

[51] Christine A. Lindberg. *Oxford American Writer's Thesaurus*. Oxford Univ., 2008. ISBN 9780195342840. URL http://books.google.de/books?id=KakNMgAACAAJ.

[52] John H Maloney and Randall B Smith. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28. ACM, 1995.

[53] Eliot Miranda. The cog smalltalk virtual machine: writing a jit in a high-level dynamic language. 2011. URL http://design.cs.iastate.edu/vmil/2011/papers/p03-miranda.pdf.

[54] George V. Neville-Neil. Coder's block. *Communications of the ACM*, 54(4):34–35, April 2011. doi: 10.1145/1924421.1924434.

[55] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.

[56] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[57] Joseph Pelrine, Alan Knight, and Adrian Cho. *Mastering ENVY/Developer*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0-521-66650-3.

[58] Michael Perscheid, Batian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt. Immediacy through interactivity: Online analysis of run-time behavior. In *WCRE'10: Proceedings of the 17th Working Conference on Reverse Engineering*, volume 10, pages 77–86, Beverly, MA, USA, 2010. IEEE Computer Society. doi: 10.1109/WCRE.2010.17.

[59] Paul Ralph and Yair Wand. A proposal for a formal definition of the design concept. *Design Requirements Engineering: A Ten-Year Perspective*, pages 103–136, 2009.

[60] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, 2007.

[61] Romain Robbes and Romain Lanza. Characterizing and understanding development sessions. In *ICPC 2007: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 155–166. IEEE Computer Society, 2007.

[62] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *ISSRE'03: International Symposium on Software Reliability Engineering*, 2003.

[63] Donald A. Schön. *The reflective practitioner: How professionals think in action*. Basic Books (AZ), 1983.

[64] Donald A. Schön and Glenn Wiggins. Kinds of seeing and their functions in designing. *Design Studies*, 13(2), 1992. ISSN 0142-694X. doi: 10.1016/0142-694X(92)90268-F.

[65] Ken Schwaber. *Agile project management with Scrum*. O'Reilly Media, Inc., 2004.

[66] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. Houghton Mifflin, 2002. ISBN 9780395615560. URL http://books.google.de/books?id=o7jaAAAAMAAJ.

[67] Ben Shneiderman and Catherine Plaisant. *Designing the user interface: strategies for effective human-computer interaction*. Pearson Addison Wesley, 5 edition, 2009. ISBN 978-0321601483.

[68] Herbert A. Simon. *The sciences of the artificial*. The MIT Press, 1996.

[69] Diomidis Spinellis. Version control systems. *Software, IEEE*, 22 (5):108–109, 2005.

[70] Keith E. Stanovich and Richard F. West. Individual differences in reasoning: Implications for the rationality debate?-open peer commentary-understanding/acceptance and adaptation: Is the non-normative thinking mode adaptive? *Behavioral and Brain Sciences*, 23(5):645–665, 2000.

[71] Bastian Steinert and Robert Hirschfeld. Applying design knowledge to programming. In Hasso Plattner, Christoph Meinel, and Larry Leifer, editors, *Design Thinking Research: Studying Co-creation in Practice*, Understanding Innovation, pages 259–277. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-21642-8. doi: 10.1007/978-3-642-21643-5_15. URL http://dx.doi.org/10.1007/978-3-642-21643-5_15.

[72] Bastian Steinert and Robert Hirschfeld. How to compare performance in program design activities: Towards an empirical evaluation of coexist. In Larry Leifer, Hasso Plattner, and

Christoph Meinel, editors, *Design Thinking Research: Building Innovation Eco-Systems*, Understanding Innovation, pages 219–238. Springer International Publishing, 2014. ISBN 978-3-319-01302-2. doi: 10.1007/978-3-319-01303-9_14. URL http://dx.doi.org/10.1007/978-3-319-01303-9_14.

[73] Bastian Steinert, Michael Grunewald, Stefan Richter, Jens Lincke, and Robert Hirschfeld. Multi-user multi-account interaction in groupware supporting single-display collaboration. In *Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009. 5th International Conference on*, pages 1–9. IEEE, 2009.

[74] Bastian Steinert, Michael Perscheid, Martin Beck, Jens Lincke, and Robert Hirschfeld. Debugging into examples: Leveraging tests for program comprehension. In *Testing of Software and Communication Systems*, pages 235–240. Springer, 2009.

[75] Bastian Steinert, Michael Haupt, Robert Krahn, and Robert Hirschfeld. Continuous selective testing. In *Agile Processes in Software Engineering and Extreme Programming*, pages 132–146. Springer, 2010.

[76] Bastian Steinert, Marcel Taeumel, Jens Lincke, Tobias Pape, and Robert Hirschfeld. Codetalk conversations about code. In *Creating Connecting and Collaborating through Computing (C5), 2010 Eighth International Conference on*, pages 11–18. IEEE, 2010.

[77] 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. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384591. URL http://doi.acm.org/10.1145/2384577.2384591.

[78] Bastian Steinert, Marcel Taeumel, Damien Cassou, and Robert Hirschfeld. Adopting design practices for programming. In Hasso Plattner, Christoph Meinel, and Larry Leifer, editors, *Design Thinking Research: Measuring Performance in Context*, Understanding Innovation, pages 247–262. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31990-7. doi: 10.1007/978-3-642-31991-4_14. URL http://dx.doi.org/10.1007/978-3-642-31991-4_14.

[79] David L Strayer and William A Johnston. Driven to distraction: Dual-task studies of simulated driving and conversing on a cellular telephone. *Psychological science*, 12(6):462–466, 2001.

[80] 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. ISBN 978-3-540-43561-7.

[81] Masaki Suwa, Terry Purcell, and John Gero. Macroscopic analysis of design processes based on a scheme for coding designers' cognitive actions. *Design Studies*, 19(4), 1998. ISSN 0142-694X. doi: 10.1016/S0142-694X(98)00016-7.

[82] Marcel Taeumel, Bastian Steinert, and Robert Hirschfeld. The vivide programming environment: Connecting run-time information with programmers' system knowledge. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '12, pages 117–126, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1562-3. doi: 10.1145/2384592.2384604. URL http://doi.acm.org/10.1145/2384592.2384604.

[83] Wikipedia: the free encyclopedia. Design, May 2014. http://en.wikipedia.org/wiki/Design.

[84] Dave Thomas and Kent Johnson. Orwell — A configuration management system for team programming. In *OOPSLA'88: International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 1988. ISBN 0-89791-284-5.

[85] Sascha Topolinski and Fritz Strack. The analysis of intuition: Processing fluency and affect in judgements of semantic coherence. *Cognition and Emotion*, 23(8):1465–1503, 2009.

[86] Linus Torvalds and Junio Hamano. Git: Fast version control system. *URL http://git-scm. com*, 2010.

[87] David Ungar and Randall B Smith. *Self: The power of simplicity*, volume 22. ACM, 1987.

[88] Evie Vergauwe, Pierre Barrouillet, and Valérie Camos. Do mental processes share a domain-general resource? *Psychological Science*, 21(3):384–390, 2010.

[89] Bret Victor. Magic Ink: Information software and the graphical interface, 2005. URL http://worrydream.com/MagicInk/.

[90] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 143–156. ACM, 2007.

[91] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. Worlds: Controlling the scope of side effects. In *ECOOP'11:*

Proceedings of the 25th European Conference on Object-Oriented Programming, pages 179–203, Lancaster, UK, 2011. Springer. doi: 10.1007/978-3-642-22655-7_9.

[92] Benjamin Hosain Wasty, Amir Semmo, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. ContextLua: Dynamic behavioral variations in computer games. In *Proceedings of the 2Nd International Workshop on Context-Oriented Programming*, COP '10, pages 5:1–5:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0531-0. doi: 10.1145/1930021.1930026. URL http://doi.acm.org/10.1145/1930021.1930026.

[93] Robert J. Youmans. The effects of physical prototyping and group work on the reduction of design fixation. *Design Studies*, 32(2), 2011. ISSN 0142-694X. doi: 10.1016/j.destud.2010.08.001.

colophon

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of July 28, 2014 (`classicthesis` version 1.0).

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertation selbst angefertigt und nur die im Literaturverzeichnis aufgeführten Quellen und Hilfsmittel verwendet habe. Alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, sind kenntlich gemacht. Diese Arbeit oder Teile davon wurden nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht.

Ich versichere weiterhin, dass ich diese Arbeit oder eine andere Abhandlung nicht bei einer anderen Fakultät oder einer anderen Universität eingereicht habe.

Berlin, May 2014

Bastian Steinert