Interacting with Code: Observations, Models, and Tools for Usable Software Development Environments

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Jan-Peter Krämer

aus Dormagen

Berichter:	Univ.-Prof. Dr. rer. nat. Jan Borchers
	Joel Brandt, Ph.D.
	Univ.-Prof. Dr. rer. nat. Horst Lichter

Tag der mündlichen Prüfung: 23.11.2016

Contents

List of Figures

List of Tables

Abstract

Comprehending source code is an essential activity for software developers. It is not only required for software maintenance but also when developers want to reuse existing code. Many facilities in the development environment, such as software-based development tools, frameworks, or documentation, are designed to support software comprehension. Changing the design of these facilities does not only allow developers to perform their comprehension strategies more efficiently, but also changes the strategies they use. In this thesis, we aim to understand in more depth how the design of the development environment changes comprehension strategies. Our results contribute to the corpus of knowledge of the cognitive models of software developers, and can influence the design of future development tools.

First, we compare two frameworks to program animations. We show that one of the most important design variables for frameworks is the level of abstraction. On the one hand, abstractions can help to align the capabilities offered by a framework with the developers' high-level programming plans, on the other hand, abstractions can lead to misconceptions about the behavior of the framework when conceptual details are hidden.

Second, we study whether developers can be encouraged to write more documentation and unit tests by improving the interaction design for these tasks. Our interaction design promotes writing documentation and unit tests as part of the edit-test-edit cycle, and leverages runtime information from manual tests performed by developers to suggest updates to these documents. We found that for this interaction design to be successful, it is crucial to not only simplify the authoring task but to also provide near-term value for the developer.

Third, we explore in which way the use of call graph navigation tools changes when these tools are designed to not require any setup and to provide information continuously. We present two tools that implement this design, and we compare these tools to existing call graph navigation facilities. Developers using our tools could solve maintenance tasks faster, because they adapted a more effective navigation strategy. We also present a novel analysis method that we used to quantitatively describe the differences in navigation behavior.

Last, we analyzed the effect of live coding environments on the developers' behavior. These environments execute the application after every change automatically in order to present information about its runtime behavior. We find that developers using such tools fix bugs they introduced faster, because they switch between writing new code and debugging existing code more frequently. This behavior can reduce the task completion time for certain coding tasks. To be able to use live coding in real-world scenarios, we present an interaction technique that allows to use live coding on-demand for short code snippets. These snippets are automatically contextualized to simulate how the code snippet would behave if it was executed as part of the complete application.

Überblick

Das Verstehen von Programmquellcode ist eine der essentiellen Aktivitäten von Softwareentwicklern. Es ist nicht nur erforderlich um Wartungsarbeiten an existierender Software durchzuführen, sondern auch, wenn bereits existierender Quellcode wiederverwendet werden soll. Viele Aspekte einer Softwareentwicklungsumgebung, wie z.B. Software-basierte Werkzeuge, Frameworks, oder Dokumentation, sind so gestaltet, dass sie das Verständnis von Software unterstützen. Die Gestaltung dieser Aspekte hilft jedoch Softwareentwicklern nicht nur ihre existierenden Strategien um Software zu verstehen effizienter anzuwenden, sondern ändert welche Strategien sie nutzen. In dieser Arbeit werden wir untersuchen wie genau die Gestaltung der Entwicklungsumgebung die Strategien zum Verständnis von Quellcode beeinflusst. Unsere Ergebnisse erweitern das Wissen über die kognitiven Modelle von Softwareentwicklern und können die Gestaltung zukünftiger Entwicklungswerkzeuge prägen.

Zuerst vergleichen wir zwei Frameworks zur programmatischen Erstellung von Animationen. Wir zeigen, dass eine der wichtigsten Variablen in der Gestaltung von Frameworks die Wahl des Abstraktionsniveaus ist. Einerseits können Abstraktionen helfen, die Möglichkeiten die das Framework bietet mit den Problemlösungsstrategien der Entwickler in Deckung zu bringen, andererseits können Abstraktionen zu Fehlvorstellungen über das Verhalten des Frameworks führen, wenn konzeptionelle Details durch die Abstraktion versteckt werden.

Im nächsten Teil untersuchen wir, ob Entwickler motiviert werden können mehr Dokumentation und Komponententests zu erstellen, wenn die Interaktion für diese Aufgabe vereinfacht wird. Unser Interaktionsdesign propagiert, Dokumentation und Komponententests als Teil des Edit-Test-Edit-Kreislaufs zu schreiben. Um Änderungen an diesen Dokumenten automatisiert vorzuschlagen, nutzen wir Laufzeitinformationen, die während den manuellen Tests, die Entwickler als Teil dieses Kreislaufs ausführen, gespeichert werden. Wir fanden heraus, dass, um diesem Interaktionsdesign zu Erfolg zu verhelfen, das resultierende Werkzeug den

Entwicklern nicht nur bei der Erstellung von Dokumentation und Komponententests helfen muss, sondern darüber hinaus auch kurzfristig nützlich sein muss.

Im dritten Projekt untersuchten wir, wie sich die Nutzung von Werkzeugen zur Call Graph Navigation ändert, wenn diese Werkzeuge so gestaltet sind, dass sie vor der Nutzung nicht erst konfiguriert werden müssen und sie Informationen kontinuierlich anzeigen. Wir stellen zunächst zwei Werkzeuge vor, die diese Gestaltungsideen implementieren, und vergleichen danach diese Werkzeuge mit anderen typischen Methoden zur Navigation entlang des Call Graphen. Entwickler, die unsere Werkzeuge nutzten, konnten Wartungsaufgaben schneller lösen, weil sie eine effizientere Strategie zur Navigation anwendeten. Wir stellen außerdem eine neue Analysetechnik vor, die es uns erlaubte die Unterscheide zwischen Navigationsstrategien quantitativ zu beschreiben.

Zuletzt analysieren wir den Effekt, den Live Coding Umgebungen, also Entwicklungsumgebungen die die Applikation nach jeder Änderung automatisch ausführen um dem Entwickler Informationen über das Laufzeitverhalten zeigen zu können, auf das Verhalten der Entwickler haben. Wir fanden heraus, dass Entwickler, die solche Werkzeuge nutzen, ihre Fehler schneller beseitigten und häufiger zwischen dem Schreiben von neuem Quellcode und dem Korrigieren von existierendem Quellcode wechseln. Dieses Verhalten erlaubt es manche Programmierarbeiten auch schneller zu erledigen. Um Live Coding Werkzeuge auch unter realen Bedingungen einsetzen zu können, präsentieren wir eine Interaktionstechnik, die es erlaubt Live Coding spontan für kurze Teile des Quellcodes zu nutzen. Zu diesen Teilen des Quellcodes wird automatisch ein Kontext generiert, der repräsentiert wie sich der Code verhalten würde, wenn er als Teil der kompletten Applikation ausgeführt würde.

Acknowledgements

This thesis would not have been possible without the help of a lot of people. First of all, I want to thank my advisor Prof. Jan Borchers for his advice and encouragement, and for a great deal of individual freedom to pursue my ideas. He made the media computing group an amazing place to work, and he made sure to always maintain an environment in which work is fun.

Secondly, I want to thank Joel Brandt for inviting me to an internship at Adobe Research and for plenty of incredibly valuable advice ever after. His support went far beyond what I would have ever expected from an internship. Special thanks for offering to be a second advisor of this thesis and for our successful collaboration on numerous projects in this thesis.

I also want to thank Prof. Horst Lichter for offering to write a third review of this thesis that was only required due to my preferences regarding the committee.

During my 8 years at the media computing group, working as a student assistant, a diploma thesis student, and as a research assistant, I have meet an incredible amount of great people. All of you have made my time at i10 special and you all have contributed to this thesis in some way. Even though every one of my colleagues was part of that experience, I want to point out some of you in particular: Jonathan Diehl for getting me into the group in the first place and repeatedly getting me excited about new ideas. Thorsten Karrer for countless successful collaborations and mentoring. Chat Wacharamanothan for teaching me how to apply statistics. Leonhard Lichtschlag, Moritz Wittenhagen, Florian Heller, Maximilian Möllers, Simon Völker for their incredibly valuable feedback and discussions.

Throughout my time as a PhD student, I had the pleasure to work with a number of amazingly smart and dedicated students who chose me as their advisor. In chronological order: Joachim Kurz, Björn Heinen, Ewgenij Belzmann, Tanja Ulmen, Hendrik Wolf, Dennis Lewandowski, Michael Hennings. Thanks to all of you for contributing to the research in this thesis.

I want to thank Shangning Postel-Heutz for proofreading this thesis. Her timely responses and thoughtful corrections have been exceptional.

I want to thank Katharina for supporting me and helping me to keep calm through all stages of creating this thesis. I am also deeply grateful for my parents who have always supported me and given me the confidence to pursue a PhD.

Conventions

The whole thesis is written in American English.

Definitions of technical terms or short excursus are set off in colored boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

Source code and implementation symbols are written in typewriter-style text.

`myClass`

Download links are set off in coloured boxes.

http://hci.rwth-aachen.de/myFile

Some of the software and studies described in this thesis have been previously published by me and by students that I supervised. At the beginning of each chapter I provide a summary of collaborations and prior publications. It goes without saying that every project was done in collaboration with my advisor, Prof. Dr. Jan Borchers, hence, he is not listed repeatedly in each of these summaries.

Chapter 1

Introduction

"I [. . .] am rarely happier than when spending an entire day programming my computer to perform automatically a task that would otherwise take me a good ten seconds to do by hand."

—Douglas Adams, Last Chance to See

Software provides generalizable solutions to real-world problems.

Software has become an essential part of our daily lives. It helps us to perform office tasks more easily, it allows us to communicate with people, or it can simply entertain us. Software has also become irreplaceable in roles where the end user does not immediately notice it, e.g., when it prevents our cars from swinging off the road after breaking suddenly, or it controls our heating to keep the temperature in our homes comfortable. Generalizing from these examples, we establish that software solves a problem of a user in the real world. The key strength of software-based solutions to problems is that they are generalizable to multiple or all instances of a problem: For example, a system to prevent a car from swinging off the road is expected to work not only in one specific situation but in all dangerous situations a driver might encounter.

From the perspective of a computer, software is comprised of a set of calculations, decomposed into algorithms that operate on a set of data structures [Wirth, 1976]. Hence, a software developer needs to translate a generalizable so-

Programming means to transform a generalizable solution to a problem into a set of calculations and data structures.

lution to a problem in the real world into a suitable set of calculations and data structures, and express the latter in a way a microprocessor can understand. For this purpose, developers use *source code*, a text (or a visualization) following strict syntactical rules. Providing a general solution to real world problems by authoring source code is called *programming* [Pair, 2011].

Our vision is to design a development environments that helps to cope with complexity.

Programming is a difficult task, because, as F. P. Brooks J. [1987] states in a famous article, software entities are "more complex than perhaps any other human construct". In the same article, he also argues that there is no silver bullet to remove this complexity, because it is essential to make programming powerful enough to allow the creation of all novel, irreplaceable technologies mentioned before. Thus, our vision is not to remove the complexity of programming but to design an environment for programmers that helps them to cope with the inherent complexity of software. This could ultimately help making software development easier and more productive, and allow more people to experience the joy in software development that is reflected in the opening quote of this chapter. To make this vision come true, we first need to understand the cognitive processes of developers to find out in more detail which challenges they have to face in their daily work.

One of the most difficult tasks for programmers is software comprehension.

One of the most difficult tasks for programmers identified in previous research is to comprehend existing source code [LaToza et al., 2006]. Unfortunately, this comprehension is also essential for successful software development [Pressman et al., 2014], because programmers have to work with existing software in the majority of cases [Winograd, 1979]. The reasons for this are manifold: First, microprocessors can only perform a handful of rudimentary operations, and assembling a modern, complex application from only these operations would be extremely tedious. Thus, developers make use of reusable components, i.e., frameworks or libraries, that encapsulate frequently used algorithms and data structures, e.g., for file access or network communication. Second, modern software often evolves for years to fit the needs of users and adapt to changing requirements, thus, a majority of software development is spent modifying or extending existing software [Fisher, 1978; Winograd, 1979].

In this thesis, we investigate how tools affect developers' code comprehension strategies.

Many facilities in existing software development environments, such as, software-based tools, frameworks, or documentation, are designed to support the strategies developers use to comprehend software. Every change to the design of the development environment, though, is likely to change these comprehension strategies [Myers et al., 2004]. The goal of this dissertation, in a broad sense, is to understand in more detail how the design of facilities that support software comprehension affects the cognitive processes of software developers. To this end, we explored novel interaction designs that are informed by existing research about the developers' cognitive processes. Using prototypical implementations of the most promising designs, we ran empirical studies to analyze how the changed interaction affected the developers' code comprehension strategies. The insights gained contribute to the overall understanding of how the design of development tools influences code comprehension strategies, and they help to identify design guidelines for development tools that can ultimately make developers more productive.

1.1 Structure

In Chapter 2, we will discuss the existing research on both the developers' cognitive models and problem solving strategies. In particular, we find that strategies for source code comprehension are similar to those for comprehension of natural language texts [Pennington, 1987], hence, program comprehension can be supported by incorporating natural language text into the source code.

Choosing the right level of abstraction is crucial when designing an API.

In frameworks, i.e., reusable software components, natural language is incorporated when developers emphasize or deemphasize certain information through naming, and decide how these named entities are to be used together [Gilmore et al., 1984]. In Chapter 3, we report on a study that compared two different frameworks to programmatically create animations. Both frameworks are similarly powerful, i.e., they allow to solve similar tasks with a similar amount of source code. They differ in their design, more specifically, in the level of abstraction they provide. We will show that a higher level of abstraction allows developers to solve certain tasks faster and with less testing. However, we also find that a higher level of abstraction is prone to hiding crucial implementation details. In this case, abstraction can make comprehension harder instead of easier. The success of the framework's design is closely tailored to the task at hand, i.e., if the abstraction does not match the task its benefits diminish. This study is by far the smallest research project presented in this dissertation, yet it already shows how the developers' strategies are affected by the tools at their disposal. We find that by taking these insights into account, the developers' cognitive models proved to be an effective method for the purpose of understanding the benefits and limitations of framework designs.

Tools can encourage developers to write more documentation and unit tests, if they also provide near-term value for the task at hand.

Another common way to incorporate natural language text into source code is through documentation, i.e., explanations of the source code written by the developer. However, developers often do not write sufficient documentation, because the creation causes additional effort. The same is true for unit tests, which represent an expectation about the source code and are known to support software maintenance, but are usually not created sufficiently. In Chapter 4, we will explore if an improved interaction design for authoring documentation and unit tests can encourage developers to create these documents more. To this end, we introduce an authoring support tool called VESTA. VESTA's design is guided by the insight that developers frequently manually test the software they are building. We leverage runtime traces from these manual executions to inform the authoring interface. In a user study, we confirmed that VESTA's documentation authoring component worked as intended, i.e., it provoked developers to write more and more accurate documentation, while VESTA's unit test component only resulted in a minor improvement. By analyzing the developers' interactions with the tool more thoroughly, we found that the crucial design aspect for the success of VESTA's documentation component is that it provided near-term benefits for program comprehension besides supporting documentation authoring.

Source code is different from natural language text due to the fact that it is often not read sequentially. Instead, the source code to solve a single goal is often spread across various methods in various locations. Thus, previous research revealed that navigation along method calls, i.e., call graph navigation, is an exceptionally important activity during program comprehension. We designed and implemented two call graph navigation tools, Stacksplorer and Blaze, that incorporate two interface design concepts which were previously uncommon for such tools: First, our tools are in close spatial vicinity to the code editor and update automatically in response to the developers' navigations, hence, they continuously provide peripheral information. Second, the navigation options shown in the tools are constrained to allow only specific call graph explorations, either breadth-first (Stacksplorer) or depth-first (Blaze). In a comparative lab study we found that both our tools lowered the task completion times compared to a control condition, an IDE without dedicated call graph exploration facilities, and to the Call Hierarchy, a standard call graph exploration tool in today's IDEs.

We formulate and verify two design guidelines for call graph navigation tools.

In the next step, we wanted to find out if and how the success of our tools is related to the developers' comprehension strategies. We invented a generalizable analysis technique to quantitatively compare the navigation behavior of developers. Using this analysis technique, we were able to show that only Stacksplorer and Blaze caused developers to change their navigation strategies compared to the control condition while the Call Hierarchy did not. Our results are especially interesting because the information accessible using each tool is similar. Hence, this project demonstrates that the effect of a tool does not only depend on the information presented but also on the tool's interaction design.

The design of more effective call graph navigation tools causes developers to change their navigation behavior.

So far, we have assumed that developers understand source code primarily by reading it. However, they also regularly execute the source code and use tools to analyze its behavior at runtime. This provides a high level overview of goals and subgoals of the software, and, when using a debugger to inspect the runtime behavior in more detail, it is also valuable to understand low-level algorithms. Live

Developers working in a live coding environment detect bugs they introduce faster.

coding tools provide runtime information without requiring the developer to manually execute the source code. Instead, they re-execute the software automatically right after it was changed. While the idea of live coding tools is well known in previous research, implementing these tools with interactive update rates has only recently become technically feasible. In Chapter 6, we present METIS, our working prototype of a live coding tool. The development of this prototype allowed us to study the effect of live coding on the behavior of developers. We find that developers who use live coding to implement three coding tasks detect bugs they introduce faster and, hence, need less lengthy debugging phases. We also identified several technical pitfalls that still remain in live programming environments when used to develop real world software. Instead of solving these problems technically, we outline how we can avoid them by using a novel interaction technique that supports developers in selecting small chunks of source code to be executed live in the remainder of the chapter.

In the final chapter of this dissertation, we will summarize the results of the individual research projects we presented and outline which questions are left open for future work. We conclude with a discussion of how ideas from this thesis could be applied to other domains of knowledge work.

1.2 Contributions

To summarize, in this dissertation we make the following contributions:

- We show that, for the design of frameworks, the level of abstraction provided by the framework and the high-level programming plans formulated by developers should align. This causes developers to need significantly fewer manual tests to be confident that their code is correct. However, the behavior of the framework needs to be plausible on the level of abstraction it provides, i.e., developers are unlikely to understand hidden assumptions.

- We show how runtime information gathered during manual tests performed by the developers can be used effectively to encourage developers to author documentation and unit tests. Our key design idea is to integrating the authoring process into the edit-test-edit cycle that developers already perform. This ensures that the runtime information used is up-to-date, and developers can be expected to have a well-formed conceptual model of the code they are about to document and to test. In the analysis of our design, we learned that it is essential that the tool provides near-term value beyond authoring documentation and unit tests, in order to encourage developers to author these documents.

- We formulated two design guidelines for call graph navigation tools: *Proactive information visualization* means that tools should provide potentially relevant information automatically and continuously while the developer is navigating. *Comprehensible relevance* means that a developer needs to be able to easily understand how the information shown in the tool is relevant to the task. We implemented these guidelines in two newly designed call graph navigation tools. In a comparative study of our tools, between the widely-used Call Hierarchy tool and an IDE without dedicated call graph navigation tools, we found that all tools increase the task success rates but only tools implementing our guidelines also succeeded in reducing the task completion time. We found that the decrease in the task completion time correlates with a change in the developers' navigation behavior.

- We present a new analysis technique that allows to quantitatively compare the navigation behavior of developers. We use a set of predictive models and calculate the accuracy with which each model predicts the recorded navigation behavior. A vector of all prediction accuracies characterizes the navigation behavior. To test whether or not the analysis technique is sound, we applied it in one of our own studies and compared the results to other quantitative metrics.

- We show that developers use a different coding strategy when working in a live coding environment, i.e., a development environment that re-execute the application automatically after each change, compared to a traditional environment. Developers in a live coding environment are more likely to use the interleaved coding strategy, i.e., they switch between writing new code and debugging the code in shorter intervals. The interleaved strategy causes developers to fix bugs faster, and yields shorter task completion times for some tasks.

- Continuously re-executing the complete application is technically challenging. We present a new interaction design that allows to invoke inline editors, which are embedded into the source code editor, to edit a small code fragment live. The code fragments are automatically contextualized, to simulate their behavior during a full program execution. A preliminary evaluation shows that the increase in the task completion time that developers can achieve using this interaction design is comparable to the increase we found before for live coding environments.

Chapter 2

Theory

"Software: do you write it like a book, grow it like a plant, accrete it like a pearl, or construct it like a building?"

—Jeff Atwood

Comprehending software is an important activity for programmers, and one of the most difficult activities.

Comprehending software is required to perform nearly all programming tasks [Pressman et al., 2014]: It is needed for creating software, when reusing code from libraries, examples, or old projects, and when performing maintenance tasks on existing software, e.g., bug fixing or refactoring. Studies have also repeatedly found code comprehension to be among the biggest problems for software developers. For example, Ko et al. [2005] observed programmers working on a maintenance task in a lab study and found that comprehending source code was the most time-consuming activity. Several subtasks involved in code comprehension were found to be laborious, for example, developers spent 35% of their time on navigation and 46% of their time inspecting code that they later recognized to be not relevant for their task. These findings were also found for real-world software developers. When surveying developers at Microsoft, LaToza et al. [2006] found that understanding the rationale behind existing source code is the biggest problem for developers. To circumvent the difficult task of comprehending source code, developers often resort to the developer who originally created the code. This, however, creates more problems down the line: Even the original developer often does not remember the rationale behind his old code, and having to deal with these interruptions also negatively impacts this developer's productivity.

In this chapter, we will discuss existing research that analyzes developers' cognitive models of software, and the strategies they use for software comprehension. We will also discuss studies that analyze developers working with current development environments to identify common problems of existing development tools. The results presented in this section will be used throughout the thesis to provide a framework to motivate our interaction designs and to discuss our results. Research projects that describe new development tools are mostly excluded from this chapter. Instead, we discuss closely related research tools as part of the the following chapters, in which our individual research projects are presented.

2.1 Developers' Models of Software

Mental models of software are multi-dimensional.

As discussed in the introduction, programming means to express a general solution to a class of real world problems in a way that a computer can comprehend, i.e., by writing source code [Pair, 2011]. Consequently, the developer's mental model of software needs to be multi-dimensional: It is at least comprised of the solution to the problem in the application domain, and the algorithmic solution as well as data structures in the computing domain [Détienne, 2002]. To understand the mapping between the application domain and the algorithmic solution, which is expressed in source code, a developer forms a hierarchy of goals and subgoals and identifies the strategies involved to achieve each goal.

As a simple example, we consider an excerpt from a fictional software to manage students at a university: Our goal (in the application domain) is to sort a list of students in a course by grades. The following code implements a sorting algorithm to solve this problem:

```
 1 var length = students.length;
 2 for (var i = 0; i < length; i++) {
 3   for (var j = 0; j < (length - i - 1); j++) {
 4     if(students[j].grade > students[j+1].grade) {
 5       var tmp = students[j];
 6       students[j] = students[j+1];
 7       students[j+1] = tmp;
 8     }
 9   }
10 }
```

This simple example can already be decomposed into a multi-level hierarchy of goals and subgoals: On the highest level, the problem can be decomposed into the subgoals *representing data*, *allowing data input*, *sorting the list of data entries*, *presenting the result*. The code example only shows the solution for one subgoal: *sorting the list of data entries*. The strategy to achieve this subgoal is a bubble sort algorithm. It can be further decomposed into smaller subgoals. This time, the subgoals are more technical and more closely related to the computing domain, because they describe parts of the algorithm: The first goal is to take multiple passes such that in the *i*-th pass the *i*-th last element of the list is correctly sorted. This goal again has two subgoals: The first goal is to implement multiple passes, which is realized with the outer `for`-loop. The second goal again has two parts: First, to iterate over the unsorted part of the list, second, to swap each element with the adjacent one if these two are ordered incorrectly. This description of the bubble sort algorithm, even though all low-level goals are solely in the computing domain, still needs to be translated into actual source code. For example, the solution to swapping two entries in an array requires three lines of code (lines 5-6).

Programs can be decomposed into a multi-level hierarchy of goals and subgoals

2.1.1 (Delocalized) Programming Plans

Developers often know appropriate solutions for goals in these hierarchies from their experience. To describe this knowledge, the concept of schemas has been widely used [Adelson, 1981; Détienne, 2002]. A schema is a knowledge structure that represents a generic solution to a problem.

A programming schema represents a generic solution to a problem.

Schemas are, just like the goals developers need to achieve, hierarchical. For the example above, a developer would first activate a schema representing the bubble sort algorithm as a generic solution for sorting problems, and then other schemas representing, e.g., how to use a temporary variable to swap two elements in an array.

A programming plan is a hierarchy of schemas to solve a problem.

The hierarchy of schemas used by a developer to solve a given programming problem is called a *programming plan*. Soloway et al. [1984] showed that what makes expert developers comprehend source code faster is their ability to quickly recognize the plans of the original developer. Recognizing plans allows developers to reason about the code at a higher level of abstraction [LaToza et al., 2007], e.g., it allows experts to think about the complete code snippet above as "sorting" instead of needing to make sense of all individual statements. From the results by Détienne [2002] and Soloway et al. [1984] we can conclude that schematic knowledge can be applied in two ways: Either when creating source code to find a solution for a problem, or when comprehending source code to reconstruct which problem is solved by a solution encountered in the source code.

When different parts of a programming plan are implemented in different locations, the plan becomes delocalized. This is also called a cross-cutting concern.

In the example we discussed above, the complete programming plan was executed in one contiguous piece of source code. For example, we did not show the source code that defines the data model, implements data entry, or presents the results. All of these would be likely implemented elsewhere, so they can be reused in other parts of the software. Similarly, the sorting algorithm we discussed would likely be implemented in a library, so that it can be reused to sort arbitrary arrays. This causes a programming plan to become *delocalized* [Détienne, 2002]; other authors have referred to the different fragments of source code that achieve a single goal using the term *cross-cutting concern*. Cross-cutting concerns are the reason why Ko et al. [2005], in their lab study we mentioned before, found navigation to be an important activity in program comprehension.

Today, object-oriented software architectures are widely adopted, because they were found to model many application domains reasonably well [Meyer, 1997]. In object-oriented software, delocalized plans are especially common, because plans and the organization of source code into objects are orthogonal [Rist, 1996], i.e., a plan can use multiple objects and a single object can be utilized in multiple plans. However, cross-cutting concerns pose a challenge for developers who need to identify and relate all relevant parts of the source code. In Section 2.2.2 we will discuss the strategies developers use to identify delocalized plans and the problems they have to face.

Cross-cutting concerns are common in object-oriented software.

OBJECT-ORIENTATION:
In object-oriented software, runtime behavior is defined by the interaction of *objects*. At runtime, objects interact by calling each other's methods. Each object is an instance of a *class*, which defines a data structure, i.e., which properties an object has, and a set of methods. In this dissertation, we will sometimes refer to an object's class as its *type*. Each class can declare its own properties and methods, but also *inherit* properties and methods from a *parent class*. Using inheritance, classes form a hierarchy that consists of one or more trees. It is called the *static hierarchy* or *inheritance hierarchy*. Some implementations of object-orientation allow more than one parent class, resulting in a more complex hierarchy.
Using an object-oriented programming language is not required for object-oriented programming. However, nowadays these languages are common because they provide support for elementary operations such as defining a class, creating an instance of a class, and calling methods of objects.

Definition: *Object-orientation*

2.1.2 Cognitive Models

In the previous subsection, we have discussed software in terms of the goals and subgoals it achieves. Often it is useful to also use different and orthogonal cognitive models to describe developers' code comprehension strategies. In this section, we will describe some of these models that are inspired by research on text comprehension. Dijk et al. [1983] have found that for text comprehension, readers form three structures: The surface structure describes the verbatim

Readers of natural language text form three structures: Surface structure, textbase, and situational model.

structure of the text, i.e., the arrangement of words into sentences, paragraphs, etc. The *textbase* is a propositional structure that represents the actions and events in the text. The *situational model* represents what the text is about referentially. The surface structure and the textbase are both on linguistic grounds and represent only the verbatim text, while the situational model includes the interpretation of the text by the reader.

Cognitive models for reading source code are similar to those for reading natural language text.

A similar model was found suitable to model the comprehension process developers use for procedural [Pennington, 1987] and object-oriented [Burkhardt et al., 1997] source code. The surface structure and the textbase are, in the discourse about software comprehension, often combined in the *program model*. The program model represents the elementary operations and structural elements in the source code. The situational model is the delinearization of the source code, i.e., it is comprised of data flow, control flow, and the hierarchical goal structure we have discussed before. Vans et al. [1999] found that developers are more likely to solve bug-fixing tasks successfully, when the situational model they form includes several different levels of abstraction.

For object-oriented source code, the situational model is comprised of static and dynamic aspects.

Burkhardt et al. [1997] have refined the model above for the comprehension of object-oriented source code. They distinguished two components of the situational model: The static aspects contain information about the object hierarchy, the objects' relationships to the application domain, and goals of the software. The dynamic aspects reflect data flow, client-server relationships, and communication between objects at runtime. Burkhardt et al. compared their results to Pennington's [1987] earlier model, which describes the comprehension of procedural source code, in order to show that the developers working with object-oriented source code build the situational model earlier. This is a first example that shows that the choice of tools in a wide sense, in this case structural programming paradigms, influences how developers work and approach comprehension tasks. We will find this fundamental insight repeatedly in our own research projects, for both APIs (Chapter 3) and software tools developers use (Chapter 4–6).

Object-orientation promotes developing a thorough static situational model.

In the same study, Burkhardt et al. [1997] found that the static situational model is better developed than the dynamic situational model. Further, more experienced programmers create a better static situational model but not a better dynamic situational model. Burkhardt et al. hypothesize that this is a property of the object-oriented programming paradigm, i.e., object-oriented source code promotes building the static situational model.

Comprehending the dynamic situational model is harder than the static situational model.

LaToza et al. [2010a] investigated the difficulties involved in developing the dynamic situational model. They coined the term reachability question to describe a search along all feasible paths through a program. Answering reachability questions would primarily contribute to the dynamic situational model. In a series of studies, including a large-scale interview of professional software developers as well as studies in the lab and in the field, LaToza and Myers found that reachability questions are frequent but hard to answer. Nine of the ten longest activities they observed in their field study were concerned with answering a reachability question.

In object-oriented source code, the static situational model is usually represented explicitly in the surface structure of the code. In contrast, other tools are required to support developers in comprehending the dynamic situational model. In Chapter 5 we will argue that current tools only provide inadequate support to develop the dynamic situational model. We will then present new tools to support developers in answering reachability questions, and we will show that the most effective tools cause a change in developers' behavior while comprehending code. Whether or not tools cause this change depends on the tools' interaction design.

2.2 Strategies for Comprehension

In the last section, we have discussed how developers represent source code mentally, i.e., by using the program and situational model, or by using programming plans. In this section, we will discuss the strategies used to build these mental representations of source code. In a survey among more than 100 developers at Microsoft, LaToza et al. [2006] found that over 80% of the time spent in understanding software is spent on working with the actual source code, i.e., reading or using a debugger. The remainder of the total time is spent in examining documentation, check-in messages in version control systems, or asking colleagues for help. For the remainder of this section, we will focus on strategies to examine the actual source code.

2.2.1 Top-down or Bottom-up Comprehension

Developer using the bottom-up comprehension strategy, form the program model first.

As we have seen before, cognitive representations of software are characterized by a hierarchical mapping from the application domain into the programming domain. Several researchers have examined whether this mapping is created top-down or bottom-up. When Pennington [1987] confirmed that the model by Dijk et al. [1983] for text comprehension also applies to the comprehension of procedural programs, she noted that the program model is built before the situational model. This means that developers first comprehend the control flow and literal structure of the program before inferring the software's functions and higher level goals. Similarly, Shneiderman et al. [1979] observed that developers start by understanding small chunks of source code and iteratively organize these chunks into bigger functional units.

Developers using the top-down comprehension strategy, form the situational model first.

In contrast, Burkhardt et al. [1997] found that the situational model was better developed even in early phases of program comprehension. This corresponds to a top-down comprehension strategy, i.e., hypotheses about the functions of the program are developed first. The hypotheses are vague initially and are refined and checked iteratively using information obtained from the source code. While the results by Burkardt et al. are specific to object-oriented software, the top-down comprehension strategy was found in numerous studies in a variety of programming paradigms: R. Brooks [1983] was among the first to theorize that the software comprehension process was top-down and hypothesis driven. This theory could be confirmed by Vessey [1989] in a series of lab studies using procedural source code. Later, LaToza et al. [2007] observed thirteen developers while comprehending a large object-oriented open-source software project and also found that "program comprehension is driven by beliefs about facts".

Top-down strategies are common for maintenance tasks.

Top-down strategies were found to be especially common for developers working on maintenance tasks. They often start by forming a situational model of the correct program that they can use as a reference [Vessey, 1989]. Starke et al. [2009] noted that this early situational model is usually a hypothesis based on past experiences.

Developers often switch between strategies as needed.

When comparing the various studies mentioned above, we find that the tasks developers had to perform are slightly different in each one. For example, in the study by Pennington [1987], where the bottom-up strategy was observed, participants had 45 minutes to read source code without a specific task. Afterwards, they first answered comprehension questions, and then worked on a given maintenance task. In contrast, in the study by Vessey [1989], where the top-down model was observed, participants worked on a maintenance task right from the start. Further, listings showing correct and incorrect program output could be used as a reference. These differences in task descriptions and setup are likely to alter the behavior of developers, i.e., whether they prioritize to build the program or the situational model [Mills et al., 1995]. Developers do not only switch their strategies depending on the task but might also utilize top-down and bottom-up strategies while working on a single task, switching between both strategies as needed [Corritore et al., 2001; Letovsky, 1987]. In summary, these results show that the task at hand, in addition to the tools used, substantially influences the behavior of developers.

2.2.2 Navigation Strategies

So far, we have distinguished between top-down and bottom-up comprehension strategies, which both align with the hierarchical structures found in the cognitive mod-

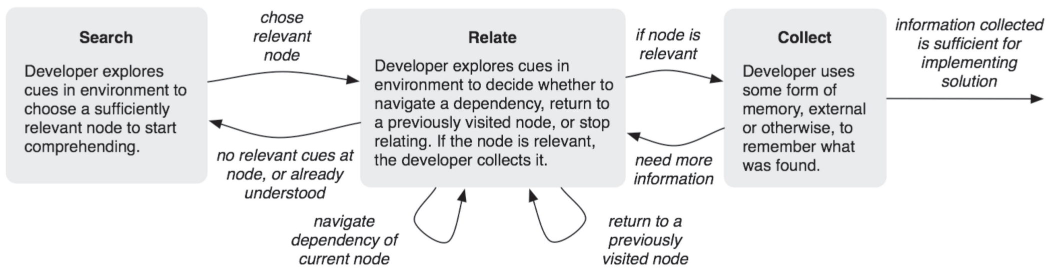

Search
Developer explores cues in environment to choose a sufficiently relevant node to start comprehending.
chose relevant node
no relevant cues at node, or already understood
Relate
Developer explores cues in environment to decide whether to navigate a dependency, return to a previously visited node, or stop relating. If the node is relevant, the developer collects it.
navigate dependency of current node
return to a previously visited node
if node is relevant
need more information
Collect
Developer uses some form of memory, external or otherwise, to remember what was found.
information collected is sufficient for implementing solution

Figure 2.1: The three-phase navigation model by Ko et al. [2006] shows how developers identify delocalized plans in source code.

els developers form about software. In this section, we will focus on developers' strategies to comprehend delocalized programming plans.

Effective developers need to understand cross-cutting concerns.

In an early study about program comprehension, Littman et al. [1987] found that developers working on a maintenance task in a lab study were only successful if they performed a structured analysis of the program, i.e., tried to form a coherent situational model. In contrast, developers using an as-needed strategy for program comprehension only focused on local program behavior and failed to successfully solve the task. Robillard et al. [2004] ran a similar study but using a substantially larger code base (about 65k SLOC instead of 250 SLOC). They could confirm that developers that read extended sections of the source code to comprehend local program behavior are less effective than those that try to comprehend delocalized plans by examining multiple locations in the source code to ultimately form a complete situational model. The challenges involved in the comprehension of delocalized plans are substantial enough to ultimately compromise the quality of real-world software: An analysis of open-source software projects found that the more scattered a concern is the more likely it is to contain defects [Eaddy et al., 2008]. Surprisingly, this effect was independent of the total amount of source code being involved of the concern.

The key problem for developers who want to apply the more successful structured strategy, which involves the identification of delocalized plans, is to find which locations in the code belong to a plan. Weiser [1982] coined the term *slicing* to describe the strategy of following the data flow to identify a delocalized plan. Francel et al. [2001] performed two lab studies with computer science students to show that developers who use slicing when debugging are more effective in identifying bugs and faulty parts of the source code than those who do not use slicing.

Following the data flow (slicing) is an effective strategy to comprehend cross-cutting concerns.

The information relevant to a cross-cutting concern is always scattered around multiple locations in the source code. Consequently, navigation is an important activity developers perform while trying to understand these concerns. Ko et al. [2006] presented a *three-phase navigation model* of how developers navigate when comprehending delocalized programming plans (Figure 2.1). For this model, we assume that the program and its metadata, such as documentation and commit messages, can be represented by a graph. This graph contains the various pieces of individual information as nodes, and different types of relationships as edges. Relationships can represent the semantics of the source code, e.g., calls, uses, and subclassing, but they can also, e.g., link a piece of the source code to its documentation. Developers start by looking for information relevant to their task in the environment (*search* phase). Once the developer has found a relevant node, he or she attempts to understand the node in the context of semantically related nodes, i.e., by navigating along edges in the graph (*relate* phase). Finally, if the information found turns out to be relevant, the developer collects it in some form of memory until he or she has understood enough information for the given task (*collect* phase). The information that developers collect to work on a task is often referred to as *working set* [Ko et al., 2005]. The three-phase navigation model is not strictly linear, instead developers are expected to frequently switch back to the search or relate phases if they did not yet find sufficient information for the task at hand yet.

The three-phase navigation model represents developers' navigation behavior while comprehending cross-cutting concerns.

The three-phase navigation model is compatible with all of the results we discussed before: The structured exploration required in the relate phase is consistent with research showing that structurally guided exploration is required to perform successful maintenance [Littman et al., 1987; Robillard et al., 2004]. Top-down and bottom-up com-

The three-phase navigation model is compatible with a wide variety of strategies.

prehension strategies can both be represented by choosing a high-level or a low-level node in the initial search phase. We found further support for the three-phase navigation model as part of the research presented in Chapter 5.

Information foraging theory (IFT) models how users search information.

The three-phase navigation model provides no insight into how developers identify relevant information in the search phase and how they pick which relationships to explore in the relate phase. *Programmer Flow by Information Scent (PFIS)* [Lawrance et al., 2008] is a model to describe this selection process. It is based on *Information Foraging Theory (IFT)* [Pirolli et al., 1999], a model that describes how people search information by using an analogy to predators hunting for prey. In IFT, the predator is a user who is hunting for an information need, i.e., the prey. In pursuit of the prey, a user can navigate to information resources (patches) by following links between the patches. IFT assumes that users follow the link with the strongest *information scent*, i.e., that is most likely to be relevant. In the original formulation, the information scent of a link depends on the linguistic similarity between the words in the link and words describing the information needed.

IFT can be adapted to programmers, to model how they identify relevant source code.

In PFIS, the adaption of IFT to model developers' navigation through source code, the predator is the developer hunting for an information. Every piece of information that is accessible in the development environment is a patch, e.g., variable definitions, methods, and documentation. Which links exist between patches depends on the capabilities of the development environment. In a modern IDE this often means that every identifier in the source code is a link, as well as search results and items shown in various navigation tools. To determine the scent of each link, PFIS compares a task description, e.g., a bug report, to the words used in the patch, i.e., the identifier in the source code. This approach is consistent with previous results that showed the importance of identifier naming for developers' comprehension process [Liblit et al., 2006]. In several iterations, PFIS was improved to also consider code structure to calculate information scent and to account for changing information needs during an exploration session [Lawrance et al., 2010; Piorkowski et al., 2011]. Studies in which the recorded navigation patterns of developers have been compared to the prediction created using PFIS have shown that the navigation target that PFIS predicted was correct up to 21% of the time, which is an impressive result considering the huge number of patches available. We will use the concept of information scent repeatedly throughout this thesis to explain results we have found in our studies.

2.3 Use of Development Tools

So far, we have described developers' mental models and the strategies they employ on an abstract level. In this section, we will relate these results to existing development tools. Providing a complete overview of existing research about novel development tools is beyond the scope of this thesis, instead, we will focus on results that we need to refer back to later.

2.3.1 APIs and Documentation

When using a framework, developers need to first comprehend the high-level design patterns it imposes.

Today, nearly all software is created using libraries or frameworks that provide common functionality for a given task. For example, graphical user interfaces are usually created using a user interface toolkit (UITK) that provides reusable interface components. The terms library and framework are often used interchangeably, but in this thesis we will use a stricter definition: A *library* is a collection of reusable components, e.g., methods. A *framework* or *API* is a collection of object-oriented components and includes a definition of patterns that describe how the components interact and should be used together [Johnson, 1997]. Because a framework includes patterns, it constraints the solution space for problems and requires developers to adapt their schematic knowledge to be applicable in this space. This causes problems for developers: They often know the programming plan they want to use to implement a feature but struggle to understand how to express this plan using the API at hand [Mandelin et al., 2005]. As APIs become more and more complex, these problems often arise early in the design process, as developers first need to comprehend the high-level design patterns imposed by the framework [Robillard, 2009].

Employing a user-centered design process helps creating comprehensible APIs.

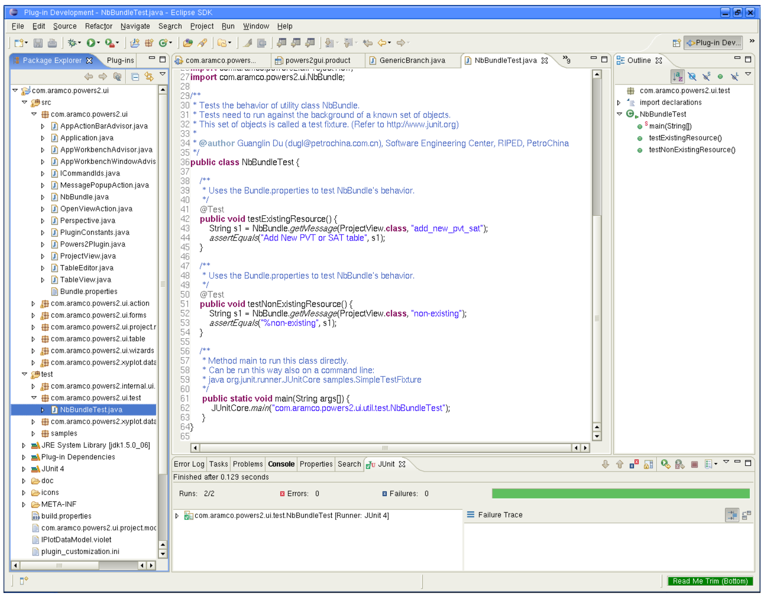
These issues have led to efforts that employ user-centered design to the APIs. Clarke [2004] adapts twelve factors that have an impact on the usability of development tools [Green et al., 1996] and reports that the Microsoft Visual Studio usability group has successfully used these factors in the design of new APIs. In a similar API design process, Stylos et al. [2008] found the most problematic design aspect was choosing an appropriate level of abstraction. In Chapter 3, we will present a study that explores in more detail the effect of different abstraction levels in APIs for designing animations.

Examples are an effective way to explain a framework's design.

A common strategy of developers to deal with problems in comprehending design patterns in frameworks, is to copy and adapt example code. Fairbanks et al. [2006] proposed that frameworks should include example code for recurring programming tasks. Using these examples has several benefits: Because they are authored by the original author of the API, they are known to be correct. Also, they can support developers reading the code later, because they can recognize the example used to derive the design intent of the original developer.

Copying examples can help exploring design alternatives in rapid edit-test-edit cycles.

In current development practice, examples are often found on the web [Brandt et al., 2009b; Hartmann et al., 2011]. Brandt et al. [2009b] report that frequent web searches are often part of an opportunistic coding strategy, that prioritizes speed and ease of coding over robustness and maintainability. Developers following this strategy use code to develop ideas and experiment instead of following a well thought out plan. To experiment efficiently, opportunistic programmers perform very short edit-test-edit cycles, i.e., they execute their application frequently to test it. One study of opportunistic programmers found 80% of edit-test-edit cycles to be shorter than 5 minutes [Brandt et al., 2009a]. Short edit-test-edit cycles in combination with opportunistic exploration of design alternatives seem to increase programming performance especially for novice developers [Hundhausen et al., 2009]. In Chapter 6 we will show that a development environment that executes the



```
import com.aramco.powers2.ui.NbBundle;

/**
 * Tests the behavior of utility class NbBundle.
 * Tests need to run against the background of a known set of objects.
 * This set of objects is called a test fixture. (Refer to http://www.junit.org)
 *
 * @author Guanglin Du (dugl@petrochina.com.cn), Software Engineering Center, RIPED, PetroChina
 */
public class NbBundleTest {

  /**
   * Uses the Bundle.properties to test NbBundle's behavior.
   */
  @Test
  public void testExistingResource() {
    String s1 = NbBundle.getMessage(ProjectView.class, "add_new_pvt_sat");
    assertEquals("Add New PVT or SAT table", s1);
  }

  /**
   * Uses the Bundle.properties to test NbBundle's behavior.
   */
  @Test
  public void testNonExistingResource() {
    String s1 = NbBundle.getMessage(ProjectView.class, "non-existing");
    assertEquals("%non-existing", s1);
  }

  /**
   * Method main to run this class directly.
   * Can be run this way also on a command line:
   * java org.junit.runner.JUnitCore samples.SimpleTestFixture
   */
   public static void main(String args[]) {
     JUnitCore.main("com.aramco.powers2.ui.util.test.NbBundleTest");
   }
}
```

```
Finished after 0.129 seconds
Runs:  2/2        Errors:  0        Failures:  0
com.aramco.powers2.ui.test.NbBundleTest [Runner: JUnit 4]
```

Plug-in Development - NbBundleTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Plug-ins
com.aramco.powers...
powers2gui.product
GenericBranch.java
NbBundleTest.java
Outline
Error Log Tasks Problems Console Properties Search JUnit
Failure Trace
Read Me Trim (Bottom)

Figure 2.2: Eclipse is representative of a modern IDE. A text editor in the center is augmented with numerous tools for structurally guided navigation, unit testing, and other tasks. Source: http://www.eclipse.org/screenshots/

application continuously, i.e., no longer requires the developer to test manually, can encourage developers to adapt opportunistic coding strategies and helps them to reduce the amount of bugs in their code. In Chapter 4 we will reveal how we can encourage opportunistic programmers to create unit tests and documentation, i.e., increase maintainability, by designing a tool that easily integrates into their current workflow and provides immediate benefits.

2.3.2 Software Tools

IDEs bundle a variety of development tools.

Today, developers have various tools at their disposal that support program comprehension and creation. These tools are often bundled in *integrated development environments (IDEs)*. A common modern IDE, such as Eclipse[1], Xcode[2],

[1]https://eclipse.org/ide/
[2]https://developer.apple.com/xcode/

or Visual Studio[3], is built around a text editor that is augmented with numerous structurally guided navigation tools, semantic editing tools, and a debugging interface (see Figure 2.2). We will not go into detail of all available tools, as this would be an insurmountable task. Instead, we will discuss what is known about how developers use existing tools and which problems remain. Tools that are closely related to the individual research projects we present, are discussed as part of the respective chapters.

Existing tools to support structurally guided navigation are often avoided by developers.

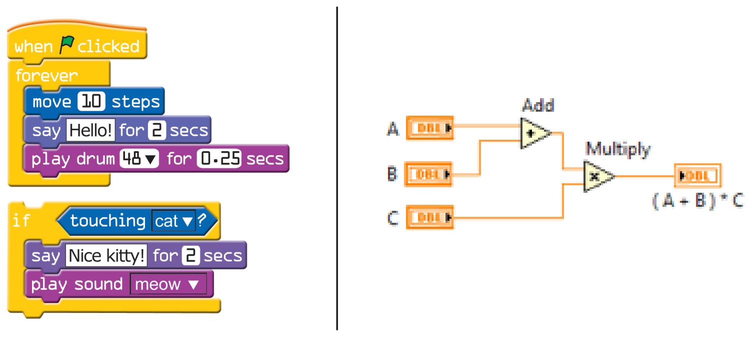
As discussed in Section 2.2.2, one of the biggest problems developers have to face in code comprehension tasks is the recovery of information about delocalized programming plans. Support for this task appears to be inadequate in many current IDEs. For example, Ko et al. [2005] observed that all developers used structurally guided navigation tools in Eclipse to solve a given maintenance task, but only two developers did so more than once. Instead, developers preferred simpler tools, such as a project-wide textual search, to find information. Ko et al. attribute this observation to the interaction design of Eclipse's more advanced tools that were found to be cumbersome to use.

Many IDEs do not support the collection of relevant information in a working set.

Once a developer found a useful piece of information, according to the three-phase navigation model, he or she wants to add it to a working set of all relevant information. Support for this behavior was repeatedly found missing in current IDEs [De Alwis et al., 2006; Ko et al., 2005; Sillito et al., 2008]. The only common way to collect multiple relevant source code locations is to open the various files in different tabs. However, tabs represent individual files instead of the specific piece of information relevant to a developer, hence developers often search through the open tabs to retrieve information from the working set [Ko et al., 2005]. Tabs are also mutually exclusive, i.e., only one tab's content is visible at a time. This makes it hard for developers to relate the different pieces of information [Green et al., 1996], causing frequent back-and-forth switches between two related code locations [De Alwis et al., 2006].

The problems we have discussed so far concerned the interaction design of existing tools. The interface design, i.e.,

[3]https://www.visualstudio.com/



```
when 🏳 clicked
forever
  move 10 steps
  say Hello! for 2 secs
  play drum 48 ▾ for 0.25 secs

if touching cat ▾ ?
  say Nice kitty! for 2 secs
  play sound meow ▾
```

Figure 2.3: Two examples of visual programming languages. Left: Code snippets from Scratch [Resnick et al., 2009], a visual programming language. Source code statements are displayed with rich visual formatting and can be assembled using drag-and-drop. Image Source: [Resnick et al., 2009]. Right: A simple LabVIEW program. LabVIEW uses a metaphor resembling an electrical circuit. Methods are represented by graphical objects instead of textual statements. In-/Output of these objects can be connected with virtual wires. Image Source: http://www.ni.com

the visual appearance of information and interactive elements, also has a substantial effect on how efficiently developers can perform software comprehension. Early studies already showed that the secondary notation, i.e., the formatting of the program text, can influence how developers comprehend source code [Green et al., 1996]. Even only changing the indentation level already significantly impacted the developers' program comprehension performance [Miara et al., 1983].

Text formatting in the source code editor affects how well programmers comprehend source code.

Secondary notion does not only refer to text formatting. More degrees of freedom in the visual layout of programs are used in visual programming languages. Visual programming is defined to be "any system that allows the user to specify the program in a two(or more)-dimensional fashion" [Myers, 1990]. A plethora of visual programming tools has been created, and researchers have found repeatedly that these tools engage novices into programming, help them to learn faster, and even support the transition to textual languages [Hundhausen et al., 2009; Resnick et al., 2009]. The large freedom in secondary notation does

Visual programming tools allow to specify programs using multi-dimensional layouts.

not come without a downside though: To utilize schematic knowledge during program comprehension, developers need to rely on recognizable patterns. Establishing these patterns, however, is harder if the notation allows even more freedom than text. This can hinder program comprehension especially for novices, leading to a significantly reduced program comprehension performance compared to merely using textual languages [Petre, 1995]. These problems are especially pronounced in larger, more complex programs [Fry, 1997], which explains why visual programming has been found beneficial for learning but is not widely used in professional programming.

In this thesis, we will not try to make a case for or against visual programming. We believe that due to the enormous amount of textual source code that already exists, it will be around for years to come. In this dissertation, we will focus on tools for textual programming languages. Nevertheless, we will refer back to the research on visual programming and the effect of secondary notation when discussing our designs for development tools.

2.4 Conclusion

Current development environments are not holistically designed around the developers' comprehension process.

In this section, we gave an overview of existing research on the cognitive models developers form when comprehending source code and the strategies they use during the comprehension process. In a brief summary of studies on developers' interaction with existing development tools, we found that these often do not support the strategies developers employ. This substantiates previous critique voiced by Sillito et al. [2008], who argued that the current development environments only provide a collection of tools instead of being holistically designed to support developers' comprehension strategies.

Nevertheless, the problem is more complex: We repeatedly found that not only tools should support existing strategies of developers, but also that developers change their strategies in response to certain tools. Hundhausen et al. [2002] found that the effect of the strategies a tool encourages can be bigger than the effect of the actual information displayed in the tool. This is also echoed in a quote by LaToza et al. [2010b]:

> "Understanding the strategies by which developers answer questions holds the potential to both reveal new opportunities for tools and to make it easier to understand how and why developers use the tools they do."

How tools and the strategies employed by developers are interdependent is the fundamental research question inspiring all projects presented in this thesis.

Chapter 3

Programming Paradigms for Animation

"Animation can explain whatever the mind of man can conceive. This facility makes it the most versatile and explicit means of communication yet devised for quick mass appreciation."

—Walt Disney as quoted in OpenGL Shading Language (2006) by Randi J. Rost

Publications: The work in this chapter was done in collaboration with Joel Brandt and Michael Hennings. It was funded in part by Adobe Research. The content of this chapter is based in part on a previous publication of this work as a short paper at the CHASE workshop 2016 [Krämer et al., 2016b]. The materials used in the study and the data collected have been published under the CC0 dedication to allow for open access [Hennings et al., 2016a,b; Krämer et al., 2016c]. Michael Hennings ran the experiment described in this chapter as part of his Master's thesis [Hennings, 2016]. The author of this thesis supervised this Master's thesis, formulated the goals and research questions of this work, and contributed to the design of the experiment and the analysis of the results.

The design of a framework imposes which programming schemas should be applied to solve certain problems.

In Chapter 2, we learned that strategies for understanding source code are similar to those for understanding natural language texts. Consequently, one way to improve the comprehensibility of source code is to incorporate natural language deliberately. This can be applied to the design of frameworks: By choosing names for variables and functions, framework authors can emphasize or deemphasize certain information [Gilmore et al., 1984]. By choosing how these named entities are supposed to be used together, the author of the framework imposes which programming schemas should be applied to solve a certain problem.

The level of abstraction is one of the most important design properties of frameworks.

As a result, the design of the framework influences how developers approach tasks and which strategies they employ. We have learned in Section 2.3.1 that one of the most important design properties of APIs is the level of abstraction they provide. Identifying the most suitable level of abstraction for a given task can be challenging. For example, Stylos et al. [2008] found for one API that its usability could be improved by increasing the level of abstraction, while Green et al. [1996] pointed out that increasing abstraction can create hidden dependencies and, thus, lead to problems. In this chapter, we will compare two frameworks to programmatically author animations, and analyze how the different levels of abstraction in these frameworks affect the developer's behavior.

Animations are widely used in digital artifacts.

Animations are widely used in a variety of scenarios and tasks [Dahotre et al., 2010]. The most obvious example is their use in movies, either as special effects or to render a complete movie. But they are also added to user interfaces to provide feedback and convey a sense of plausibility [Chang et al., 1993], or in scientific visualizations to add another dimension to the display of data. What makes animations interesting for us, is that they frequently appear in digital artifacts. Hence, it is not uncommon for software developers to be tasked with creating animations.

An animation defines property changes of an object over time.

Many animation can be abstracted as follows: Given a scene that contains a number of objects, where each object has a set of animatable properties, such as position or size, an animation defines how and when each of these properties changes over time [Parent, 2012]. Animations differ in how these property changes are defined. For example, motion capturing allows to generate timing information for property changes of an object from a motion that is recorded in the real world. Alternatively, a physics simulation can be used to generate similar data. For this research project, we focus on *artistic* animations, i.e., the animator defines the property changes in an artistic process [Parent, 2012].

Animations can be authored using graphical tools or programming languages.

Very broadly, two kinds of authoring tools for animations can be distinguished: graphical tools and textual programming languages. Graphical tools are frequently used, if the scene and the objects in the scene are fixed. They allow animators to previsualize how the finished animation will look like. Also, developers can use direct manipulation [Shneiderman, 1983] to modify the properties of each object at any given point in time during the animation. If the scene and objects are not fixed, it often makes sense to define the animation programmatically. This allows developers to modularize their animation definition and reuse animation source code for various animation tasks. For example, an animation can be implemented once and be applied repeatedly to various interface elements. Similarly, when animating a character, a developer can implement a breathing animation that runs continuously and adapts to the character's other actions.

In existing frameworks for the programmatic creation of animations, we found two prevalent paradigms: *procedural* and *declarative*.

We found two prevalent programming paradigms implemented in existing frameworks to support the programmatic creation of animations: In *procedural* frameworks, the developer implements an update method that is called by the system at fixed time intervals. The update method is responsible for updating the properties of all objects for the given point in time. In *declarative* frameworks, the developer defines the animation by providing tuples that specify the value of an object's property at a point in time during the animation. These tuples are called keyframes. Once all keyframes are specified, the system runs the animation automatically by interpolating all property values between two keyframes. The interpolation function can be changed by the developer if needed. We can interpret the difference between procedural and declarative animation programming as a difference in the abstraction provided by

```javascript
function render(time) {
  var loc = time % 300;
  if (loc > 150) {
    loc = 150-(loc % 150);
  }
  ball.top = loc;
}

window.requestAnimationFrame(render);
```

```javascript
var actor = animator.addActor({
  context: ball
});

actor.keyframe(0, {top: 0});
actor.keyframe(150, {top: 150});
actor.keyframe(300, {top: 0});

animator.play();
```

Listing 1: This first code snippet is a procedural implementation of a ball moving up and down. The second snipped implements the same movement declaratively. Example from [Krämer et al., 2016b]

either paradigm. While using a procedural framework the developer needs to calculate all intermediate property values during an animation, in a declarative framework this task is abstracted and performed by the library. An example of both a declarative and a procedural definition of an animation is shown in Listing 1.

In the remainder of this chapter, we will report on a within-groups lab study in which we compared a procedural with a declarative animation library in terms of how developers work with the respective libraries.

3.1 Comparing Declarative and Procedural Animation Frameworks

We performed a within-groups experiment with two conditions: *procedural* and *declarative*. Our hypothesis is that the higher abstraction provided in the declarative framework allows for an easier mapping of high-level programming plans to framework functions. We expect that this results in a) a lower task completion time when using a declarative framework, and b) in users preferring the declarative condition.

3.1.1 Setup

Study participants animated HTML elements using JavaScript, once using a procedural and once using a declarative framework.

In both conditions, participants used JavaScript to animate HTML elements on a website. The conditions differed in the JavaScript framework used: In the procedural condition, developers used JavaScript's native animation frames. They allow to register a callback that is called for every frame and has to update the properties of all animated HTML elements. The predominant technique to animate HTML elements declaratively is to specify the animation using CSS. However, we opted to use Rekapi[1] in the declarative condition instead. Rekapi is a JavaScript framework which makes its syntax more comparable to JavaScript's animation frames that are used in the procedural condition. The examples in Listing 1 show both APIs in use. There are plenty of other procedural and declarative animation frameworks for JavaScript we could have chosen but in our study we designed the tasks to only require the most fundamental features of the frameworks used. We assume that these features exist virtually in any other implementation of the respective programming paradigm as well. We provided cheat sheets containing the necessary API documentation for each condition [Hennings et al., 2016a], which were designed following the guidelines by Mayer [1997].

Participants had to implement five animation programming tasks once in each condition. The starting condi-

[1]http://rekapi.com/

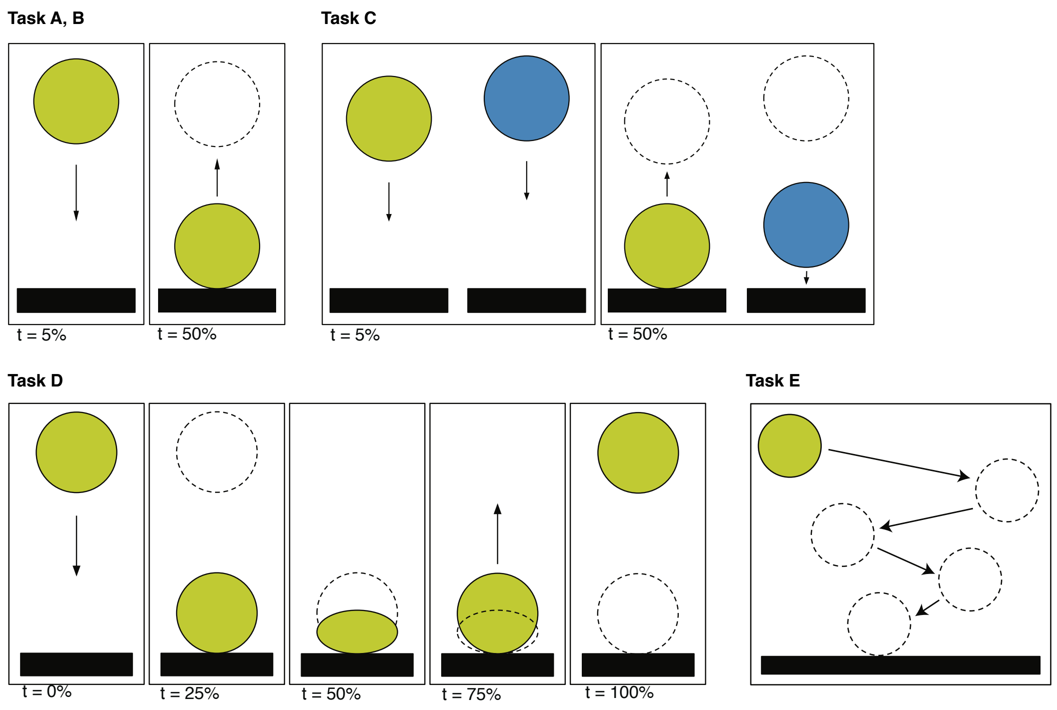


Figure 3.1: Participants had to implement each of the tasks above using both a procedural and a declarative animation framework. Figure adapted from [Krämer et al., 2016b].

Participants implemented five animations once in each condition.

tion was randomized for each participant, while keeping the number of participants starting in each condition equal. Participants solved each task in both conditions before proceeding to the next task. The duration of the study was fixed to two hours, i.e., when time ran out later tasks were skipped. Participants had to solve the following tasks (see also Figure 3.1):

Task A Participants had to animate a ball that continuously moves up and down on a black bar. A realistic bouncing animation was not required, i.e., the ball did not need to deform when touching the bar, and it should move at a constant speed.

Task B Participants had to add easing to the movement of the ball, i.e., the ball should accelerate and decelerate while moving up and down. Rekapi provides pre-made easing functions that can be attached to a keyframe. They change the interpolation function used to calculate property values between the keyframe and the previous state. To allow a fair comparison, we provided similar helper functions in the procedural condition.

Task C Participants had to add a second ball that moves like the first one but with a slight delay.

Task D Participants had to modify the bouncing animation of a single ball to include a deformation of the ball when it hits the ground. The key challenge in this task is that the ball's position is controlled by its offset from the top. While the ball deforms and the height is reduced, the offset from the top needs to be adjusted continuously to compensate for the loss in height to keep the bottom of the ball steady on the black bar.

Task E Participants had to implement a ball that moves downwards and follows a pendulum-like movement, i.e., it oscillates from left to right while falling. After arriving on the black bar, the animation should stop.

In all tasks, we provided a code skeleton that included the complete HTML and CSS code as well as all JavaScript code except for the animation itself. For Tasks B and C, the JavaScript skeleton included a working solution of the previous task.

We provided all required HTML and CSS code.

The tasks were designed in an iterative design process that included multiple pilot studies to make sure that they could be feasibly solved within two hours. Further, the tasks were designed to be representative of common animations in user interfaces that often only change few properties of an object over time. For example, in Keynote[2], a popular tool to author presentation slides, 40% of all animations to reveal an object can be recreated by changing no more than three properties of the object over time.

The tasks were designed to be realistic but solvable in two hours.

Before the study, participants had to fill out a questionnaire to assess their coding skills and their previous experience with programmatically defining animations. We informed

We extended the Brackets editor to support the study procedure.

[2]http://www.apple.com/mac/keynote/

the participants that we wanted to compare two different types of frameworks but did not reveal our research hypotheses before the study. During the study, all participants worked using an identical 15-inch laptop computer, optionally using an external mouse and keyboard. Participants had to use the Brackets[3] code editor that we modified to support the study procedure. At the start of a task, Brackets opened all required documents, including the code skeleton and the task description. Participants had read-only access to their own solutions for previous tasks. Brackets included a play button to open the animation in the Google Chrome[4] browser that also provided a debugger. When participants wanted to finish a task, they could submit their solution using Brackets. This caused all open files to be saved and closed and the next task to be started. In the background, Brackets recorded timestamps of all scroll events, key strokes, and manual code executions. We also recorded a video of the computer screen throughout the trial.

3.1.2 Results

14 students participated in our study.

We recruited 14 students to participate in our study, all of them were majoring in computer science or a related field. Participants were on average 24.1 years old ($SD = 3.4$) and reported to spend an average of 14.3 hours ($SD = 7.6$) per week programming. Three participants created animations regularly, while six reported to rarely or never implement animations. Two participants knew Rekapi before the study and reported to have rudimentary knowledge about it.

Most participants did not solve all tasks in time.

Most participants did not manage to complete all tasks within the two-hour time limit. Table 3.1 shows how many participants submitted a solution for each task, and how many of these solutions were correct.

First, we compared the developers' subjective ratings of both libraries. Half of all developers preferred the declara-

[3]http://brackets.io
[4]https://www.google.com/chrome/

Task	A		B		C		D		E	
Condition	P	D	P	D	P	D	P	D	P	D
Submitted	14	14	13	13	12	11	7	7	2	2
Succesful	13	14	8	13	12	10	4	6	2	2

Table 3.1: For every combination of task and condition, this table shows the number of participants handing in a solution and the number of correct solutions.

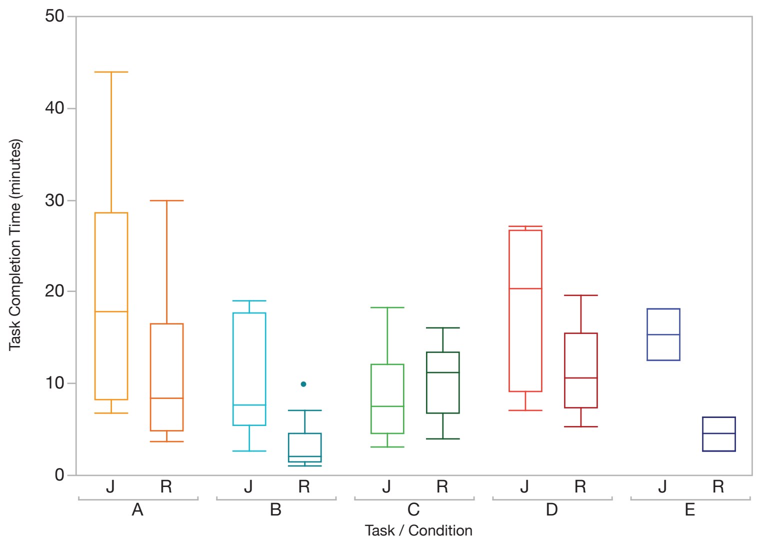


Figure 3.2: Participants solved tasks 2.4 times faster on average in the declarative condition. Task C is a notable exception, in this tasks developers on average needed more time using the declarative framework.

tive condition, while only one developer preferred the procedural condition. The remaining six participants reported that they liked the declarative condition better but felt that the expressiveness of the declarative framework was limited. Hence, they would have liked a combination of both approaches that allowed them to switch to a procedural implementation as needed. Several participants suggested to introduce a special type of keyframe that defines a timespan during which properties are not interpolated but have to be updated manually in a callback, as in the procedural condition.

Only one participant preferred the procedural condition.

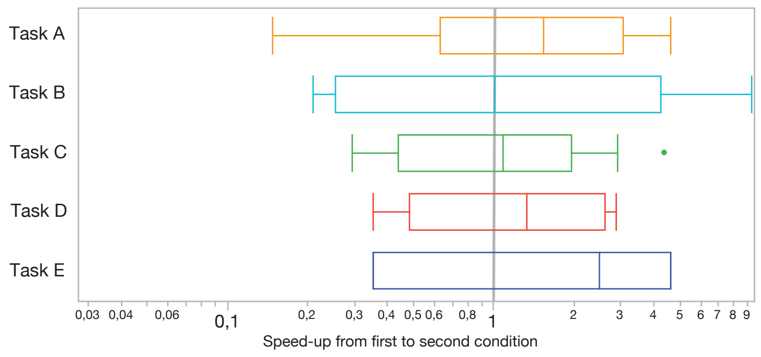


Figure 3.3: Participants solved every task twice, once in each condition. The figure shows how much faster each task was solved in the second trial. Values greater than one indicate that participants finished faster in the second trial.

On average, participants solved tasks 2.4 times faster in the declarative condition.

To analyze the developers' performance, we compared the task completion times of the developers who solved a task in both conditions (see Figure 3.2). We excluded Task E due to the low number of participants completing this task. Condition and task had a significant effect on task completion time with participants in the declarative condition solving tasks 2.4 times faster on average.

$$\begin{aligned} &\text{Task:} && F(3, 53.37) = 7.62 && \mathbf{p = 0.002} \\ &\text{Condition:} && F(1, 50.68) = 13.4 && \mathbf{p = 0.006} \\ &\text{Interaction:} && F(3, 50.25) = 3.71 && \mathbf{p = 0.017} \end{aligned}$$

We observed a significant interaction effect between task and condition. In task C, unlike any other tasks, developers were faster in the procedural condition. We found no learning effect caused by repeating the same task in both conditions, i.e., when solving the same task again in a different condition participants were not significantly faster (see Figure 3.3).

Developers performed less manual tests when using the procedural framework.

To compare the developers' strategies between both conditions, we use two different analysis methods. We started with a quantitative analysis of the number of manual executions of the animation, i.e., the number of edit-test-edit cycles (see Figure 3.4). Previous research has shown that frequent edit-test-edit cycles (see Chapter 2) are especially common in animation creation tasks [Brinkmann, 2008]. We found a significant effect of condition on the number of manual executions per task, and a significant interaction between condition and task.

Task:	$F(4, 67.6) = 2.05$	$p = 0.0.097$
Condition:	$F(1, 60.5) = 10.1$	$\mathbf{p = 0.002}$
Interaction:	$F(4, 60.5) = 2.93$	$\mathbf{p = 0.028}$

In the declarative condition, developers performed significantly less manual executions then in the procedural condition. Closer inspection of the results shows that the observed difference is caused by Task A and B, while the number of manual tests is nearly equal for Task C and D. We found no effect of condition on the test frequency, i.e., when normalizing the number of tests to the task completion time.

We expected solutions using the declarative framework to be more consistent.

The second analysis of the developers' strategies compared the consistency of the solutions created by different participants and of the strategies used to implement these solutions. We expected that developers can easier express their high-level programming plans if the framework provides a higher level of abstraction, because fewer intermediate programming schemas need to be instantiated. Hence, we expect to find less diversity in solutions in the declarative condition.

Task A was solved nearly identical by all participants in the declarative condition.

We could only confirm this hypothesis for task A: In the declarative condition, nearly all participants implemented the same programming plan that includes three keyframes (see Listing 1). Only one participant came up with an alternative solution and calculated keyframes for all intermediate positions of the ball. In the procedural condition, we found two different strategies, each of which was used by half of the participants. The first strategy was to calculate the ball's offset from the starting position as a function of elapsed time (see Listing 1). The second strategy was to move the ball by a fixed distance in every frame. Both strategies result in a visually similar animation, however, only using the first strategy the speed of the animation is independent of the system refresh rate.

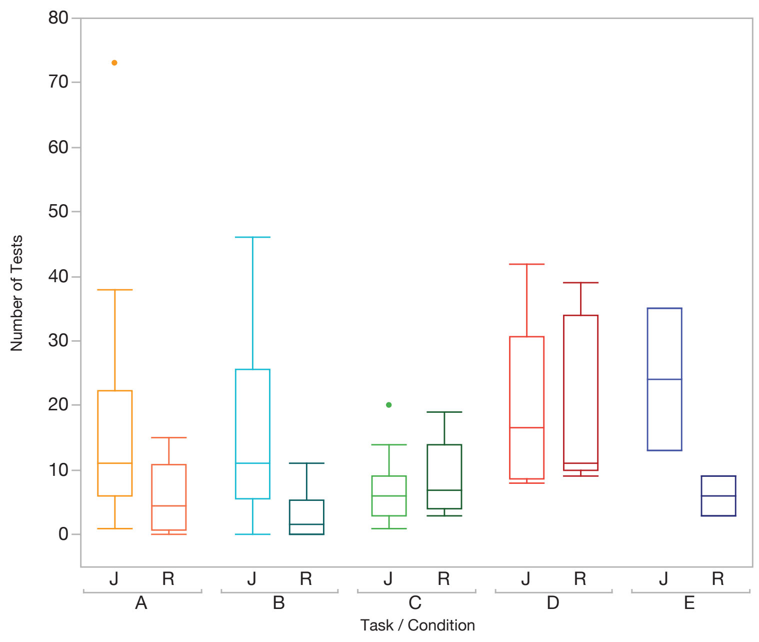


Figure 3.4: The figure shows how often developers manually tested the animation by task and condition. In the declarative condition, developers performed significantly less manual tests. This result is caused mostly by a substantial difference in Task A and B.

In task B, solutions were consistent in both conditions.

In task B, we found consistent strategies in both conditions: All participants who solved the task successfully ended up using the provided helper functions. One participant in the procedural condition and two participants in the declarative condition split the animation into two parts in order to then use separate helper functions for easing in and easing out. Two participants in the declarative condition added easing to all keyframes, even though adding it to the starting keyframe has no effect.

To solve task C, all developers in both conditions copied existing code.

For task C, strategies were not only consistent within each condition but also across all conditions. All developers in both conditions solved the task by copying the first animation and altering the copied code to implement the delay. No developer attempted to use modularization and parametrization, e.g., by implementing a method to animate an arbitrary ball with an arbitrary delay.

The strategies we observed in task D were again mostly consistent in both conditions. In the procedural condition, three participants moved the ball further downwards while changing its height. One participant implemented a state machine to distinguish two phases of the animation. In the first phase, the ball falls and in the second phase, the ball deforms. In the declarative condition, five participants added keyframes to represent these two phases. One participant introduced a separate animator for the deformation of the ball, i.e., a set of keyframes that is animated using a separate timeline. Hence, the participant needed to manually synchronize two animators, one for the deformation and one for the downward movement.

For task D, we observed some diversity in both conditions.

3.1.3 Discussion

Participants subjectively preferred declarative animation programming. This preference aligns with many quantitative results we found: Developers were significantly faster using the declarative framework, and they performed fewer manual tests. For two of the tasks, we found that several participants could not solve them at all unless they used the declarative framework.

Overall, the declarative framework was preferred and more efficient.

In the task-wise analysis of the developers' strategies, we found several examples in which the abstraction provided by the declarative framework is helpful: For example, in task A, we observed that the declarative framework successfully hides low level implementation details. This prevented potentially unintended results, such as the animation speed being related to the system frame rate. In task B, participants again benefit from the higher level of abstraction, because they do not need to worry how easing affects each intermediate position of the ball. This allows them to solve the task faster and with more confidence in their solution. Even if developers did not fully comprehend how the framework works, e.g., when adding easing to the starting keyframe, the framework proved to be fault resistant and the animation worked as intended. In task D, we observed one participant implementing the declarative concept in the procedural condition. This supports our hy-

In many tasks, the abstraction provided in the declarative framework matched developers' programming plans.

pothesis that the declarative paradigm aligns well with the developer's high-level programming plans.

When developers get stuck, they want to go back to the procedural paradigm.

Despite these positive results in favor of declarative animation programming, several developers indicated that they would like to have the option to switch to the procedural paradigm if needed. This is surprising, because in our study both frameworks were sufficient enough to solve all tasks. We assume that understanding how to utilize the framework was sometimes more difficult in the declarative condition than in the procedural condition.

In task C, the abstraction provided by the declarative framework led to unexpected behavior.

During the study, we first found support for this hypothesis in task C, which is the only task where participants were slower on average in the declarative condition than in the procedural condition. All but one developer first attempted to copy the keyframes that existed for the first ball and adapt them for the second ball. To implement the delay, they then moved all new keyframes back on the timeline by a fixed delay. This extends the complete length of the timeline, i.e., when the animation loops the first ball will move up and down and then stop until the second ball has also reached the top. The problem was that the timeline was not explicitly represented in code but created implicitly by adding keyframes. In this case, the abstraction provided led to unexpected behavior for the developer. This effect was severe enough to mitigate all positive effects of the declarative paradigm. To solve the problem, developers resorted to a trial-and-error strategy, in which they tested various parameter values. Thus, the number of manual tests is comparable to the procedural condition. Trial-and-error exploration of possible parameter values is a common but ineffective method that developer use to avoid completely comprehending the existing code [Détienne, 2002], in this case the abstract concepts of the declarative framework.

The declarative paradigm provided little support for animating two dependent properties concurrently.

In task D, we again found no clear benefit of declarative animation programming over procedural animation programming. The key difficulty in task D is that the ball's position has to be changed according to its height, i.e., two dependent properties are animated concurrently. The declarative paradigm does not provide any abstraction that helps with this particular problem. As a result, we found that developers needed a similar amount of testing as in the procedural conditions. However, more participants completed the task successfully in the declarative condition, and participants could solve the task faster. We assume that for task D the abstraction provided by the declarative framework lowers the cognitive load of developers, because it allows them to reason about a solution to the problem on a higher level of abstraction.

3.2 Conclusion and Future Work

The abstraction provided by a framework should align with developers' high-level programming plans.

We found that developers could implement many animations faster using a declarative framework than using a procedural framework. Our results indicate that this is caused by a better coherence between the level of abstraction provided by the framework and the high-level programming plans formulated by developers. This influences the developers' testing behavior: They need significantly fewer manual tests to be confident that their code is correct. We believe that this result has implications for the design of frameworks, in general: If common high-level programming schemas are known, the level of abstraction in the framework can be designed to align with these schemas. This reduces the depth of the hierarchy of lower level schemas that is required to solve the programming task.

One problem we observed was caused by a lack of abstraction suitable to support the implementation of animations that need to change multiple dependent parameters concurrently. This problem is consistent with the positive remarks above: It exemplifies that for programming schemas that do not match the exposed interface of the framework, abstraction yields no benefits.

A framework should expose all underlying concepts that affect its behavior.

The second problem we observed points to a potential pitfall in the design of frameworks with a high level of abstraction: Developers are likely to only comprehend a framework at the level of abstraction they need to apply programming schemas. In our study, this led to unexpected behavior because the interface exposed by the framework did not facilitate comprehension of important underlying concepts, e.g., the fact that all keyframes define a timeline.

We contribute two design recommendations for designers of APIs.

We conclude that considering the developers' conceptual models when designing a programming framework is important to successfully support developers. This result is consistent with previous case studies on the design of frameworks [Green et al., 1996; Stylos et al., 2008]. These previous studies pointed out that the level of abstraction is one of the most crucial design aspects for programming frameworks. In our example, we specifically compared frameworks that provide a different level of abstraction and found two design recommendations: First, the designer of a framework should try to find well-known high-level programming schemas developers use and design the abstractions in the framework to make it easy to express these plans. Second, the designer should check which underlying concepts need to be exposed to prevent unexpected behavior.

We believe there is potential for future improvements of tools for the programmatic design of animations:

Specialized tools could support developers in repeatedly testing their animation.

First, we observed that developers often resorted to short edit-test-edit cycles when problems occurred. These could be more effective when using a live coding environment, i.e., restarting the animation automatically after every change. We will discuss live coding environments for general purpose programming in more detail in Chapter 6. Animations are special, though, because all intermediate states are transient, hence, important timing and movement details can be hard to spot. Also, if the animation is very long the relevant part of it might be visible only after a substantial waiting period. Future research projects should explore how to visualize important information about a continuously restarted animation effectively.

Dependencies between multiple animated properties could be expressed using constraints.

Second, we propose that concurrently changing properties can be modeled more effectively when introducing constraints as another abstraction into declarative animation frameworks [Duisberg, 2009; Oney et al., 2012b]. A constraint should represent a temporal or spatial relationship between objects, that is maintained throughout the animation. For example, this would allow to solve task C by animating the first ball and defining two constraints for the second ball. The first constraint fixes the second ball's x-coordinate at 200px right of the first ball. The second constraint fixes the second ball's y-coordinate to be the first ball's y-coordinate 500ms ago.

Chapter 4

Authoring of Documentation and Unit Tests

"I don't think a program is finished until you've written some reasonable documentation. [. . .] I think it's unprofessional these people who say, 'What does it do? Read the code.' The code shows me what it does. It doesn't show me what it's supposed to do."

—Joe Armstrong, from an interview in Peter Seibel – Coders at Work

Publications: This work started during an internship of the author at Adobe Research under supervision of Joel Brandt. The collaboration and this project continued after the internship ended, and was partially funded by Adobe Research. This project was published before as a short paper at CHI 2016 [Krämer et al., 2016a]. The author contributed to the definition of the research questions, the interaction design, the study setup, and the analysis of the results. He also implemented the prototype and administered the user study.

Documentation and unit tests describe expectations about the source code.

In the previous chapter we found that the design of frameworks affects the behavior of developers. A huge part of a framework's design is made up through its use of natural language for the naming of variables and functions. Another important and common way to use natural language in source code is by including documentation. Documentation is a plain text comment written by the author of the code to explain the expected behavior of the source code to readers, i.e., other developers. Documentation does not alter the behavior of the program, it is ignored by the computer when the software is executed. Unit tests are similar to documentation in that they also describe an expectation about the software and do not alter the behavior of the software. For example, documentation can describe that a parameter is expected to have the specified type, or unit tests can describe that a function is expected to always throw an exception when certain parameter values are passed in. In contrast to documentation, unit tests can be executed to enable a computer to verify that the expectation is met.

Documentation and unit tests increase software maintainability but are often missing or out-of-date.

Documentation and unit tests are well known to facilitate the construction of the situational model [Détienne, 2002] and, thus, support software maintainability [Bhat et al., 2006; Kotula, 2000]. However, creating documentation and unit tests is laborious and writing them does not immediately improve the application. The benefit of maintaining up-to-date documentation and unit tests is often only obvious in hindsight. As a result, these documents are often missing or are out of date [Kajko-Mattsson, 2001; Maalej et al., 2014; Taylor et al., 2002]

JavaScript source code represents few expectations about a program explicitly.

We decided to work on this problem in the context of JavaScript. JavaScript is a dynamically and weakly typed language, i.e., the type of a variable can change at runtime and is generally unknown at compile time. This flexible type system yields call sites that are potentially polymorphic, because which implementation of a method is called, depends on the current type of a variable. As a result, JavaScript source code represents few expectations for a program explicitly. This makes documentation and unit tests particularly valuable.

Several tools have been proposed to specify expectations about JavaScript applications, using either static or dynamic analysis. Many static tools perform type inference and check for errors based on the inferred types. Several of these tools have gained attention in the developer community, e.g., Flow[1], TypeScript[2], or Tern[3]. Often the type inference is bootstrapped using initial manual source code annotations. These can be provided either by using a JavaScript dialect that syntactically allows typed variables, or by providing JSDoc[4]-formatted function documentation. Even with manual type annotations, the dynamic features of JavaScript always limit the capabilities of static analysis techniques [Richards et al., 2010]. Dynamic tools inspect the actual program execution to circumvent this problem. Commonly, these tools first identify potentially invariant expectations about a program. This information can be used, for example, to create corresponding unit tests that ensure that the invariants hold [Artzi et al., 2011; Mesbah et al., 2009].

Static and dynamic approaches can be used to infer some expectations about code automatically.

Despite the variety of existing tools to support capturing expectations about the source code, developers still do not create sufficient documentation and unit tests in many real-world projects [Taylor et al., 2002]. We believe that this originates from two reasons: First, the interaction design of existing tools requires developers to change their workflow, e.g., by asking them to manually create type annotations, or by requiring them to setup and regularly execute a dynamic analysis tool. These imposed requirements on the workflow of developers can impair their flexibility, which is one of the key reasons for developers to use a dynamic language [Paulson, 2007]. Second, as discussed above, documentation and unit tests have little near-term value. Hence, developers are likely to prioritize other tasks that they would consider to be more productive.

Current tools require developers to change their workflow, while providing little near-term value.

In this chapter, we will investigate whether or not an improved interaction design can encourage developers to create more documentation and unit tests. Our key idea is

[1]http://flowtype.org/
[2]http://www.typescriptlang.org/
[3]http://ternjs.net/
[4]http://usejsdoc.org/

VESTA integrates documentation and unit test authoring into the edit-test-edit cycle.

to integrate these activities into the edit-test-edit cycle that developers already perform – in one study, 80% of these cycles were shorter than 5 minutes [Brandt et al., 2009a]. We implemented our idea in VESTA, an authoring tool for documentation and unit tests. It uses runtime traces of the manual executions developers perform to suggest updates to documentation and unit tests. Further, while actively working on a part of the source code, developers should have a rich situational model of the code. Consequently, right after a successful test would be an ideal time to write corresponding unit tests and documentation. The downside of our design is that VESTA can be used only in a workflow in which tests and documentation are written after the actual source code, i.e., we do not target proponents of test-first development. VESTA shares this limitation with all tools that rely on static or dynamic analyses of the existing application to generate information.

In the next section, we are going to present previous research projects related to our ideas. After that, we will present a detailed rationale for VESTA's design, and finally we will report on a study to evaluate VESTA's effect on developers' documentation and unit test authoring practice.

4.1 Related Work

Runtime analysis is often performed to analyze erroneous behavior.

Developers usually manually execute their program to test its correctness. Most tools analyzing these manual executions, such as a traditional debugger, support this task and focus on debugging erroneous executions. For example, tools can allow to recreate failure states repeatedly [Burg et al., 2013], or retrospectively analyze the execution to find which sequence of method calls caused erroneous behavior [Ko et al., 2009]. Theseus [Lieber et al., 2014] aimed to provide timely information about the manual execution: It visualizes calls to each method immediately after it was called during the manual execution. VESTA is different from these tools in two ways: First, while all of these runtime analysis tools focus on analyzing erroneous executions, VESTA utilizes error-free test runs best. Second, all of these tools support developers in the primary task for which they perform a manual execution, i.e., testing the correctness of the program, whereas VESTA uses runtime information to encourage developers to work on a different task once the program works as intended.

Mixed-initiative interfaces implement a collaborative interaction between a system and the user.

VESTA's interface is based on the concept of mixed-initiative user interfaces [Horvitz, 1999]. These interfaces implement a collaborative interaction in which a system provides automatically generated information that the user can easily customize and extend. Several other authoring tools for documentation and unit tests from research [Buse et al., 2010; Tan et al., 2012] and industry (e.g., JAutodoc[5]), are built around the same concept.

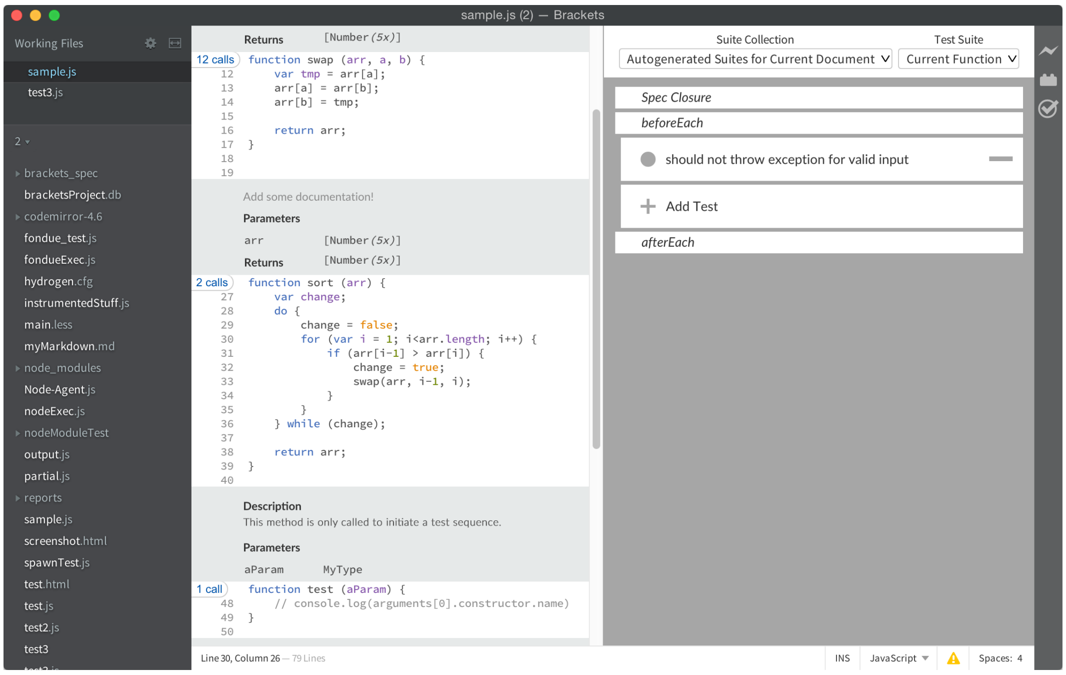
Different static and dynamic techniques could complement VESTA's capabilities in the future.

To generate the system provided information, tools can use either static or dynamic analysis techniques. Using static analysis, Sridhara et al. [2010] analyzed a function's body to identify the most important function calls and then translates these into a natural language description. Buse et al. [2010] were able to automatically document the effect of a change to the source code, or possible exceptions thrown by a function [Buse et al., 2008]. Dynamic analysis, as used in VESTA, was previously used in Daikon [Ernst et al., 2007] to search for likely invariants, e.g., invariable variable types or fixed values. The Daikon invariant detector has been used successfully to generate suggestions for new unit tests [Xie et al., 2006]. Atusa [Mesbah et al., 2009] also utilizes dynamic analysis, and searches possible paths through the user interface of a website to generate tests that assert user specified invariants. Research on different techniques to automatically generate information about invariants is orthogonal to VESTA. Many of these projects could complement VESTA's capabilities in the future and, hence, benefit from our interaction design.

In spreadsheets, runtime information to support test authoring is always available.

Rothermel et al. [2000] explored, similar to our research question, how the generation of tests for spreadsheets can be integrated into the workflow of spreadsheet users. Users can check off cells as correct, which generates a test case that ensures that given the current input of relevant input cells, the output cell shows the current value. Because a spreadsheet updates automatically immediately after a cell

[5]http://jautodoc.sourceforge.net/



```
Returns        [Number (5x)]
12 calls  function swap (arr, a, b) {
              var tmp = arr[a];
              arr[a] = arr[b];
              arr[b] = tmp;

              return arr;
          }


Add some documentation!
Parameters
arr            [Number (5x)]
Returns        [Number (5x)]
2 calls   function sort (arr) {
              var change;
              do {
                  change = false;
                  for (var i = 1; i<arr.length; i++) {
                      if (arr[i-1] > arr[i]) {
                          change = true;
                          swap(arr, i-1, i);
                      }
                  }
              } while (change);

              return arr;
          }

Description
This method is only called to initiate a test sequence.
Parameters
aParam         MyType
1 call    function test (aParam) {
              // console.log(arguments[0].constructor.name)
          }
```

Figure 4.1: This screenshot shows an overview of Brackets with the VESTA extension. VESTA adds two authoring interfaces: Documentation editors are embedded into the source code editor, the unit test editor is located on the right side of the editor.

or formula is changed, recent runtime results to extract a test case are always available. We intent to complement their work and explore a similar interaction in an environment that relies on the developer to manually execute the program as part of his or her existing workflow.

4.2 The prototype VESTA

Our prototype analyzes only function parameters and return values.

To prototype the interaction we suggest, we used a simple dynamic analysis technique that is only capable of analyzing function parameters and return values. In consequence, the prototype of VESTA is only able to create function documentation. We believe that this subset of documentation is already useful for developers, while being reasonably easy to capture.

VESTA includes two authoring components, one for documentation and one for unit tests (see Figure 4.1). In the remainder of this section, we will first present design challenges we were confronted with when exploring the design of VESTA. Then, we will briefly describe the key technical problems we encountered when implementing the prototype and how we solved them.

Vesta contains separate authoring components for documentation and unit tests.

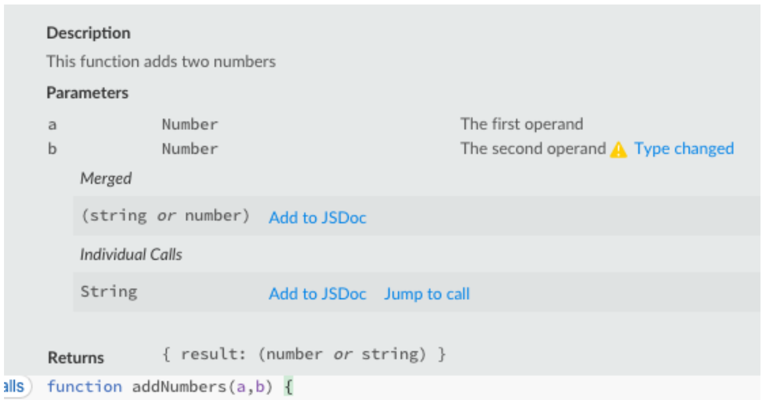
4.2.1 Design

When exploring how to integrate runtime information from manual tests of the application into the development process, we found four design challenges that needed to be solved:

1. Usually, not all possible paths through the code are executed at runtime. Hence, runtime information obtained from an arbitrary execution is likely **incomplete**.
2. Because runtime traces are only captured from manual executions that are performed by the developer, information can become **outdated**.
3. The code that developers test manually is often incorrect and unfinished. In this case, expectations about the code that are inferred from a runtime trace are likely also **incorrect**.
4. Software changes throughout the development, causing information to be **impermanent**, i.e., information that was correct at one point is not necessarily correct for all future versions of the software.

Interaction Design: Authoring Documentation and Unit Tests in the Edit-Test-Edit Cycle

To solve the first two problems, we designed the interaction with VESTA to encourage writing documentation and unit tests continuously while working on the source code.



```
Description
This function adds two numbers
Parameters
a          Number                              The first operand
b          Number                              The second operand ⚠ Type changed
   Merged
   (string or number)   Add to JSDoc
   Individual Calls
   String               Add to JSDoc   Jump to call

Returns    { result: (number or string) }
alls)  function addNumbers(a,b) {
```

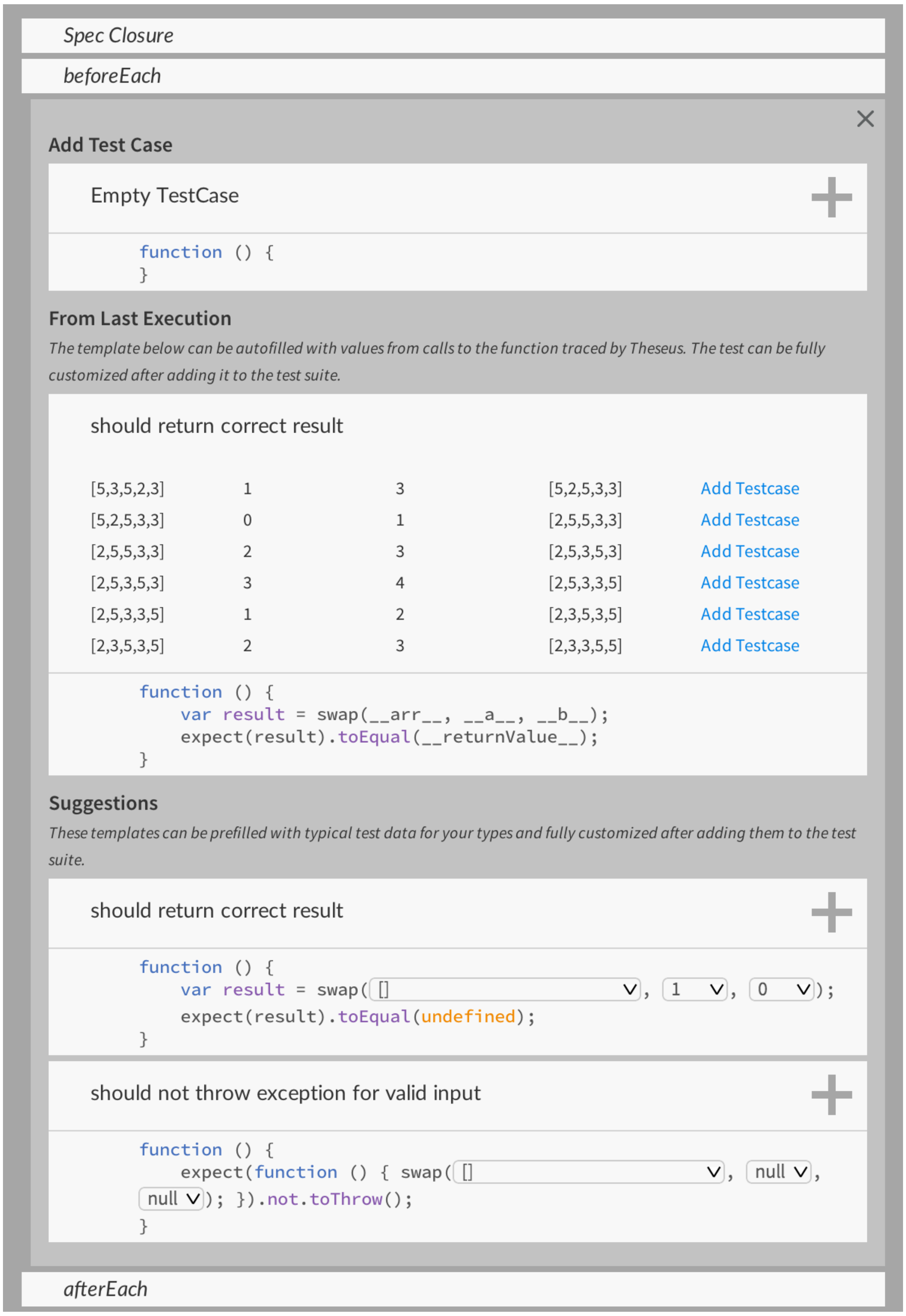
Figure 4.2: Vesta detected that type information recorded at runtime does not match the documentation. It allows to navigate to the responsible call site, or to update the documentation. Figure Source: [Krämer et al., 2016a]

Writing documentation and unit tests as part of the edit-test-edit cycle solves problem 1 and 2.

Ideally, developers maintain these documents as part of the rapid edit-test-edit cycles they already perform routinely. To encourage this behavior, VESTA suggests likely updates to documentation and unit tests immediately after each manual execution. The manual execution exercised the same code that is about to be documented, hence, the information presented in VESTA is likely relevant even though it is **incomplete**. Information is also not **outdated**, because the developer has just performed the execution.

Vesta checks the documentation after every manual execution.

To implement the workflow as described above for documentation, VESTA compares the information obtained at runtime, i.e., types of function parameters and return values, with the current documentation. If a function is not yet documented, VESTA inserts the recorded types into the documentation automatically. Otherwise, VESTA shows type conflicts in case the types recorded at runtime and the documented types do not match (see Figure 4.2). VESTA supports two ways to resolve the type conflict: First, if the conflict may constitute a bug, VESTA allows developers to navigate to the erroneous call site to fix the bug. Second, developers can instruct VESTA to update the documentation by either replacing the current documentation to reflect the newly observed types, or by merging the new and the existing type information to indicate polymorphism.



Add Test Case

Empty TestCase

```
function () {
}
```

From Last Execution

The template below can be autofilled with values from calls to the function traced by Theseus. The test can be fully customized after adding it to the test suite.

should return correct result

[5,3,5,2,3]	1	3	[5,2,5,3,3]	Add Testcase
[5,2,5,3,3]	0	1	[2,5,5,3,3]	Add Testcase
[2,5,5,3,3]	2	3	[2,5,3,5,3]	Add Testcase
[2,5,3,5,3]	3	4	[2,5,3,3,5]	Add Testcase
[2,5,3,3,5]	1	2	[2,3,5,3,5]	Add Testcase
[2,3,5,3,5]	2	3	[2,3,3,5,5]	Add Testcase

```
function () {
    var result = swap(__arr__, __a__, __b__);
    expect(result).toEqual(__returnValue__);
}
```

Suggestions

These templates can be prefilled with typical test data for your types and fully customized after adding them to the test suite.

should return correct result

```
function () {
    var result = swap([], 1, 0);
    expect(result).toEqual(undefined);
}
```

should not throw exception for valid input

```
function () {
    expect(function () { swap([], null,
null); }).not.toThrow();
}
```

Figure 4.3: Vesta offers two test templates that can be instantiated with runtime information. The first template recreates a recorded method invocation and tests if it returns the recorded result. The second template helps create tests for common edge cases based on the documented types of function parameters.

Unit tests are created from templates that are pre-populated with runtime information.

While documentation is inserted automatically if no previous documentation exists, unit tests are not created without user interaction. Instead, VESTA supports the user in assembling test suites by offering two test templates that can be pre-populated with runtime information (see Figure 4.3). The first available template recreates a previous invocation of a function. The resulting test calls the tested function with the previously recorded parameters and tests if it returns the recorded result. Using this template alone, developers can already capture their manual tests in persistent, re-executable unit tests. The test coverage that can be achieved using only tests generated from this template correlates with how thoroughly developers test their application using manual executions.

Common edge cases to be checked in unit tests are inferred from the documented types.

A second test template helps developers to create tests for common edge cases, based on the types of function parameters. This template uses the type information that is stored in the function documentation. For example, if a function expects a string parameter, the test template would suggest to test the function behavior in case the parameter is an empty string or `undefined`. The list of suggested values for each type is currently hardcoded in VESTA. This template should encourage developers to write more tests that cover erroneous behavior.

Developers can customize all tests created with VESTA.

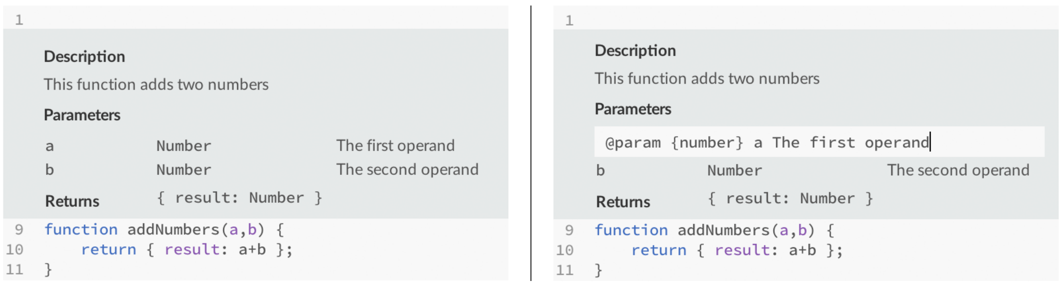
All tests created with VESTA can be modified, and users can create their own custom tests. The current implementation of our prototype is designed for the Jasmine[6] testing framework. Hence, VESTA's test templates use the Jasmine API, and users can use all Jasmine features when creating or modifying tests.

Interface Design: Different Representations for Reading and Editing

Following two design guidelines can solve problem 3 and 4.

The remaining problems, potentially **incorrect** and **impermanent** information, can be solved by following two design guidelines: First, developers need to judge the correctness of information quickly, to deal with potentially incor-

[6]http://jasmine.github.io/


Description
This function adds two numbers
Parameters

a	Number	The first operand
b	Number	The second operand
Returns	{ result: Number }	

```
function addNumbers(a,b) {
    return { result: a+b };
}
```

Description
This function adds two numbers
Parameters

```
@param {number} a The first operand
```

b	Number	The second operand
Returns	{ result: Number }	

```
function addNumbers(a,b) {
    return { result: a+b };
}
```

Figure 4.4: Vesta switches between two representations of documentation as needed. For reading, documentation is formatted in an easy to parse, non-code format (left). For editing, it shows a textual representation of the documentation formatted in JSDoc (right).

rect information. Thus, information needs to be presented in a way that is easy to parse but does not draw attention after every manual execution, because they often occur during debugging phases that are already cognitively demanding. Second, to deal with impermanent information, developers need to be able to change or extend information quickly. To implement these guidelines, VESTA makes use of secondary notation (see Chapter 2.3.2), i.e., visually rich formatting.

VESTA's documentation component switches between two representations for reading or writing.

To implement these guidelines, VESTA's documentation component switches between two modes (see Figure 4.4): For reading, documentation is formatted in an easy to parse, visually appealing, non-code format. For editing, VESTA shows a textual representation of individual documentation lines formatted in JSDoc, a markup language for documentation in JavaScript. Additionally, Markdown[7], a popular markup language for text formatting, can be used to format the documentation for the reading mode. VESTA switches between both representations automatically, when developers move the cursor from the source code into the documentation. Developers can rely on standard cursor navigation using either the keyboard or the mouse as they are used to from regular source code editing.

VESTA hides the file-based storage of unit tests.

Unit tests are commonly organized in *test suites*, which bundle a set of related unit tests. Groups of related test suites are usually stored in one source file. VESTA hides the file-

[7]https://daringfireball.net/projects/markdown/

based storage from the user and instead calls a group of related test suites a *test collection*. To provide a starting point for test organization, VESTA maintains one test collection associated with each source file in the project, and one test suite associated with each function.

VESTA uses secondary notation to improve the readability of unit tests.

VESTA's unit test component is shown in a separate area on the right side of the source code (see Figure 4.1). It always shows one test suite at a time. By default, this is the associated test suite for the currently edited function. At the top, the user can select the displayed test collection and test suite using drop-down menus. Individual tests are shown in boxes that include a header to show the test name and failure status. The source code of each test can be collapsed (as is the case for the test shown in Figure 4.1), which allows to read the test suite as a concise list of expectations and their failure status. Similar as for documentation, keyboard-based cursor navigation is possible in the entire unit test component.

Tests can be executed from within VESTA.

VESTA also introduces a toolbar button that allows to run all unit tests. When running unit tests from within VESTA, a runtime trace is automatically recorded to provide additional information about the tests. In particular, VESTA recognizes which functions are called at some point during the execution of each test. This allows developers to see all unit tests from which the currently edited function is called at some point. Given that every test case specifies a behavior of the application, this yields a quick overview of the different behaviors or cross-cutting concerns a single function is involved in. Of course, VESTA also shows all errors recorded during the test execution.

4.2.2 Implementation

VESTA is implemented as a plugin for Brackets.

VESTA is implemented as an extension of Brackets, an open-source source code editor. For runtime tracing, VESTA uses a modified version of the Theseus [Lieber et al., 2014] runtime tracer.

VESTA stores all information transparently: Unit tests are stored as source files and documentation is stored in documentation blocks inside the edited source file. To uniquely identify a function even after it was renamed or moved, Vesta stores a `uniqueFunctionIdentifier` in the documentation block for each function. Theseus was changed to also read these function identifiers when recording a runtime trace.

All information is stored transparently in source files.

To manage information internally, VESTA represents function documentation, type information, and unit tests using an object-oriented data model. This model can be rendered into different representations as needed: To source code for editing and saving, or to VESTA'S rich HTML-based visualizations for viewing.

Internally, VESTA uses an object-oriented data model.

Our prototype is available online as an open-source project.

http://hci.rwth-aachen.de/vesta

4.3 How Developers use VESTA

VESTA was created to explore if we can encourage documentation and unit test authoring by designing an interaction around current development practice. In this section, we will present a between-groups lab study we performed in order to understand how developers use VESTA. We expect that developers using VESTA write more documentation and unit tests and in addition, the quality of these documents improves. To assess the quality of documentation and unit test, we used the following metrics:

We expect developers using Vesta to write more and better documentation and unit tests.

For documentation:

1. amount of created documentation
2. accuracy of documentation
3. completeness of documentation

For unit tests:

1. number of test cases
2. number of test cases testing failure cases
3. source code lines covered by all tests

Further, we assumed that developers would not need to radically change their strategies to incorporate VESTA into their workflow.

4.3.1 Setup

We used a between-groups study design with two conditions.

We randomly assigned participants to one of these two conditions: In the VESTA condition, participants used Brackets and the VESTA extension. In the control condition, participants used Brackets and the FuncDocr[8] extension, which analyzes function headers to provide a documentation skeleton that includes markup for all required statements to document parameter types and the return type. Features matching those of FuncDocr are available in many IDEs to support the creation of function documentation.

Participants worked on a large open-ended coding task.

We designed our study around a large open-ended task that could give us rich and ecologically valid qualitative results, even though this could come at the expense of statistical significance in low-level quantitative measures [Seaman, 1999]. When researching the effect of development tools on developer productivity, effects can be over-exaggerated if the tasks used in a study are too simple [De Alwis et al., 2007]. In our experiment, participants implemented a server application that fetches and parses the menus of restaurants on a college campus from the official college website, and provides a web-based API to access these menus. We allowed participants to work on this tasks for up to 6 hours.

To keep solutions comparable, we restricted the use of third-party libraries.

To implement the server, participants had to use Node.js[9]. A large number of libraries for Node.js is available that provide useful abstractions for parts of the task. To keep solu-

[8]https://github.com/Wikunia/brackets-FuncDocr
[9]https://nodejs.org/

tions comparable, we selected a set of libraries that participants were allowed to use:

Express[10] Express provides basic routing for server applications, i.e., it allows to attach functions to individual HTTP requests.

lodash[11] Lodash is a widely used utility library to manipulate arrays and objects.

cheerio[12] Cheerio provides functions for parsing HTML that are similar to those provided by jQuery, which is widely used for HTML parsing and manipulation on websites. jQuery itself cannot be used in Node.js, because it relies on a web browser to provide functions for standard DOM manipulation.

All participants used identical hardware with all required tools preinstalled.

Participants used 27-inch iMacs on which we preinstalled all libraries, required software, and a minimal project template. The experimenter was present at all times as a silent observer, and we recorded the participants' screens throughout the study. In each session, three to four participants worked in parallel. Participants were allowed to talk to share ideas and help each other out. This not only made the study more manageable, but also made the scenario more realistic, because interruptions from co-workers were found to be common to regular software developers [González et al., 2004]. Due to the length of the study, we included a lunch break. We compensated participants for their expenditure of time with a 100€ gift certificate.

Participants filled out a questionnaire before and after the study.

To assess the participants' experience, we asked them to fill out a questionnaire about their current documentation and unit testing practices before the trial. We informally interviewed participants during the lunch break and after the study about their problems and their opinions about our prototype. After the study, we asked participants in the VESTA condition to fill out an additional questionnaire about the tool. All questionnaire questions were answered using a 5-point Likert scale, where 1 corresponds to "strongly disagree" and 5 corresponds to "strongly agree".

4.3.2 Results

All participants were students with prior experience in JavaScript.

We recruited 14 participants (12 males, 2 female) for our study, one of which was excluded from the analysis due to technical problems during the trial. This left us with 7 participants in the VESTA condition, and 6 participants in the control condition. Participants were 24 years old on average ($SD = 2.6$). All participants were computer science students. Participants reported to have an average of 3.0 years ($SD = 1.8$) experience with JavaScript, and to spent 13.1 hours/week ($SD = 7.3$) programming.

Participants report value documentation and unit tests but rarely write them.

Analyzing responses to the pre-study questionnaire, we found that participants were not consistently required to write documentation ($Mdn = 3$) or unit tests ($Mdn = 2$) for their regular programming work. Most agreed to know how good tests should look like ($Mdn = 4$). Participants very consistently disagreed that it is not worth it to write tests ($Mdn = 1$) or documentation ($Mdn = 1$), but still admitted to skip writing these documents ($Mdn = 4$ for both). Consequently, most participants found that they should write more documentation ($Mdn = 4$) and unit tests ($Mdn = 4$).

Documentation

The number of created documentation lines is similar in both conditions.

We analyzed the quality of documentation using the metrics listed before. To quantify the amount of created documentation, we used the number of documentation lines per function. In both conditions, we counted the lines of documentation when formatted using JSDoc. We found no significant effect of condition on the number of created documentation lines (one-sided t-test $p = 0.330$, $t(8.8) = 0.45$).

Documentation created using VESTA is nearly always correct.

To quantify the accuracy of the created documentation, we analyzed the individual type specifications contained in the documentation. We considered a type to be documented accurately, if it was syntactically and semantically correct, i.e., if it conformed to the rules according to the task description. VESTA is designed to perform well under this metric, as the automatically created type specifications are

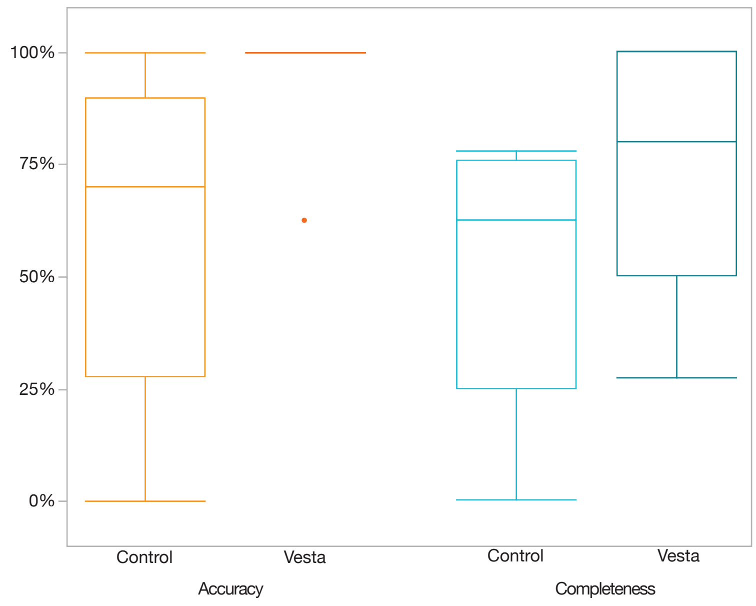


Figure 4.5: Type specifications created with VESTA were nearly always accurate, because VESTA creates them automatically, while about one in three types documented without VESTA was inaccurate. Developers using VESTA also created more complete documentation. None of the differences is statistically significant.

always accurate. A type specification can become inaccurate, only if a user manually adjusts a type, e.g., to generalize it to allow more types, and then does not execute the function again in a manual test. This happened for one user in our study, because time ran out. Hence, we ended up with three inaccurate type specifications (created by the same user) in the VESTA condition, while 38.9% ($SD = 37.2\%$) of type specifications in the control condition were inaccurate (see Figure 4.5). Only one user in the control condition documented all types accurately. The effect of condition on the percentage of inaccurate types is close to being significant (one-sided t-test $p = 0.072$, $t(6.2) = -1.67$).

We defined a function to be completely documented, if type specifications were present for every parameter and the return value (if applicable), and a description was present

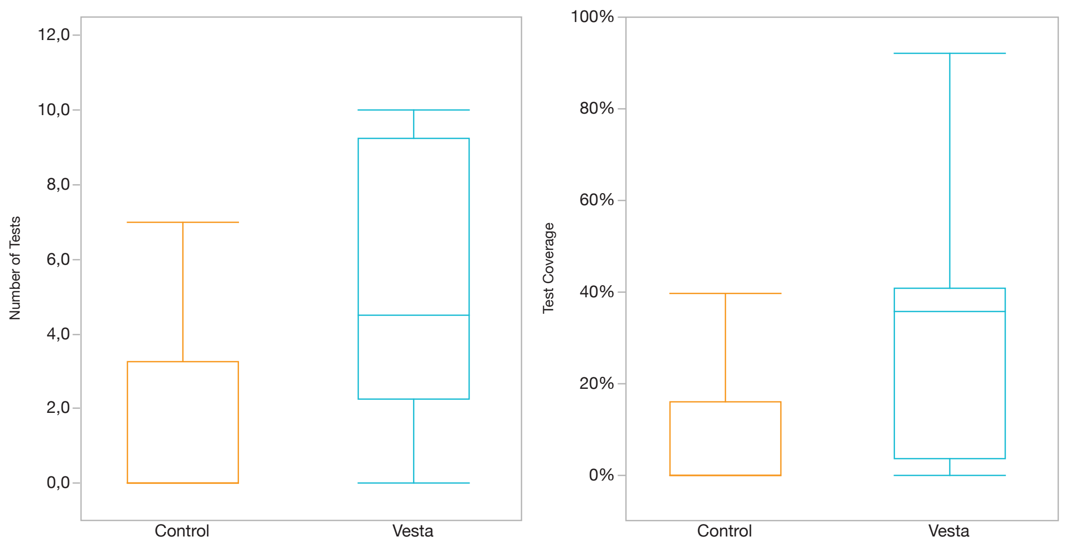


Figure 4.6: The graphs show the number of tests created by participants in each condition, and the percentage of statements executed when running all tests. Four participants in the control condition and one participant in the VESTA condition wrote no tests at all.

Documentation created with VESTA was slightly more complete.

for every parameter and the complete function. This metric does not favor VESTA, because descriptions have to be created manually in both conditions. Participants in the control condition documented 51.9% ($SD = 30.9\%$) of all functions completely, compared to 68.4% ($SD = 32.7\%$) in the VESTA condition (see Figure 4.5). Again, this difference is not statistically significant (one-sided t-test $p = 0.093$, $t(10.3) = 1.42$).

Unit Tests

In the control condition more than half of all participants did not write unit tests at all.

For unit tests, we again started by comparing the amount of unit tests created in each condition (see Figure 4.6). Four of six participants in the control condition and one participant in the VESTA condition wrote no tests at all. The effect of condition on the number of tests is marginal significant (one-sided t-test, $p = 0.056$, $t(11.0) = 1.73$).

Only one participant in each condition tested a failure case, e.g., the behavior of a function called with invalid parameters or in case of network failures. Four of these tests were created in the VESTA condition compared to two in the control condition. Because only a single participant in each condition created tests for failure cases at all, we performed no statistical tests.

Only one participant per condition tested for failure cases.

We analyzed the code coverage of each test suite by measuring the percentage of statements exercised when running all tests (see Figure 4.6). As a result of the low number of test cases, the percentage of source code lines covered by the test suites is mediocre in both conditions as well. The effect of condition on the percentage of statements exercised by the complete test suite is marginal significant (one-sided t-test, $p = 0.060$, $t(9.1) = 1.72$).

Participants using VESTA achieved higher test coverage.

4.3.3 Discussion

VESTA encouraged developers to create more documentation and unit tests. However, while we found substantial improvements for documentation, the amount of unit tests created was little in both conditions. To find reasons for this result, we will discuss in this section how developers adopted each component.

Documentation

The design idea we explored with VESTA is to encourage documentation and unit test authoring by integrating these activities into the existing workflow of developers. To implement a seamless integration, we collect runtime information from manual tests performed by the developer, and use this information to support the authoring process. For documentation authoring, this approach yielded promising results: The documentation was more accurate and more complete when it was authored using VESTA. The developers strongly agreed ($Mdn = 5$) that VESTA made authoring documentation more enjoyable.

Our interaction design improved documentation.

When analyzing the type information developers recorded, we found that they usually assumed a fixed type and did

Developers usually assumed fixed types.

not actually use dynamic typing. Storing type information in the documentation instead of introducing strong typing as a language feature matched the developers' requirements: Type information can evolve over time, and multiple types can be specified if polymorphism is allowed explicitly. We also confirmed that the type information recorded using VESTA was sufficient to transition to more advanced static analyzing tools, such as Tern[13].

VESTA solved three problems typically leading to incomplete documentation.

By observing participants during the trial and reviewing screen recordings after the trial, we found that the reason for increased accuracy and completeness of documentation when using VESTA is its interaction design. We found that VESTA provided support for three common problems developers in both conditions have faced:

1. Developers forgot to update documentation whenever code evolved. VESTA showed a type mismatch when the documentation needed to be updated; that served as a reminder for developers. Without VESTA some developers tried to avoid dealing with updates to the documentation throughout the development process and instead wrote all documentation at the end of trial. This often led to inaccuracies when documenting methods that were implemented a while ago.

2. Developers neglected to create any documentation or to include all required parts. When using VESTA, developers regularly checked the automatically created type information to check if their code was correct, i.e., if the created type descriptions match their expectations. When performing this check, developers opportunistically added the required descriptions. Further, because developers have just experienced the value of documentation, they are more willing to put effort into completing it.

3. Developers had insufficient knowledge about third-party APIs, e.g., they confused whether a DOM manipulation method returned an array or a single DOM element. The type information generated by VESTA

[13]http://ternjs.net/

showed developers the correct type and, hence, prevented numerous programming errors.

VESTA increased the near-term value of documentation.

These observations, especially the latter two, show that developers turned to the information in VESTA not in order to author documentation but because it was useful for their current task at hand, i.e., creating working software. Developers regularly identified bugs by inspecting the information or warnings shown in VESTA. The interaction design we proposed for authoring documentation was successful, because it enabled additional uses of runtime information besides authoring documentation. We conclude that an effective strategy to motivate developers to pay more attention to the creation of documentation is to provide them with more near-term rewards for this task, i.e., support during their actual programming task.

Unit Tests

No developer created sufficient unit tests.

All developers in our study strikingly under-prioritized unit tests authoring. This matches their self-assessment in the pre-study questionnaire. Surprisingly, several participants in the control condition did not write a single unit test, even though the task description clearly asked to prioritize proper testing and documentation of partial results over finishing the task.

The cost-benefit-tradeoff for unit tests is perceived to be worse than for documentation.

For documentation, we found that increasing its near-term value was important to encourage developers to create it. We believe that the near-term value of unit tests was unclear for the participants in our study. Further, unit tests are harder to create than documentation, because they necessitate to write executable source code using a specific unit test framework. Once unit tests require mocking objects or test asynchronous methods, the source code of the unit test can become quite complex and creating it becomes cognitively demanding. We assume that developers consider the cost-benefit-tradeoff for creating unit tests to be worse than for creating documentation, where they only need to adhere to a simple syntax.

VESTA succeeded in simplifying the creation of unit tests.

VESTA is designed to improve the cost-benefit-tradeoff for creating unit tests. When relying on test templates to create new tests, developers do not need to know how to use the unit test API. VESTA also automatically provides parameter and return values to use in the test, hence, the developer does not need to manually create examples for complex input data, such as HTML strings. These improvements in the authoring process encouraged all but one participants in the VESTA condition to create at least some unit tests. This is a promising result which indicates that the use of runtime information we propose is useful for authoring unit tests.

The abstracted organization of unit tests was hard to grasp.

The tests created in the VESTA condition were still not sufficiently thorough. One reason for this is that developers rarely test failure conditions when manually executing their application. This is likely to translate into test suites that also lack these tests, because the majority of tests created in VESTA use the template that recreates a manual execution. Another reason is that users found VESTA's organization concept for unit tests, which hides files and instead uses dropdown menus, to be cumbersome to use. Many participants agreed ($Mdn = 4$) that they would write better tests without this system. We conclude that, while the interaction design of VESTA's unit test component is promising, a future iteration of the tools should explore alternative interfaces to organize tests.

4.4 Conclusion and Future Work

Our key idea is to use runtime traces to integrate documentation and unit test authoring into the edit-test-edit cycle.

In this chapter, we explored how we can encourage developers to author more documentation and unit tests. To this end, we presented an interaction design for authoring documentation and unit tests that integrates into current development workflows. Our key idea is to use runtime information to provide updates to these documents immediately after each manual execution, which is already performed regularly developers. We implemented this interaction design in a prototype called VESTA.

In studying this prototype, we found that it is crucial to design the tool in such a way that it provides near-term value for the developer. For example, VESTA's authoring interface for documentation was very successful, because it allowed developers not only to author documentation more efficiently but also to check the correctness of their source code. It successfully increased the near-term value of documentation enough to encourage developers to properly maintain it. VESTA's unit test authoring tool also provided useful assistance to create unit tests, but, on the other hand, could not increase their near-term value for the developer enough to encourage them to create more tests.

It is crucial to provide near-term value for the developer.

To expand VESTA's existing strengths, we propose to integrate a static analysis tool that uses the information stored in the documentation. This tool would be automatically bootstrapped using runtime information, to allow developers to experience the benefits of static type checking without needing to set it up first. This could further extend the near-term value of documentation. More advanced methods to generate unit tests should be added, as we found that developers highly regarded the option to use template-based unit tests suggestions to assemble their test suites.

More advanced static and dynamic analysis techniques could extend VESTA in the future.

To increase the near-term value of unit tests, we propose to add continuous testing to VESTA. This would provide immediate feedback about program errors, similar to the automatic checks performed on documentation after manual executions. Additionally, we propose to study the effect of a system that further lowers the barrier to creating unit tests. For example, we envision a one-button interface to specify whether the last manual execution was erroneous or successful. With this information, the system could generate a characterization test [Feathers, 2004] without requiring further interaction. We prototyped this interaction for a live coding environment [Ulmen, 2014], but an evaluation of this idea is still in need of future work.

Continuous testing and even simpler generation of tests could increase the near-term value of unit tests.

Lastly, we recommend to evaluate the interaction design we propose in a field study over a longer period of time in a real development project. This study should analyze if VESTA's positive effects diminish after the novelty effect wears off. While logistically challenging, we believe that

A longer evaluation can test if the positive effects remain once the novelty wears off.

such a study could reveal interesting insights in the way real developers approach documentation and unit test authoring.

Chapter 5

Call Graph Navigation

"Habitability is the characteristic of source code that enables programmers coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently. [. . .] What is important is that it be easy for programmers to come up to speed with the code, to be able to navigate through it effectively, to be able to understand what changes to make, and to be able to make them safely and correctly."

—Richard P. Gabriel, Patterns of Software

Publications: The work in this chapter was done in collaboration with Björn Hartmann, Thorsten Karrer, Jonathan Diehl, Moritz Wittenhagen, and Joachim Kurz. Initial results on Stacksplorer were first published as a poster at UIST 2010 [Krämer et al., 2010] and later as a full paper at UIST 2011 [Karrer et al., 2011]. Blaze was presented in 2012 as a poster at CHI [Krämer et al., 2012b] and as a demo at ICSE [Krämer et al., 2012a]. The complete analysis of call graph navigation tools presented here is based on a full paper published at CHI 2013 [Krämer et al., 2013]. The author of this thesis implemented and evaluated Stacksplorer as part of his Diploma thesis [Krämer, 2011] under supervision of Björn Hartmann, Thorsten Karrer, and Jonathan Diehl. Blaze was implemented and evaluated as part of the Bachelor's thesis by Joachim Kurz [Kurz, 2011], under supervision of the author. The author of this thesis contributed to the design of all tools, the study design, the development of the analysis methodology, and the analysis of all results.

Navigation is a time-consuming activity in object-oriented source code.

Source code is different from natural language text in that it is often not read sequentially. Instead, the source code often includes cross-cutting concerns, i.e., a single programming plan whose implementation is delocalized and spread across different locations. Developers then need to perform navigation between different parts of the source code. In Chapter 2.1.1 we learned that delocalized programming plans are especially common in object-oriented source code. Unfortunately, current navigation tools were found to provide inadequate support for comprehending cross-cutting concerns: Ko et al. [2005] studied developers working on a maintenance task in an object-oriented project and found that they spent on average 35% of their time on the mechanics of navigation alone, which is more than they spent on actually reading code.

Object-orientation is one of the most widely used programming paradigms today.

Object-oriented programming is one of the most widely used programming paradigms today. Nine of the top ten languages on the TIOBE language index[1], an index rating the popularity of programming languages, support object-oriented programming as a language feature. In consequence, there is a huge potential benefit in improving the navigation tools for object-oriented source code.

To work with object-oriented software, developers need to comprehend the object structure and the procedural structure.

The structure of object-oriented software can help to understand which types of navigation are required in object-oriented source code. Every object-oriented software is composed of two orthogonal dimensions [Détienne, 2002]: The object structure describes how data attributes and associated functions are organized in classes, and how classes are organized in a hierarchy formed through inheritance. The procedural structure describes how data and functions of various objects are used together to accomplish a given goal. It is usually formed by message-passing between objects, i.e., method calls. The hierarchy of method calls in an application can be represented by the call graph, a graph that represents each method as a node and represents method calls as edges. We have discussed the consequences these two orthogonal dimensions have for program comprehension in Chapter 2: Because the procedural structure spans various objects, the source code implementing one concern is often delocalized. Hence, navigation along the

[1]http://www.tiobe.com/tiobe_index

call graph becomes a crucial activity for developers understanding object-oriented software projects.

The semantic structure of object-oriented source code only represents the object structure.

On the one hand, the semantic structure of object-oriented source code usually aligns well with the object structure. Each class is implemented in a separate file, i.e., developers can navigate inside a class by scrolling and between classes by switching files. On the other hand, to navigate the call graph developers need to find out from where methods are called or where a called method is implemented. Call graph navigation tools aim to support this task. Ko et al. [2005] found that existing call graph navigation tools in the Eclipse IDE[2] are rarely used, because they require complex setup. In addition, they make it difficult for developers to relate two methods that are connected in the call graph.

In this project, we wanted to explore how an improved interaction design can help developers navigate the call graph more effectively. We suspected that two design principles can mitigate the problems of existing tools: *Proactive information visualization* means that the tools present navigation targets without prior setup, and *comprehensible relevance* means that the navigation targets presented in the tool can be easily related to the code currently being inspected by the developer. Over the course of this research project, we created two call graph navigation tools, Stacksplorer and Blaze. Both tools implement the principles above. Blaze, additionally, implements two dedicated modes, one for the Search phase and one for the Relate and Collect phases in the three-phase navigation model (see Chapter 2). We will compare these tools in a laboratory study to two comparative conditions: First, as a baseline, to an IDE without any dedicated call graph exploration tool, and second, to the Call Hierarchy, a call graph navigation tool that is common in many existing IDEs. Our comparison will reveal that while all tools led to better task success rates, only Stacksplorer and Blaze managed to also reduce task completion times. Finally, we will introduce an analysis framework to compare inter-method navigation behavior. Using this framework, we found that the design principles implemented by Stacksplorer and Blaze cause developers adopt a more effective navigation behavior. We

[2]https://eclipse.org/ide/

will start the chapter by presenting closely related research projects.

5.1 Existing Tools for Call Graph Navigation

Method calls in object-oriented software can be formalized as the call graph.

In an object-oriented programming language, the procedural structure is formed by message passing, i.e., method calls between different objects that accomplish a goal. The method calls in a program can be formalized as the call graph $G = (V, E)$, with $V = \{m_1, m_2, ..., m_n\}$ being the set of methods in the program and $E = \{(v, u) \mid v, u \in V, u \text{ is called from the implementation of } v\}$. Navigation tools for the procedural structure usually facilitate the exploration of a subgraph of the call graph.

Commonly, IDEs allow to navigate from a method call to the definition of the called method.

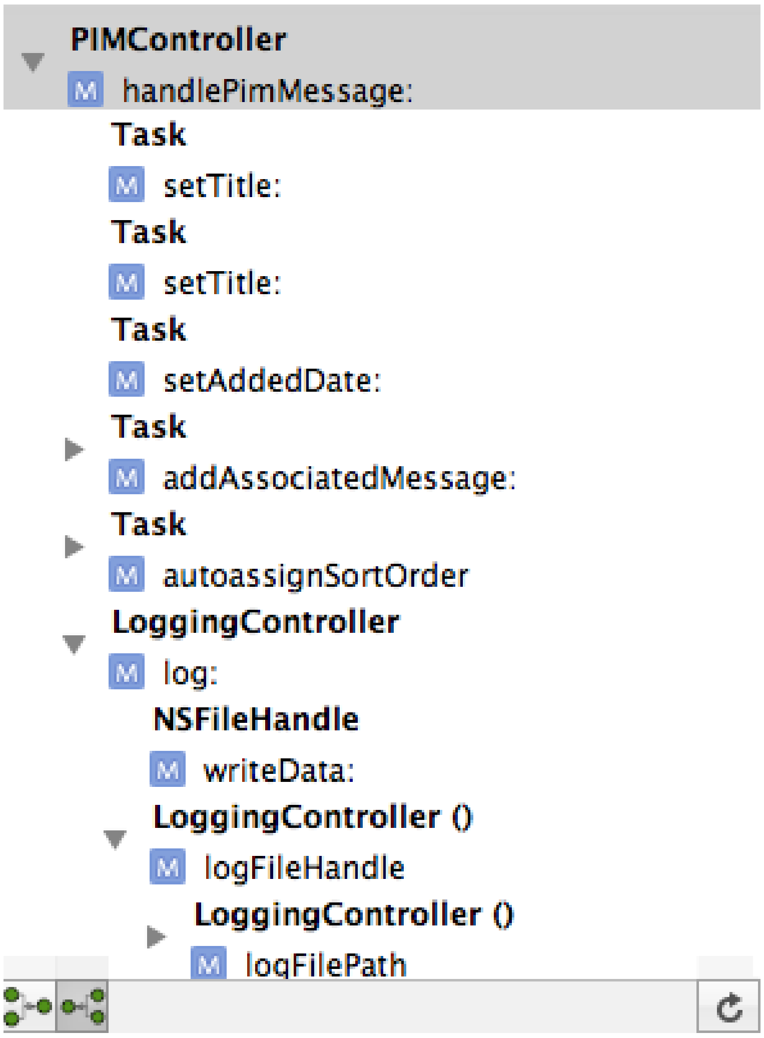
The most commonly available method for call graph navigation in current IDEs is to jump from a method call to the definition of this method, i.e., navigate the call graph in the direction of the edges, one edge at a time. This feature is usually invoked from the source code editor, either using a shortcut or a context menu.

The call hierarchy shows all method calls originating from or leading to one method.

Several IDEs, such as Eclipse and Visual Studio, offer a Call Hierarchy tool (see Figure 5.1), which allows to select one method as a starting point and to browse all paths in the call graph that include the starting point. Usually navigation is performed in a tree view that can be configured to show either paths originating from the starting point or paths leading to the starting point.

Advanced call graph navigation tools are often not used.

More advanced navigation tools, if they exist in an IDE, are often not widely used. Ko et al. [2006] observed that few developers working on a small (500LOC) Java application using the Eclipse IDE employed the available tools for call graph navigation more than once. Instead, they resorted to the package browser and the search tool, both of which are very ineffective in answering questions about the call graph. Similar results are also found when analyzing Eclipse usage data obtained from real world developers



PIMController
handlePimMessage:
Task
setTitle:
Task
setTitle:
Task
setAddedDate:
Task
addAssociatedMessage:
Task
autoassignSortOrder
LoggingController
log:
NSFileHandle
writeData:
LoggingController ()
logFileHandle
LoggingController ()
logFilePath

Figure 5.1: A Call Hierarchy tool allows to pick one method as the root of a tree view and browse the hierarchy of callers or callees of the root method. The figure shows our implementation. Figure adapted from [Krämer et al., 2013]

[Murphy et al., 2006]. We suspect that existing tools are not well aligned with the developers' strategies for code comprehension.

Research has tackled this problem by building tools in two categories: Tools in the first category try to provide useful information in the context of a task automatically. Tools in the second category implement more sophisticated search features to find information in the call graph.

5.1.1 Recommender Tools

Recommender tools try to predict the working set.

In Chapter 2 we introduced the two phase navigation model by Ko et al. [2006]. It shows that by navigating, developers try to collect all information that is required for their task. This set of information is called the *working set*. *Recommender systems* try to estimate the working set automatically. They calculate a *degree of interest* (DOI) for every entity (depending on the systems this can be files, classes, methods, or more) in the project, which is then used to filter the entities shown in other tools of the IDE, e.g., the package browser, such that only entities are shown where the DOI exceeds a threshold. Most recommender systems use an algorithm based on the concept of *computational wear* [Hill et al., 1992] to determine the DOI for each entity. Computational wear represents information about the history of an entity, i.e., when it was authored, when it was edited, and finally, when it was read. To calculate the DOI of an entity in a software project, tools have analyzed a variety of factors, i.e., when the developer looked at the entity, when the developer edited the entity, the history of edits by all developers, and the information provided about the entity in version control systems [Čubranić et al., 2003; R DeLine et al., 2005; Kersten et al., 2005; Singer et al., 2005].

Recommender tools only reduce the number of navigation targets in existing tools.

In lab studies, all of these tools caused a significant reduction of navigation effort. They also allowed participants to orient themselves faster in unknown projects. Recommender tools, however, leave developers with the same navigation tools and only reduce the amount of navigation targets.

5.1.2 Search Tools

The second category of tools we would like to discuss aims to simplify navigation by allowing developers to search for relevant information more effectively. REACHER [LaToza et al., 2010c] is a tool specifically designed to answer questions regarding the procedural structure of an application. Searches can include search strings but also restrictions on the parts of the call graph that should be considered. For example, a developer could search for a method that includes the term "PDF" and occurs downstream from a method called "export". REACHER automatically removes paths from the search that are infeasible, i.e., which can never be executed. To present search results, REACHER uses a visual representation of the relevant portion of the call graph. A lab study showed that REACHER significantly improves answering times for reachability questions, but due to the rather complex visualization used, it does require some training.

REACHER offers semantic search across paths in the call graph.

JQuery [Janzen et al., 2003] uses a logical programming language to specify queries allowing even more specific searches. In contrast to REACHER, it is not limited to searches in the call graph, but can also inspect the object hierarchy. Results are visualized in a hierarchical browser, similar to Eclipse's Package Browser. To refine the results, every entry in the browser can again be queried using JQuery. Similar to REACHER, JQuery suffered from complexity and a steep learning curve, causing many developers to use extremely simplistic queries only.

JQuery is a semantic query language to search in both the procedural and the object hierarchy.

The tools presented so far assume that developers comprehend the source code exclusively by reading and navigating through it. However, it is not uncommon for developers to manually execute the application to understand what it does. The Whyline [Ko et al., 2008] supports this behavior and allows developers to ask questions about the visual output of an application. The Whyline will then show the relevant methods that contributed to the selected output property. Because the Whyline relies on the application being executed, it can filter the call graph based on which methods were actually called during execution. Exploiting runtime information to identify relevant information was shown to substantially increase maintenance performance. However, the Whyline is only applicable for specific problems: The application needs to have a graphical user interface, the defect needs to manifest in the graphical output,

The Whyline analyzes which methods were actually called during an execution of the program.

and the defect must not prevent the application from compiling.

Little is known about the effectiveness of these tools for real-world development tasks.

The tools presented differ in the part of the call graph they make available for navigation, in their design, i.e., the visualization used to show the relevant part of the call graph, and in the interaction used to navigate or refine a search. Unfortunately, little is known about the potential adoption of these tools in real development scenarios. One study comparing JQuery to two recommender systems found that in real world development tasks the effect of inter-subject differences completely overshadowed the effect of any tool [De Alwis et al., 2007].

Canvas-based IDEs allow to represent both the object structure and the procedural structure visually.

Code Bubbles [Bragdon et al., 2010] and Code Canvas [Robert DeLine et al., 2010] introduced IDE concepts that allow developers to lay out individual parts of the source code, e.g., methods, in bubbles on a 2D canvas. The parts can be connected to indicate relationships between two parts, such as a call graph edge. This concept allows to lay out a complete delocalized programming plan so that all parts are visible simultaneously. Because both tools allow for unconstraint layout of bubbles, they might suffer from some disadvantages of visual programming approaches we have discussed before (see Chapter 2). An experimental comparison between our tools and canvas-based programming tools is an interesting endeavor for future work.

5.2 Stacksplorer and Blaze

Over the course of this research project, we designed two call graph exploration tools: Stacksplorer and Blaze. Our design is built around two key concepts:

Proactive Information Visualization Our tools should be usable without prior setup, e.g., by specifying a search query. Instead, both continuously provide relevant information and navigation affordances.

Comprehensible Relevance Developers should be able to easily relate the navigation targets presented to the source code they are currently reading.

Stacksplorer visualizes the 1-neighborhood of the currently edited method.

Based on these concepts, we first created Stacksplorer. Stacksplorer refers to the method currently being edited in the source code editor as the *focus node*. Stacksplorer visualizes and allows navigation to the 1-neighborhood of the focus method in the call graph. It implements comprehensible relevance, because the focus method and the potential navigation targets are closely connected in the call graph, and we expected developers to be able to easily understand how the navigation targets are related to the focus method. To implement proactive information visualization, Stacksplorer automatically updates whenever the focus method changes, due to any kind of navigation. The navigation that can be performed using Stacksplorer can be described as breadth-first call graph exploration.

Stacksplorer provides no support to navigate to indirect dependencies.

In a first study comparing Stacksplorer to an IDE without dedicated call graph navigation tools, we found a substantial positive effect of Stacksplorer on task completion time and task success rates for certain maintenance tasks. But we have also found potential limitations of Stacksplorer: First, Stacksplorer's breadth-first navigation does not provide support for navigating indirect dependencies, i.e., navigation to a method that is connected to the focus method through a longer path in the call graph. Ko et al. [2005] pointed out that these navigations are important to allow developers to understand the connection between different items in the working set. Second, previous studies have shown that several phases can be distinguished in the developers' navigation behavior (see Chapter 2): Developers first search an initial node that seems relevant, and then start exploring different paths through the source code from there [Ko et al., 2006]. The design of Stacksplorer does not make allowance for these phases.

Blaze visualizes one path including the currently edited method.

Based on these insights, we developed Blaze, a second call graph navigation tool: The key difference between Blaze and Stacksplorer is that Blaze does not show the 1-neighborhood of the focus method but one path through the call graph that includes the focus method. In contrast to Stacksplorer, the navigation that can be performed using Blaze can be considered as a depth-first exploration of the call graph. This allows developers to navigate to distant navigation targets in the call graph, i.e., to navigate to indirect dependencies. Blaze reuses the concept of a focus node: By default, the focus node is always the method currently being edited, but Blaze allows to lock the focus node to prevent automatic updates to the visualized path. This design supports the two navigation phases described above: During the initial search phase, Blaze shows context for the method in the source code editor. Once a relevant initial node was identified, this node can be stored by locking the focus node and Blaze can be used to explore the various paths that include the initial node.

Working prototypes of both tools were implemeted as Xcode plugins.

Both Stacksplorer and Blaze were implemented as fully functional prototypes on top of Apple's Xcode IDE[3] in version 3 for the Objective-C langauge. For the purpose of exposition, we will first introduce the design of both tools in depth, before presenting a comparative evaluation of both tools.

5.2.1 Stacksplorer

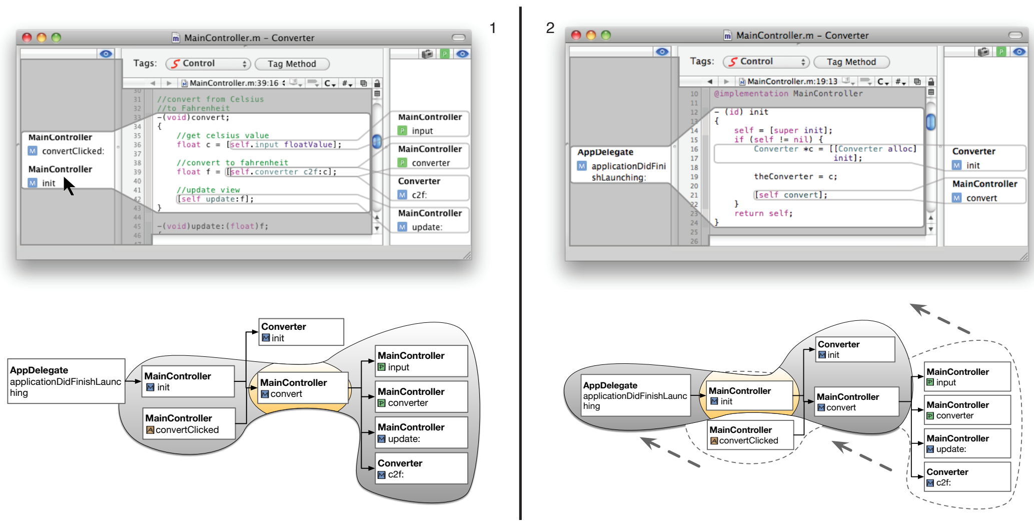
In Stacksplorer, the focus node is the method currently being edited.

Herman et al. [2000] presented a technique for graph exploration that places a window on top of the graph to show one logical frame at a time. The content of the logical frame is determined by its focus node [Huang et al., 1998]. Stacksplorer applies this concept to the call graph: The focus node is the method currently being edited in the source code editor, and the logical frame includes the call-graph neighborhood of this method. Semantically, this corresponds to all callers of and all methods called by (callees of) the focus node.

The list of callees can serve as a summary of the method.

Showing the neighborhood of the focus method has several potential benefits: A list of methods called from the focus method that is ordered by the occurrence of the method call in the focus node's implementation can serve as a simple summary of the method implementation, even though

[3]https://developer.apple.com/xcode/



```
//convert from Celsius
//to Fahrenheit
-(void)convert;
{
    //get celsius value
    float c = [self.input floatValue];

    //convert to fahrenheit
    float f = [self.converter c2f:c];

    //update view
    [self update:f];
}

-(void)update:(float)f;
```

```
@implementation MainController

- (id) init
{
    self = [super init];
    if (self != nil) {
        Converter *c = [[Converter alloc]
                        init];

        theConverter = c;

        [self convert];
    }
    return self;
}
```

Figure 5.2: A navigation to a caller in Stacksplorer causes the logical frame to shift. The navigation target becomes the new focus method, the old focus method now appears in the right column, and the left column is populated with new information. Figure adapted from [Karrer et al., 2011].

it does not retain information about any control structures. Further, the list of methods calling the focus method provides a quick overview of the contexts in which the focus method is used.

Stacksplorer shows callers on the left side of the editor showing the focus method, and callees on the right.

To visualize information as close to the focus node as possible and, hence, implement comprehensible relevance, Stacksplorer is integrated into the IDE and adds two interactive views, one on each side of the source code editor (see Figure 5.2). The side views combined with the central source code editor show the current logical frame and act as a fisheye view [Furnas, 1986]: The focus node is shown completely, while its neighborhood is visualized in a semantically simplified version. Stacksplorer lists all methods calling the focus node (i.e., incoming edges of the call graph) on the left, and all methods that are called from the focus node (i.e., outgoing edges of the call graph) on the right.

This design enables two orthogonal navigation "axes" to be used: Usually, developers can navigate through all methods that belong to one class by scrolling vertically in the central source code editor (given that in most object-

Stacksplorer's design enables use of a horizontal navigation axis for call graph navigation.

oriented projects each class is implemented in a separate source file). With Stacksplorer, developers can also navigate horizontally to a method visible in one of the side views. Clicking a method reveals it in the central source code editor and it becomes the new focus node, causing the side views to update accordingly. Consider the example in Figure 5.2: In the first picture, the developer is currently reading the `convert` method in `MainController`. When clicking on the `init` method of `MainController` that is listed as a caller, the logical frame shifts to the right: The old focus node `convert` moves to the right column and is now listed as one method being called by the new focus node `init`; the `init` method moves from the left column to the center and is now visible completely; and the left column is updated with new information.

Methods defined in external frameworks can be hidden.

Methods in the side views are represented by their name, the name of the class in which they are defined, and by an icon that allows to discern methods and properties (i.e., accessor methods defined in a formalized way in Objective-C). The side columns can be filtered to hide properties or methods defined in external frameworks, if the developer wants to focus on the control structure within their own code.

Optional graphical overlays connect navigation targets in Stacksplorer with the focus node's source code.

Stacksplorer automatically arranges the list entries in the side views to minimize the on-screen distance to the corresponding code in the editor. For densely written code, e.g., nested method calls, the assignment of list entries in the side columns to code locations can still be unclear. In this case, graphical overlays connecting a method call in the source code with the corresponding entry in the side views can be enabled. These overlays visually clarify Stacksplorer's implementation of comprehensible relevance. Highlighting the correspondence between source code and visualization was identified by Hundhausen et al. [2009] to be an important feature for development tools.

Methods could be tagged to store important paths through the source code.

As discussed before, developers collect the relevant information they found during navigation in a working set. To support this collection of knowledge, Stacksplorer implements a tagging mechanism that allows to assign one or more tags to every method. An important path through the call graph, e.g., a successfully identified delocalized programming plan, can be stored by tagging all methods on the path. In our first evaluation, however, we found that this mechanism was nearly never used. Our design required developers to tag methods one at a time while navigating, but until developers reached the method they were looking for they could not know whether or not they are on the correct path. This insight helped us design a different form of support for the Relate and Collect phases of the three-phase navigation model in Blaze.

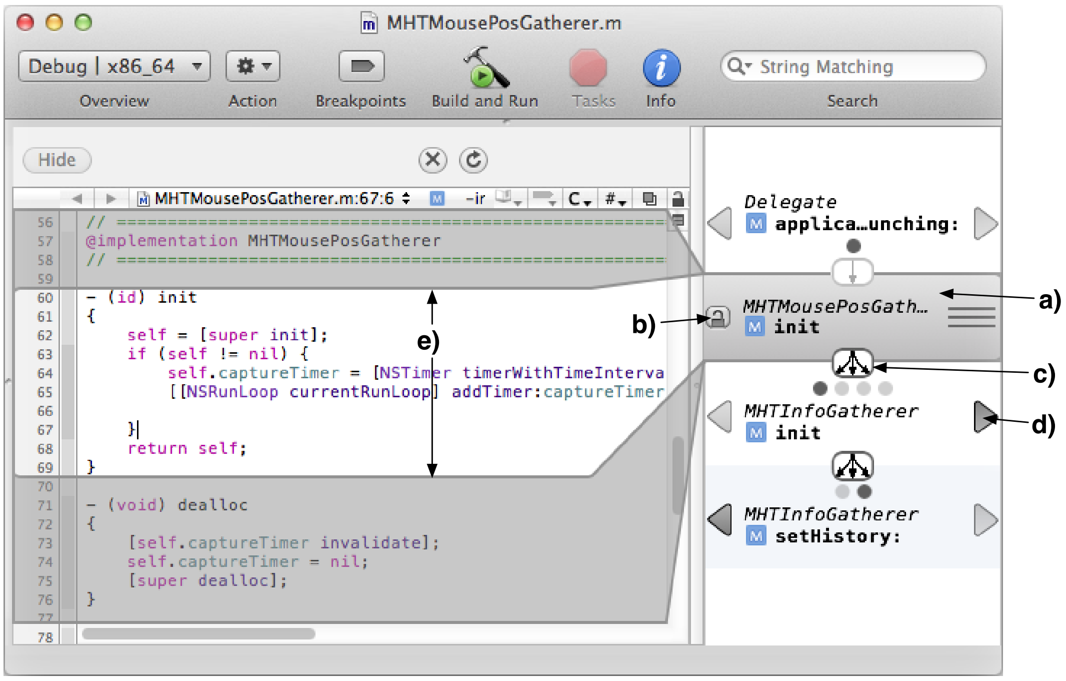
5.2.2 Blaze

Blaze can show a complete path including the focus node.

Stacksplorer is designed to show navigation targets that are closely related to the focus method, such that either side column becomes semantically meaningful: The left column shows the context of the focus method, the right column can serve as a summary of the focus method. This design does not allow, however, to visualize a complete delocalized programming plan at a time, i.e., sequences of method calls that implement a certain goal. Blaze is designed specifically to support this task: Instead of showing the neighborhood of the focus node, Blaze shows one complete path through the call graph that contains the focus node.

Blaze uses a column on the right side of the source code editor.

Blaze (see Figure 5.3) shows only one list of methods on the right side of the source code editor. Each method in the list calls the one below. This ordering and the inverse were both tested in a pen and paper prototype, but developers showed no clear preference for either design. The focus node (**a)**) is highlighted visually and always remains in a fixed position, to allow developers to utilize spatial memory to find it on screen. It splits the path in two halves, which both can be scrolled individually. The focus node can be moved up and down to change the ratio in which the available screen space is distributed between both halves of the path. To visualize comprehensible relevance, the focus node is connected to the corresponding method in the source code editor using an overlay (**e)**) spanning both views, similar to the overlays in Stacksplorer.



```
// ===================================================
@implementation MHTMousePosGatherer
// ===================================================

- (id) init
{
    self = [super init];
    if (self != nil) {
        self.captureTimer = [NSTimer timerWithTimeInterva
        [[NSRunLoop currentRunLoop] addTimer:captureTimer

    }
    return self;
}

- (void) dealloc
{
    [self.captureTimer invalidate];
    self.captureTimer = nil;
    [super dealloc];
}
```

Figure 5.3: Blaze visualizes a complete path through the source code that includes the currently edited method. The developer can change the path using the arrows next to each method on the path. Image Source: [Krämer et al., 2012a]

In the Search phase, Blaze updates automatically when the focus node changes.

Interaction with Blaze is designed to support the three-phase navigation model [Ko et al., 2006] discussed before: During the search phase, the focus node is synchronized to the method currently being edited in the editor. To implement proactive information visualization, Blaze updates after every change of the focus node to show a path that includes it. The amount of changes to the path are minimized, e.g., by keeping the path unchanged if the developer navigates to a method that is already on the path. During the search phase, Blaze aims to show additional navigation targets that are likely to be relevant, similar to Stacksplorer.

In the Relate and Collect phases, developers can explore different paths using Blaze.

Once developers move to the second phase, they can lock the focus node (**b)**), thus, disabling any automatic updates. They can now explore different paths that include the focus node. Blaze uses a combination lock metaphor to enable developers to exchange methods on the path with other alternatives while guaranteeing that the focus method remains on the path: Upstream of the focus node, i.e., in the part of the path that ends up calling the focus node, every method can be exchanged with another caller of the following method; downstream of the focus node, i.e., in the part of the path that originates in the focus node, every method can be exchanged with another method called by the preceding one. Developers can either scroll through all possible alternatives using arrow buttons on either side of each list entry (**d)**), or reveal a context menu showing all options using a button that connects two methods (**c)**).

5.2.3 A Common Backend

To obtain information about the call graph, Stacksplorer and Blaze share the same technical backend. Our backed uses static analysis and is built on top of Xcode's internal parser. This yields a level of robustness that is comparable to Xcode's existing tools.

Stacksplorer and Blaze share a backend built on top of Xcode's internal features.

After opening a project, our backend caches a complete, doubly linked representation of the call graph to reduce access times for information. An edge from method A to method B is stored in the call graph, if a method call exists in A such that B's name is equivalent to the called method's name and the name of the class implementing B is equivalent to the receiver's class name.

The call graph is cached to improve performance.

Whether or not a path in the graph is actually reachable at runtime is an undecidable problem [Lewis et al., 1997]. *Static traces* as proposed by LaToza et al. [2010c] can detect some infeasible paths and could extend our backend in the future. Further, our static analysis technique does not find method calls that are performed using dynamic dispatch at runtime. However, for the purpose of comparing the different interaction techniques in a controlled experiment, these technical tradeoffs were acceptable.

The backend cannot detect method calls performed using dynamic dispatch.

5.3 Comparing Call Graph Navigation Tools

In our first experiment, we compared Stacksplorer to an unmodified version of Xcode.

The first call graph navigation tool we created was Stacksplorer. To evaluate this tool, we compared it to an unmodified version of Xcode 3 as a baseline condition [Karrer et al., 2011]. Xcode 3, by default, only supports call graph explorations with a *Jump to definition* command. This allows developers to select a method call in the source code and navigate to the implementation of this method, effectively following one edge of the call graph.

Blaze was informed by the results from the first study and evaluated using the same setup.

As we have explained before, the insights we had gained in the first study informed the design of Blaze. Both Stacksplorer and Blaze implement the same key design ideas, comprehensible relevance and proactive information visualization, but while Stacksplorer implements a breadth-first approach and shows the complete neighborhood of the focus node in the call graph, Blaze implements a depth-first approach and shows one complete path including the focus node. To investigate the effect of the design changes we implemented in Blaze, we reused the study setup we had used in the first comparison between Stacksplorer and our baseline condition.

In our second experiment, we also compared our tools to the widely used Call Hierarchy.

At the same time, we decided to add a fourth condition to the study. We wanted to compare our call graph navigation tools to the popular Call Hierarchy tool, which is available in many IDEs except for Xcode. The Call Hierarchy is an interesting alternative design for comparison: Compared to Stacksplorer and Blaze, it can show a larger fraction of the call graph at once, but it does not implement our design principles. To allow for a fair comparison, we implemented a version of this tool ourselves (see Figure 5.1). Our implementation matches the interaction design used in the Eclipse IDE, and uses the same backend as Stacksplorer and Blaze.

In our first study, each participant solved two sets of two tasks, one in each condition. We found this design to provide little benefits for the analysis. Because the tasks are not comparable in terms of difficultly, we had to perform a separate between-groups analysis for each task. When we extended the study to include Blaze and the Call Hierarchy, we removed the second set of tasks and only assigned participants to one condition. This allowed us to perform a complete between-groups analysis with four conditions and two tasks, while reducing the time needed for each trial.

We streamlined the study procedure when analyzing Blaze and the Call Hierarchy.

In this thesis, we will only report on the combined analysis of all four conditions, i.e., we will not address the earlier comparison between Stacksplorer and Xcode separately. We found that all call graph exploration tools, namely, the Call Hierarchy, Stacksplorer, and Blaze, improve task success rates for certain bug fixing tasks when compared with the baseline. As call graph exploration is required to understand procedural schemas, this result is to be expected. More surprisingly, only Stacksplorer and Blaze were able to also improve task completion times. Possible reasons for this are either that Stacksplorer and Blaze are just easier to use or that they change the navigation strategies of developers to more efficient ones.

In this thesis we will present a joint analysis of both experiments.

In this section, we will first explain how the study was set up and which differences in performance metrics we have observed. Next, we are going to introduce an analysis technique that allows to operationalize navigation behavior. Finally, we will apply this technique to show that only Stacksplorer and Blaze cause a change in navigation behavior which in turn causes a reduction in task completion times.

5.3.1 Study Setup

The user study was designed to simulate a realistic software maintenance task in a controlled laboratory environment.

Tasks

Participants solved maintenance tasks in BibDesk.

Participants worked on two typical maintenance tasks that built on one another in BibDesk[4], an open-source bibliography manager for Mac OS X. During the study, participants should imagine that BibDesk was a commercial software and they were responsible for creating a trial version that comes with some limitations. Task 1 was concerned with BibDesk's Autofile feature that moves and renames PDF files according to a user-defined format string. Participants should modify this feature to always prepend a string to the generated filename to indicate the file was saved by a trial version. In task 2, participants were asked to identify a side effect of one possible solution of task 1.

Participants only had to point out which method needed to be changed.

As we were only interested in analyzing navigation behavior, participants did not need to actually implement changes. In task 1, the participants had to name the method in which the change could be implemented. Two different methods were possible which we both considered as correct. In task 2, the participants had to verbally describe which side effect could occur. We had no issues to determine whether or not answers were correct, since all incorrect answers given during the study could easily be proved to be incorrect.

Participants

Participants were mostly students and had to have at least some experience with Objective-C.

In our study, 33 Objective-C developers (32 males, 1 female) participated. Two participants were professional software developers while all other participants were students majoring in computer science. By recruiting mostly students, we intended to reduce the difference in programming expertise between participants, as proposed by Bragdon et al. [2010]. Participants were on average 26.3 years old ($SD = 2.6$). A minimum of 0.5 years of experience with Objective-C was required to participate in the study; the actual participants had an average of 2.6 years ($SD = 2.1$)

[4]http://bibdesk.sourceforge.net/

experience and spent an average of 12.6 hours per week ($SD = 11.6$) programming.

Procedure

First, participants were assigned to one of four conditions: In the Stacksplorer, Blaze, and Call Hierarchy conditions, participants worked using Xcode and the respective call graph navigation tool; in the control condition, participants worked using an unmodified version of Xcode. Nine participants worked in the Call Hierarchy condition, eight participants in each other condition. All conditions were similarly sampled in terms of coding experience in years and coding done per week.

We used a between-groups experiment design.

Before each trial, we prepared a Mac Pro computer with a fresh installation of Mac OS X, Xcode, and a call graph navigation tool depending on the condition. Participants used a 23-inch screen with a pixel resolution of 1920×1200.

All participants used identical hardware.

In all but the control conditions, the sessions started with the experimenter explaining the call graph navigation tool. Participants could try the tool while navigating in a code base unrelated to the task. The experimenter made sure not to reveal that the tool was designed by us.

The experimenter explained the call graph navigation tool.

Next, participants were allowed to browse the source code of the BibDesk project for 10 minutes. After this exploration phase, the experimenter handed task 1 and task 2 to the participants one task at a time, i.e., while working on task 1 participants did not know about task 2. Participants had a maximum of 25 minutes to finish task 1 and a maximum of 15 minutes to finish task 2. We recorded task completion time and task success as qualitative productivity measures. The task completion time ended after the participants gave the first answer to each task, regardless of the correctness of the answer. If time ran out, we recorded the maximum time as task completion time and the task as not being solved correctly.

The study started with an open exploration of BibDesk's source code.

Use of runtime analysis tools was prohibited during the study.

During the study participants had access to a compiled version of BibDesk to understand what the software does. They were not allowed, however, to use runtime analysis tools, which is consistent with previous studies [Bragdon et al., 2010; Robillard et al., 2004]. The experimenter was present at all times during the experiment and did answer questions about Xcode, Objective-C, and the Cocoa frameworks, as long as they did not directly relate to the BibDesk implementation.

We recorded audio and video during the study.

Participants were asked to think aloud during the study but not reminded to do so if they stopped, in order to not disturb them. Audio and screen contents were recorded throughout the study.

Call graph navigation tools were evaluated by participants after the study using a questionnaire.

After all tasks were completed, participants using any of the call graph navigation tools were asked to fill out a questionnaire that was comprised of the questions of the standard System Usability Scale (SUS) and six tool-specific questions in similar style. The study concluded with an open interview with every participant.

5.3.2 Performance Measures

We analyzed the effect of condition on two performance metrics: task completion time and task success.

Task Success

Developers were more successful when using any call graph exploration tool.

Using any call graph navigation tool, at least half of the participants were successful in completing both tasks, while only one participant in the control condition solved both tasks correctly. This difference is significant (one-sided Fisher's exact test, $p = 0.015$), and was also found for task 2 alone ($p = 0.024$) but not for task 1. No other pairwise comparisons between conditions showed any significant differences in task success.

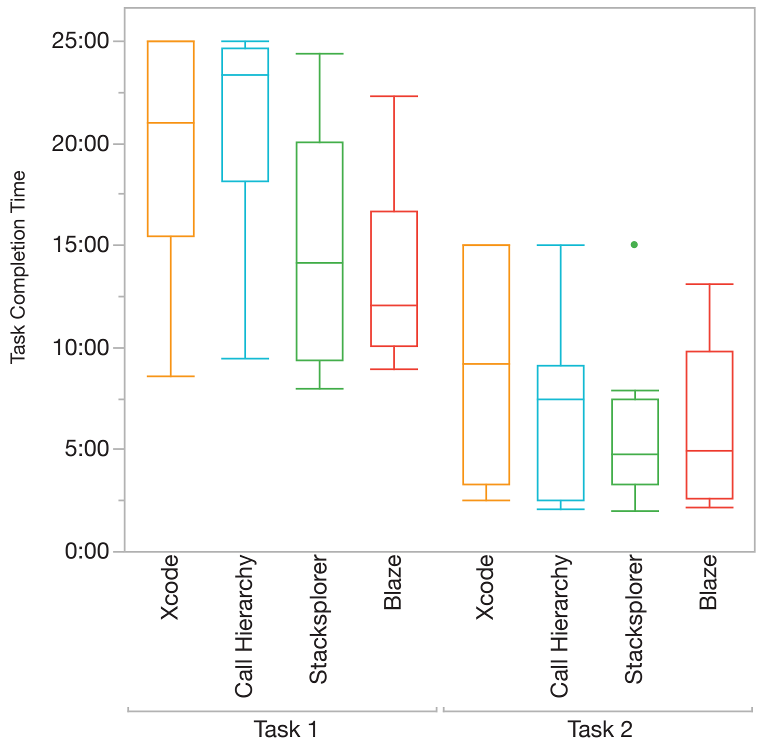


Figure 5.4: The graph shows the average task completion time by task and condition. An ANOVA shows a significant effect of task and condition, and Stacksplorer and Blaze are significantly faster than the control condition in pairwise tests.

Task Completion Time

We found an effect of condition on task completion time.

We expected similar results for the task completion time (see Figure 5.4): Any tool to explore the call graph should allow developers to solve maintenance tasks faster. A MANOVA with condition modeled as a between-groups factor and task modeled as a within-groups factor revealed a significant effect of task and condition but no interaction.

Task:	$F(1,29) = 65.281$	$\mathbf{p < 0.001}$
Condition:	$F(3,29) = 3.720$	$\mathbf{p = 0.022}$
Interaction:	$F(3,29) = 1.257$	$p = 0.307$

Task completion time was only reduced when using Stacksplorer or Blaze.

Pairwise post-hoc comparisons using a Dunnet-t test revealed that task completion times were significantly lower when using Stacksplorer or Blaze compared to the control condition, but there was no significant advantage of using the Call Hierarchy compared to Xcode. Post-hoc Tukey's tests comparing the different call graph navigation tools to each other showed no significant differences.

XC vs.	CH: $p = 0.662$	ST: $\mathbf{p = 0.038}$	BL: $\mathbf{p = 0.020}$
CH vs.	ST: $p = 0.167$	BL: $p = 0.095$	
ST vs.	BL: $p = 0.992$		

Discussion

Subjective ratings of all tools were similar.

Participants' responses to our post-study questionnaire match the quantitative findings about task success rates. We found no significant differences in terms of users' usability rating of the tool, or the perceived benefits for code comprehension, navigation speed, or orientation in the source code.

In summary, we found that all call graph navigation tools successfully increased task success rates but only Stacksplorer and Blaze also decreased the task completion times. This is surprising, as the same information can be explored with all tools. In fact, the Call Hierarchy is able to show the biggest amount of information at a time. There are two potential reasons for this result: Either, Stacksplorer and Blaze are easier to use, i.e., the time is saved by reducing the time required to interact with the tool, or Stacksplorer and Blaze encourage developers to use different navigation strategies that are more effective and hence reduce task completion times.

5.3.3 Model-based Comparison of Navigation Strategies

In the following analysis, we want to test the latter hypothesis: Do Stacksplorer and Blaze cause developers to employ different navigation strategies than the other tools used in the other conditions? If this hypothesis is true, it would show that the interdependence between tools and developers' strategies can be influenced through interaction design, even if the accessible information is similar. To compare navigation strategies in different conditions, we first need to be able to consistently quantify navigation behavior. In this section, we will present a methodology for this task, which is independent of the development environment and programming language used. We will use this method to compare the navigation strategies observed in our study and verify the obtained results using alternative metrics.

To understand the results of our study, we want to analyze the differences in navigation behavior between conditions.

Methodology

We are interested only in navigation between methods, consequently, we represent the navigation that was observed in one study session formally as a sequence of methods visited by the developer $H = (m_1, m_2, ..., m_n)$ where $\forall m_i, m_{i+1} \in H : m_i \neq m_{i+1}$. To quantify the behavior exhibited in these sequences, we propose to characterize each sequence by a set of features. Each feature represents to which degree the sequence can be explained by one of a set of well-known micro navigation patterns. We calculate this degree by using predictive models that predict navigation according to these micro navigation patterns. The models, and hence, the micro navigation strategies we use as features, have been compared by Piorkowski et al. [2011] before. The important difference between their use of these models and ours is that we are not interested in comparing the models in terms of their prediction accuracy. Instead, we use the prediction accuracy of *all* models to characterize a navigation sequence.

We use the accuracy with which a set of models can predict participants' navigations to describe their behavior.

Formally, each model takes a subsequence of H up to the element m_i as input and tries to predict m_{i+1}. The models predict m_{i+1} from the set $M_i - \{m_i\}$, with M_i approximating the methods known to a developer. That approximation includes all methods in files that have been opened so far, have been visible in a call graph navigation tool, or were included in a search result. The models first assign

Each model is characterized by the activation function it uses.

an activation value to every method in $M_i - \{m_i\}$ using an activation function $A_i : M_i - \{m_i\} \mapsto \mathbb{R}$. The higher the activation value assigned to a method the more likely the developer is navigating to this method. The activation function is characteristic to each model and depends on the micro navigation pattern represented by the model. Next, A_i is rank-transformed to obtain a ranking function $R_i : M_i - \{m_i\} \mapsto \mathbb{N}$. If multiple methods share the same activation value, the average rank is used. The models return a list of possible navigation targets that is comprised of the N top ranked methods. If the highest rank is assigned to more than N methods, the returned list is empty. We consider the prediction of a model correct, if the actual method m_{i+1} is contained in the returned list. Hence, N is a tuning parameter for all models: With N increasing, the prediction accuracy of a model also increases, at the expense of practical relevance of the prediction.

The difference between the individual models used is how each activation function works. We use the same activation functions as Piorkowski et al. [2011] and will only provide qualitative descriptions of the micro-navigation patterns represented by the models here:

Recency
The Recency model represents back and forth navigation between related methods, which is common to be able to understand the connection between two methods. It assigns higher activation values the more recently a method was visited.

Frequency
The Frequency model represents frequent revisits of important methods, e.g., the anchor point. It assigns higher activation values to more frequently visited methods.

Working Set
The Working Set model represents revisiting any method in the working set, i.e., a set of methods relevant to the task. The model assumes that the working set is comprised of the methods visited in the last δ navigation steps. It assigns methods in the working

set an activation value of 1 and an activation value of 0 otherwise. Our analysis technique uses $\delta = N$.

Bug Report Similarity
The Bug Report Similarity model represents searching for methods using keywords found in the bug report, or, more generally, the task description. It uses the tf-idf weight [Baeza-Yates et al., 1999], a well-known text similarity metric, of the bug report compared to the words in a method as the activation value. To calculate the tf-idf weight, a cleanup is performed by removing stop words and breaking camelCase identifiers apart.

The remaining models all use a graph $G' = (M_i, E')$. The activation value assigned to a method is higher the smaller the distance of the method is to m_i. The models differ in their definition of E'.

Within-File Distance
The Within-File Distance model represents navigations between methods that are close to each other in the source file. Since in object-oriented source code each class is often implemented in a separate file, this navigation represents navigation in the object hierarchy. E' is defined to contain undirected edged between methods that are adjacent in a source file.

Forward Call Depth
The Forward Call Depth model represents navigations to methods called from the current method, which is possible in many IDEs using the "Jump to definition" command. Consequently, E' is defined such that G' is the subgraph of the call graph G that is induced by M_i. More explicitly, $E' = \{(u, v) \mid (u, v) \in V \text{ and } u, v \in M_i\}$.

Undirected Call Depth
The Undirected Call Depth model represents navigations to methods called from the current methods as well as to methods calling the current method, i.e., all navigations along the call graph. E' is defined as for the Forward Call Depth model but using undirected edges.

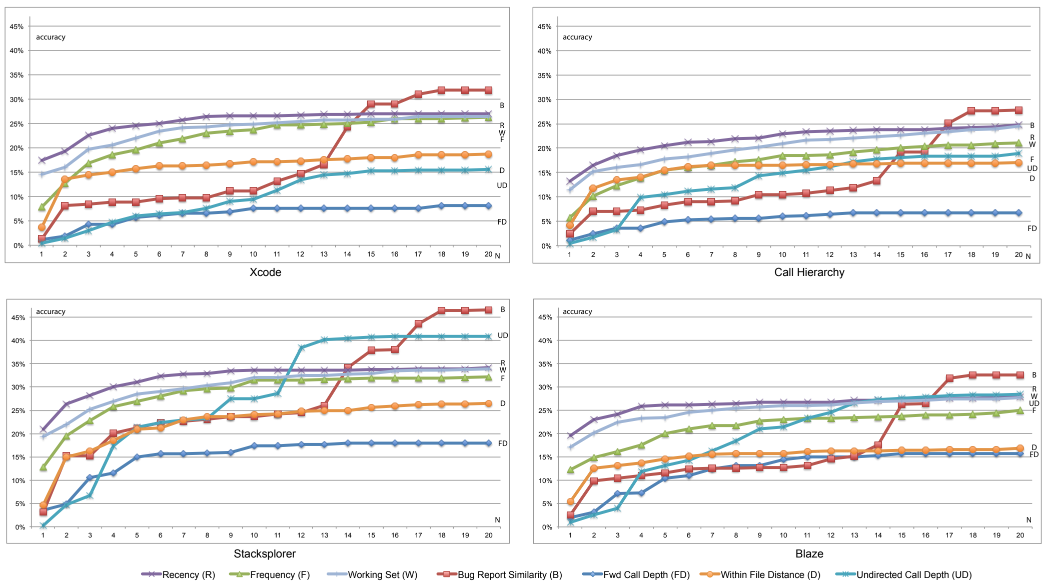


Figure 5.5: The graphs show for each condition the prediction accuracy of each model for values of N between 1 and 20. A visual inspection of the graphs reveals more call graph navigation in the Stacksplorer and Blaze conditions. Figure adopted from [Krämer et al., 2013].

Differences in Navigation Strategies

The navigation sequences of our study participants were generated using video annotation.

To apply the analysis of differences in navigation strategies to our study, we first needed to generate the formal navigation sequences from the video recordings of each trial. For this task, we used ChronoViz [Fouse et al., 2011] to annotate all navigation events in each user study session. We stored the target method of the navigation action and the tool used. For some actions, no single method could be determined as the target of a navigation. In these cases, we recorded the most precise abstract navigation target, such as a file, or a set of methods, e.g., when performing a search. Model predictions were still considered correct if at least one method in a set of methods was correctly predicted. For navigation targets that were not either a method or a set of methods, model predictions were always considered incorrect. Successive navigation actions that were less than 0.5 seconds apart were merged into one (e.g., navigating back twice).

For the analysis of our study, we calculated the prediction accuracy of each model for values of N between 1 and 20. The prediction accuracy averages in each condition are plotted over N in Figure 5.5. Statistical comparisons were performed only for $N = 1$, $N = 10$, and $N = 20$. An overview of all statistical results is presented in Table 5.1.

Prediction accuracies were compared for values of N between 1 and 20.

		$N = 1$		$N = 10$		$N = 20$	
Model	Factor	F	p	F	p	F	p
Frequency	Task	27.13	.001	28.60	.001	18.31	.001
	Condition	2.729	.062	3.384	**.031**	2.482	.081
	Interaction	1.733	.182	.384	.766	.813	.497
Recency	Task	27.82	.001	11.64	.002	9.215	.005
	Condition	2.4	.088	2.728	.062	2.009	.122
	Interaction	.696	.562	.793	.508	1.248	.311
Working Set	Task	22.28	.001	14.49	.001	9.518	.004
	Condition	2.757	.06	3.432	**.030**	2.222	.107
	Interaction	.823	.492	1.559	.221	1.124	.356
Bug Report Similarity	Task	3.8	.061	36.88	.001	142.3	.001
	Condition	.366	.778	3.056	**.044**	5.279	**.005**
	Interaction	.182	.908	2.784	.059	1.681	.193
Within-File Distance	Task	39.09	.001	11.30	.002	13.55	.001
	Condition	.679	.572	1.178	.335	1.682	.193
	Interaction	1.269	.303	.914	.447	.561	.645
Forward Call Depth	Task	.048	.828	.548	.465	0.299	.589
	Condition	.1.688	.191	6.470	**.002**	7.023	**.001**
	Interaction	.334	.801	1.693	.190	1.771	.175
Undirected Call Depth	Task	5.969	.021	6.000	.021	5.395	.027
	Condition	.857	.474	5.791	**.003**	9.514	**.001**
	Interaction	.125	.944	2.141	.117	5.344	.005

Table 5.1: The table shows the results of ANOVAs testing the effect of task, condition, and their interaction on the prediction accuracy of each model. For task $df = 1$, for condition and interaction $df = 3$, for error $df = 29$. All significant effects of condition are in bold face. Table adapted from [Krämer et al., 2013]

Frequency Revisiting few methods repeatedly is typical in the three-phase navigation model: Developers backtrack to the focus method and explore different paths from there. Blaze and the Call Hierarchy both allow to save the focus method in the tool and allow exploration of different paths including the focus method without opening the fo-

The Frequency model correlates with frequently backtracking to the focus method.

cus method in the editor again. Hence, we assumed that the prediction accuracy of the Frequency model was reduced in these conditions.

Prediction accuracy in the Stacksplorer condition was lower than in the Call Hierarchy condition.

The prediction accuracy of the Frequency model was significantly different between conditions for $N = 10$. Pairwise, post-hoc Tukey tests only find a significant difference between Stacksplorer and the Call Hierarchy ($p = 0.018$) with the prediction accuracy being higher in the Stacksplorer condition. This confirms our hypothesis that participants performed less revisits to previously visited methods when using the Call Hierarchy.

These models represent back and forth navigation between related methods.

Recency & Working Set Developers frequently navigate back and forth between two methods to be able to understand how they are related (see Chapter 2). For this reason, we expected this kind of navigation to happen less frequently in conditions with lower task completion time. This behavior is represented by the Recency model, by the very similar Working Set model, and, in part, by the Frequency model.

A significant difference was only found between Stacksplorer and the Call Hierarchy.

The prediction accuracy of the Recency model was not significantly different between conditions for any N. For the Working Set model, in contrast, we found a significant effect of condition for $N = 10$. Pairwise, post-hoc Tukey tests again only found a significant difference between Stacksplorer and the Call Hierarchy with prediction accuracy being higher in the Stacksplorer condition. The increase in back and forth navigation between previously visited methods did not influence the number of distinct methods visited throughout the trial, as those were similar in the Stacksplorer and Call Hierarchy condition ($p = 0.815$).

Stacksplorer allowed frequent back-and-forth navigations without impairing performance.

Contrary to previous results and our own assumptions, we found Stacksplorer to allow frequent back and forth navigation without impairing performance. Because revisiting some functions frequently was also more common in the Stacksplorer condition than in the Call Hierarchy condition, we conclude that using Stacksplorer allowed developers to perform a thorough exploration of relatively closed subsets of methods that are connected by the call graph. Understanding those subsets well was shown to be important to for code comprehension before [Sillito et al., 2008].

The maximum prediction accuracy for the Recency and Working Set models is achieved roughly for $N = 10$. Further increasing N only marginally improves the prediction accuracy. This can serve as an estimate of the size of a typical working set in our study.

These models also allow to estimate the size of the working set.

Bug Report Similarity We did expect participants to search for textual clues from the task description to get started. However, this task was not supported by any call graph navigation tool, as none of these shows information about the textual content of the method.

This model represents searches for textual clues from the task description.

Surprisingly, we found a significant effect of condition on the prediction accuracy of the Bug Report Similarity model for $N = 20$. Pairwise, post-hoc Tukey's tests show a significant difference between Stacksplorer and the Call Hierarchy as well as between Stacksplorer and the control condition (XC: $p = 0.035$, CH: $p = 0.004$).

We found significant differences between Stacksplorer and both the Call Hierarchy and control condition.

This result can be explained using information foraging theory: By implementing proactive information visualization and constantly updating automatically, Stacksplorer provides additional information targets. Developers are likely to select from those targets based on textual similarity to the task description [Lawrance et al., 2008]. Stacksplorer can only increase the discoverability of these targets, textually meaningful identifiers have to be already present in the source code. Because developers using Stacksplorer were also faster than those using the Call Hierarchy or Xcode, these results show that using good identifiers improves code comprehension. This is consistent with previous research that indicates that a careful API design can make source code easier to understand (see Section 2.3.1 and Chapter 3).

Stacksplorer provides additional navigation targets and developers likely pick textually related ones.

Within-File Distance Navigation that is predicted correctly by the Within-File Distance model corresponds to the navigation along the object hierarchy, because a source file

This model correlates with navigation along the object hierarchy.

usually correlates with one class. We expected call graph navigation tools, which specifically support navigating the procedural hierarchy, to have no effect on the prediction accuracy of this model.

We found no differences between conditions.

The statistical results match our expectation: Condition had no effect on the prediction accuracy of the Within-File Distance model for any N.

These models correlate with call graph navigation.

Forward Call Depth and Undirected Call Depth The Forward Call Depth and Undirected Call Depth models specifically represent navigation along the call graph, and therefore, we expect their prediction accuracy to improve when a call graph navigation tool is available.

Prediction accuracy in the Stacksplorer and Blaze conditions is higher than in the other conditions.

In fact, for both models and values of $N = 10$ and $N = 20$, we found a significant effect of condition on prediction accuracy. Post-hoc tests (for $N = 20$ and the Forward Call Depth model) show results similar to those observed for the task completion time: In both the Stacksplorer and Blaze conditions prediction accuracy is significantly higher than in the control condition (Dunnett test, ST: $p = 0.004$, BL: $p = 0.022$) or the Call Hierarchy condition (Tukey's test, ST: $p = 0.003$, BL: $p = 0.022$). Results for the Undirected Call Depth model are similar, except that the pairwise comparison between the Blaze and the Call Hierarchy conditions was not significant. No difference between Xcode and the Call Hierarchy could be found.

These models need larger N to provide useful predictions.

It is important to note that both Call Depth models predict badly for small values of N. In our data accuracy increases notably for $N > 6$. The reason for this effect is that for the Forward Call Depth model all methods called by m_i (with m_i being the current method) are assigned the highest rank. For the Undirected Call Depth model, all methods calling m_i are also assigned the highest rank. However, a model only predicts anything if N is greater or equal to the number of methods having the highest rank. The 277 methods visited during the study by at least one participant had an average of 1.81 ($SD = 5.93$) callers and 3.49 ($SD = 4.63$) callees. Those methods that were visited by at least half the participants had even more neighbors (callers:

3.33 ($SD = 2.58$), callees: 11.33 ($SD = 6.83$)). On that account, useful predictions can only be expected for relatively large N.

Factor Analysis

When designing the model-based analysis methodology, we assumed that every model corresponds to a specific navigation micro-pattern. After having applied this methodology, we performed a factor analysis to find models with similar prediction accuracy, i.e., to find navigational micro-patterns that commonly appear together.

A factor analysis can determine which navigational patterns commonly appear together.

We performed a factor analysis for $N = 10$ and $N = 20$ for each task individually, yielding four separate analyses. For all analyses, the Kaiser criterion indicated three underlying factors, except for $N = 20$ and task 2 where it only indicated two underlying factors. Hence, in the next step, we calculated the factor loading for three underlying factors in all analyses. We used the Maximum Likelihood method with Oblimin rotation. For $N = 20$ and task 2 the correlation matrix was not positive definite, so the Maximum Likelihood method was not applicable and we removed this case from the further analysis.

We used the Maximum Likelyhood method with Oblimin rotation.

In the next step, we performed chi-square tests to test whether or not common factors still exist and whether or not three factors are sufficient. The first test was significant in all cases, while the second test was significant in no case.

We found three underlying factors.

	$N = 10$		$N = 20$
	task 1	task 2	task 1
H0: no common factors	$p < 0.001$	$p < 0.001$	$p < 0.001$
H0: 3 factors are sufficient	$p = 0.831$	$p = 0.719$	$p = 0.316$

In combination, these results confirm that the information can be reduced to three factors. On average, the three extracted components were able to explain $85,5\%$ ($SD = 12,8\%$) of the total variance found.

$N = 10$

	Task 1			Task 2		
Components	1	2	3	1	2	3
Frequency	0.584	0.253	-0.376	0.637	-0.018	-0.005
Recency	0.970	0.084	0.049	0.637	-0.018	-0.005
Working Set	1.001	-0.032	0.073	0.600	0.068	0.009
Bug Report Similarity	-0.150	0.070	0.544	0.077	-0.114	1.006
Within-File Distance	0.199	-0.053	0.781	-0.062	0.101	0.572
Forward Call Depth	0.108	0.714	-0.160	0.150	0.827	0.052
Undirected Call Depth	-0.033	0.982	0.158	-0.066	0.876	-0.019

$N = 20$

	Task 1		
Components	1	2	3
Frequency	0.827	0.016	-0.225
Recency	1.001	-0.011	0.113
Working Set	0.987	0.015	0.110
Bug Report Similarity	-0.037	-0.087	0.499
Within-File Distance	0.067	0.106	1.015
Forward Call Depth	0.014	0.969	-0.095
Undirected Call Depth	-0.010	0.871	0.067

Table 5.2: The table shows the factor loading for three underlying factors. We performed task-wise analyses for $N = 10$ and $N = 20$. For $N = 20$ and task 2 the Maximum Likelihood method was not applicable. Factor loadings are consistent across analyses.

The structure of the three factors was identical in every analysis.

The structure of the three components identified was identical in every analysis (see Table 5.2): The first component is comprised of the Frequency, Recency and Working Set models. The alignment of these models was to be expected: The Recency and Working Set models are very similar because the Working Set model estimates the working set using the most recently used methods. Further, navigating back to a recently visited method will always also increase the frequency with which this method is visited. The second component is comprised of the Forward and Undirected Call Depth models. Again, the similar definitions of these models are indicative for their alignment as one factor. The third component is comprised of the Within-File Distance and the Bug Report Similarity models. Here, the similarity between both models is less intuitive and statistically less pronounced. We assume that developers first searched for a term they found in the task description to find a class which is relevant to implement the specific feature they needed to work on. Then, developers would explore the object structure of this class, which, given it is relevant to the task, likely contains other words found in the task description as well.

The results of the factor analysis seem plausible but should nevertheless be interpreted carefully, because the factor analysis of our model-based analysis technique is based on data from only one study. Further, we could not apply a consistent statistical method in all analyses.

The factor analysis should be confirmed after applying the analysis method in more studies.

5.3.4 Non Model-Based Comparison of Call Graph Navigation Strategies

The model-based analysis methodology we used to characterize and to compare navigation strategies between conditions is a new technique we have created. To verify that the results obtained using this technique are sound, we also analyzed our data using other quantitative measures that describe the amount of call graph navigation performed and the tools used for this purpose.

We used other measures for the amount of call graph navigation to verify our analysis method.

We first analyzed if the percentage of navigations along a call graph edge is influenced by either task or condition (see Figure 5.6). An ANOVA reveals a significant effect of both factors as well as a significant interaction effect.

We compared the percentage of call graph navigations between conditions.

Task:	$F(1, 29) = 10.892$	$\mathbf{p = 0.003}$
Condition:	$F(3, 29) = 11.002$	$\mathbf{p < 0.001}$
Interaction:	$F(3, 29) = 3.877$	$\mathbf{p = 0.019}$

The effect of task is to be expected as both tasks were different in difficulty and complexity. Post-hoc tests on the effect of condition have to performed individually for each task to understand the interaction effect. We found that in task 1 the percentage of call graph navigations was generally lower and there were less differences between the conditions, while in task 2 the differences between the conditions increased substantially (see Figure 5.6). This is also

The analysis revealed a significant interaction effect between task and condition.

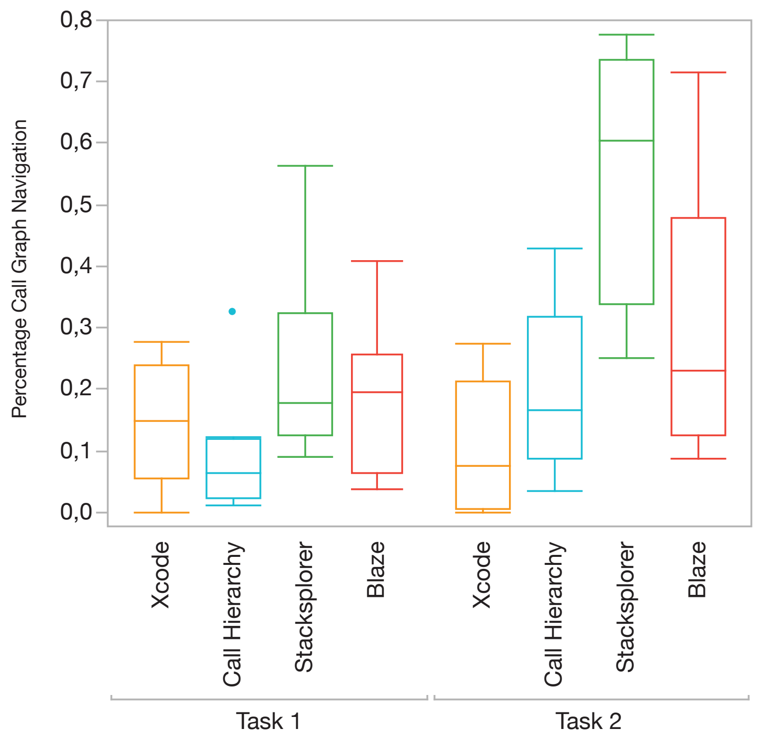


Figure 5.6: The figure shows the percentage of navigation actions that followed an edge in the call graph. In task 2, we find substantial differences between the conditions.

reflected in the results of post-hoc t-tests using Bonferroni correction. In task 1, no differences between individual conditions could be found.

XC vs.	CH: $p = 1.000$	ST: $p = 0.972$	BL: $p = 1.000$
CH vs.	ST: $p = 0.112$	BL: $p = 0.694$	
ST vs.	BL: $p = 1.000$		

In task 2, we found that developers using Stacksplorer performed significantly more navigations along the call graph than those who did not use a call graph navigation tool or the Call Hierarchy.

XC vs.	CH: $p = 1.000$	ST: $\mathbf{p < 0.001}$	BL: $p = 0.133$
CH vs.	ST: $\mathbf{p = 0.001}$	BL: $p = 0.584$	
ST vs.	BL: $p = 0.118$		

In the second task, developers did not need to perform an initial search phase, because a starting point was given in the task description. We suspect that this difference to task 1 caused the more pronounced effects in task 2.

A call graph navigation sequence is a series of navigations along a call graph edge.

Next, we analyzed the length of call graph navigation sequences, i.e., exploration phases during which only call graph navigation occurred. Formally, a call graph navigation sequence $S = (m_m, m_{m+1}, \ldots, m_{n-1}, m_n)$ is a substring of the developer's navigation history H, such that $\forall m_i, m_{i+1} \in S, (m_i, m_{i+1}) \in E$ or $(m_{i+1}, m_i) \in E$. An ANOVA revealed a significant effect of condition on the average length of these sequences.

Task:	$F(1, 31) = 0.379$	$p = 0.543$
Condition:	$F(3, 31) = 5.834$	$\mathbf{p = 0.003}$
Interaction:	$F(3, 31) = 1.679$	$p = 0.192$

Call graph navigation sequences were longer using Stacksplorer and Blaze compared to the control condition.

Post-hoc Tukey's tests again showed that sequences in the Stacksplorer and Blaze conditions were significantly longer than those in the control condition, while sequences in the Call Hierarchy condition were not. Additionally, call graph navigation sequences in the Stacksplorer condition were significantly longer than those in the Call Hierarchy condition.

XC vs.	CH: $p = 0.901$	ST: $\mathbf{p = 0.007}$	BL: $\mathbf{p = 0.035}$
CH vs.	ST: $p = \mathbf{0.031}$	BL: $p = 0.133$	
ST vs.	BL: $p = 0.904$		

Indirect call graph navigation only occurred very rarely.

An exploration phase in the call graph was not counted as a call graph navigation sequence, when developers navigated along more than one edge at a time. This could potentially influence results for Blaze and the Call Hierarchy, as both tools support these navigations. Surprisingly though, in all trials combined, we found only nine instances of such navigations.

Developers all preferred call graph navigation tools to existing means in Xcode to perform call graph navigations. In

All tools were used for a similar percentage of call graph navigations.

all treatment conditions, developers performed about two thirds of navigations along a call graph edge using the call graph navigation tool.

CH:	$M = 68.2\%$	$SD = 37.1\%$
ST:	$M = 65.0\%$	$SD = 31.2\%$
BL:	$M = 65.0\%$	$SD = 39.8\%$

Call graph navigation tools reduced the amount of project-wide textual searches.

In the control condition, participants still needed to perform call graph navigation to solve the tasks. The most important strategy to do so was to use project-wide textual search. Developers wanting to find all callers of a method would search for the method name using the project-wide search. This technique is slow and error prone. Errors often occur when methods are called similarly, especially when a method's name is the prefix of another method's name. Condition had a significant effect on the number of times a project-wide search was performed.

Task:	$F(1, 29) = 9.542$	$\mathbf{p = 0.004}$
Condition:	$F(3, 29) = 4.224$	$\mathbf{p = 0.014}$
Interaction:	$F(3, 29) = 0.522$	$p = 0.671$

Using a post-hoc Dunnett test, we found that the usage of the project-wide search declined significantly in all treatment conditions compared to the control condition ($p \leq 0.001$ for all conditions).

5.3.5 Discussion

We investigated if differences in task completion times can be explained by changed navigation strategies.

After comparing task completion times and task success rates between conditions, we wondered why participants using the Call Hierarchy could not solve tasks faster than those in the control condition, but participants using Stacksplorer and Blaze could. We suspected that Stacksplorer and Blaze are not just easier to use, but encourage developers to use different navigation strategies, which in turn were more effective to comprehend the source code in our study.

To be able to detect changes in navigation strategies, we presented a new analysis technique that allows to quantify and compare inter-method navigation strategies. In the remainder of this section, we will first discuss our results with respect to the initial research question and then discuss what we have learned about our analysis technique.

Differences Between Call Graph Navigation Tools

All tools replaced cumbersome project-wide searches.

The analysis of tool usage in the different conditions showed that adoption rates in the Call Hierarchy conditions were comparable to the Stacksplorer and Blaze conditions. In the control condition, developers often explore the call graph using project-wide searches. This interaction is replaced by a far more streamlined and less error prone interaction in all tool conditions. We believe this shift to a more reliable interaction causes participants in all tool conditions to achieve higher task success rates compared to developers in the control condition.

Only Stacksplorer and Blaze encouraged developers to change their navigation behavior.

Stacksplorer and Blaze, in contrast to the Call Hierarchy, change the developers' navigation strategies towards a strategy that is more focused on call graph exploration than before. Adopting this strategy allowed developers to solve tasks faster than developers in the Xcode and Call Hierarchy conditions. We believe that this change in navigation strategy is promoted by the two design concepts shared by Stacksplorer and Blaze: *proactive information visualization* and *comprehensible relevance*.

Comprehensible relevance increases the usefulness of navigation targets for developers.

With *comprehensible relevance* we refer to the fact that visualized information should be connected to the task in a way that is immediately understandable to a developer. Stacksplorer, which caused the most substantial shift in the developer's behavior, only showed methods directly connected to the focus method in the call graph. The relationship between these methods and the focus method is easy to understand, especially when using a bottom-up comprehension strategy. Stacksplorer does not allow developers to disable automatic updates, hence developers can rely on the side columns always being related to the focus method in the same way. Blaze still adheres to the principle of comprehensible relevance more than the Call Hierarchy, as one path through the call graph roughly maps to a single concern the focus method is involved in. In contrast, the Call Hierarchy allows nearly unrestricted exploration, only enforcing that every visible method is connected to the focus method in the call graph somehow.

Comprehensible relevance of navigation targets seems to decrease with their distance in the call graph.

The presumed benefit of Blaze and the Call Hierarchy is that both allow navigation to methods that were more than one edge away from the focus method. However, in contrast to previous results [Ko et al., 2005], these navigations were rarely performed in our study. We suspect that the relevance of methods further away in the call graph is hard to assess, especially, when developers are, like in our particular setup, unfamiliar with the source code and fear to get lost. Additionally, both Blaze and the Call Hierarchy allow some exploration of distant navigation targets, i.e., reading the method and class name, without actually performing the navigation.

By implementing proactive information visualization, Stacskplorer and Blaze provide additional information scent.

With *proactive information visualization* we refer to automatically displaying information instead of waiting for the user to perform a query. This provides navigation targets and, thus, information scent (see Chapter 2.2.2) automatically and can encourage navigations that a user would otherwise not have performed. Both Stacksplorer and Blaze implement this property, and about half of the participants in the Call Hierarchy condition proposed to add automatic updates in post-session questionnaires, even though they were not aware of the other tools.

Different design decisions of the tools could have cause the change in strategies.

Limitations We could show a correlation between the performance measures that distinguish the different tools and the navigation strategies that the tools promote. However, it remains unclear if the ability to change navigation behavior is caused by the tools visualizing different parts of the call graph or by the different presentation of the information. For example, overlays help parsing the displayed information in Stacksplorer and Blaze, but are absent in the Call Hierarchy condition.

One of the most common errors we observed in the Blaze and Call Hierarchy conditions were mode errors: Participants using the Call Hierarchy often confused caller and callee view modes, or they forgot that the currently selected method in the Call Hierarchy is not necessarily the one visible in the editor. When using Blaze, mode errors occurred when the path was too long to fit the screen. In this case, developers could scroll the part of the path below and above the focus method, eventually hiding a part of the path between the focus node and the next visible method. Changing the design to prevent these mode errors could improve the performance of both tools.

The interaction of Blaze and the Call Hierarchy could be improved by preventing mode errors.

Even though we identified different problems of the Blaze user interface, e.g., that participants rarely navigated to targets distant from the focus method, or that mode errors occurred, task completion times in the Blaze condition were on par with those in the Stacksplorer condition. This indicates that selecting which information to display (comprehensible relevance) and providing information automatically (proactive information visualization) are important enough to mitigate certain interface design problems.

The design guidelines we applied mitigated the negative effects of these mode errors in Blaze.

All results we have found are specific to navigation in Objective-C source code. To be able to generalize the results to other object-oriented programming languages, we performed a formative study about navigation behavior beforehand [Karrer et al., 2011; Krämer, 2011]. In observing and interviewing six Objective-C developers in a real life scenario, we found that the navigation behavior we observed with these developers is qualitatively identical to the behavior observed in previous studies (see Chapter 5).

Our results are specific to Objective-C source code.

In our study, only six methods were visited by more than half of the participants. These methods were either the solution to a task or very closely related. Nearly half of all the methods we saw being visited during the sessions (45%), were only visited by one participant. These methods were probably visited during the first phase of the three-phase navigation model, during which developers search for an anchor point. This diversity of methods visited during the first phase likely added noise to our model-based analysis.

A huge diversity in methods visited during the Search phase likely added noise to our analysis.

Model-based Navigation Analysis Method

The model-based analysis method to describe and compare navigation behavior was first proposed and used in this study. While it is difficult to appropriately judge the capabilities of such a method after only one application, we found several indications that the method we propose is sound.

The factor analysis showed that our analysis method detects similar navigation strategies as observed in previous studies.

In the factor analysis, we found that our method generates results along three major axes. A first indication for the soundness of the obtained results is that these axes correspond to previous empirical results of common navigation patterns of developers: According to the three-phase navigation model, developers collect a working set of important methods and try to understand their relationship in depth. For that task, they frequently switch back and forth between methods in the working set. The first underlying factor of our analysis method is comprised of models specifically designed to represent such navigation: Frequency, Recency, and Working Set. The second axis describes navigation related to exploration of the procedural structure of the program, represented by the Forward and Undirected Call Depth models. The last axis describes navigation related to exploration of the (orthogonal) object structure of the program, represented by the Within-File Distance and Bug Report Similarity model.

Within-File Distance and Bug Report Similarity are reliant on assumptions about the project.

The last axis is likely most reliant on artificial assumptions about the project: The object structure of a program only aligns with Within-File Distance if each class is implemented in a separate file and can be browsed by scrolling through this file. While this assumption relies on the specific development environment used, we think it likely holds for many software projects implemented with current tools. Bug Report Similarity only aligns with the object structure if the terms used in the bug report correlate with terms used in the source code, and if the source code related to these terms is located in a common class or sub-graph of the class hierarchy. We suspect that whether or not these requirements are met is largely dependent on the specific bug report and project.

We obtained similar results from our model-based analysis as using other quantitative measures to describe the amount of call graph navigation performed (see Section 5.3.5). This further supports the validity of our method. When discussing the results of all the analyses we performed, we found that the model-based analysis is helpful to establish that a difference in navigation behavior does exist between different conditions. To identify the reasons for the observed difference, we propose to cross-validate results with other metrics that describe specific details on the navigation behavior.

Other metrics for call graph navigation confirm the results obtained using our analysis method.

To allow other researchers to apply our analysis methodology, we have released a tool for Mac OS X to calculate prediction accuracies from a ChronoViz annotation file and an XML representation of the call graph.

http://hci.rwth-aachen.de/developerNavigation

5.4 Future Work

This work leaves several opportunities for future work. First, we hope to see our analysis methodology being applied in a broader variety of studies to find further support for its viability. The software we provided should help researchers in adopting our method. With a larger set of analyses of navigation behavior, researchers could identify differences depending on tasks, work environments, and various tools used in the development process.

We hope that our analysis method is adopted in future research projects.

In our study, we sampled a comparatively small variety of call graph navigation tools. Apart from two conditions representative of current development environments, we created two tools based on the same design concepts. In the future, this evaluation should be extended to other tool designs, especially those that more holistically rethink the interface paradigm of an IDE, such as Code Bubbles. This could provide further support for our design guide-

Stacksplorer and Blaze should be compared to a broader range of development tools.

lines and help identify other design aspects responsible for changes in developers' behavior. Further, our guidelines should be evaluated individually, e.g., by creating a variant of the Call Hierarchy that updates automatically to implement proactive information visualization, to explore the effect of each individual guideline.

We propose to investigate how a changed navigation behavior correlates with other aspects of development.

We found that the observed changes of developers' strategies correlate with reduced task completion times. In the future, it would be interesting to explore how the changed strategies correlate with other aspects of development, such as learnability or the quality of the resulting code.

Chapter 6

Live Coding

"I object to doing things that computers can do."

—Olin Shivers

Publications: The work presented in this chapter on METIS was done in collaboration with Thorsten Karrer, Björn Heinen, Ewgenij Belzmann, and Joachim Kurz. It was presented as a short paper at VL/HCC 2014 [Krämer et al., 2014]. The work on FIDDLETS was done in collaboration with Joel Brandt, and Dennis Lewandowski. Björn Heinen performed an early exploration of possible designs for a live coding editor in his Bachelor's thesis [Heinen, 2012]. This refined version of this prototype, METIS, was created and studied as part of the Master's thesis of Joachim Kurz [Kurz, 2013]. At the same time, Ewgenij Belzmann implemented the required backend [Belzmann, 2013]. Hendrik Wolf explored novel visualization techniques for complex runtime information in his Bachelor's thesis [Wolf, 2014]. FIDDLETS was iteratively designed and studied as part of Dennis Lewandowski's Master's thesis [Lewandowski, 2015]. The author of this thesis supervised all theses mentioned and contributed to all stages of prototypes, to the software architecture, and to the design of the studies. He later extended the analyses of Kurz and Lewandowski to the state presented here.

Developers regularly execute the application they develop manually.

So far, we have supported developers in comprehending source code by reading it. However, because source code is written to be executed by a computer, it is often easier for developers to execute the code instead of simulating the computation mentally, and to observe the behavior of the code using debugging tools [Maalej et al., 2014]. Developers perform these manual executions of the code frequently [Brandt et al., 2009a], and we already exploited this behavior in Chapter 4.

Manual edit-test-edit cycles require developers to switch between activities.

Manually executing the source code repeatedly throughout the development process can lead to several problems: First, developers have to switch constantly between multiple tools, usually a debugging tool and a code editor. Secondly, developers need to repeatedly provide input manually for the application they create in order to exercise the part of the software they are currently working on. Due to this cumbersome interaction, developers do not often perform manual tests if they are confident that their change is correct. However, edit-test-edit cycles that are too long can delay the discovery of newly introduced bugs, because the sooner a bug is discovered, the faster it can be fixed [Saff et al., 2003].

Live development tools automatically execute the application after each change.

To mitigate these problems, we will apply the concept of *proactive information visualization* that we have identified in Chapter 5 to the design of runtime analysis tools. The resulting tool is a *live* programming environment that provides feedback on the program execution immediately after each change to the source code. More precisely, these systems can show values of method parameters, conditions, variable assignments, or the order in which methods were called. Live coding environments are not a new concept; and in the first section of this chapter we will summarize previous work on the topic. Next, we are going to present a study to analyze how using a live coding environment affects the developers' behavior. We will show that developers working with a live coding environment fix bugs significantly faster and switch between creating and correcting code more frequently. In the last section of this chapter, we will discuss the technical challenges of live coding tools. We will present an approach to scale live coding to big, real-world projects by using live coding only for a small snippet of the actual application. Our system contextualizes this snippet to simulate an execution of the complete application.

6.1 Related Work

Tanimoto introduced a scale for the liveness of programming environments.

The term *liveness* to classify the immediacy with which programming environments provide feedback to the developer was first introduced by Tanimoto [1990, 2013]. In Tanimoto's original classification [Tanimoto, 1990], systems can achieve four different levels of liveness; later the scale was extended to include a fifth and sixth level [Tanimoto, 2013]. The first level refers to non-executable program descriptions, such as UML diagrams or requirements documents. The second level refers to executable program descriptions, i.e., source code. The third level describes executable program descriptions that are automatically executed after every change. The fourth level refers to environments that keep the software running while code is changed, so that changes to the source code affect all future events without requiring a restart. The fifth level describes systems that concurrently execute nearby versions that a developer might want to explore. The sixth level extends this idea to describe systems that understand higher level goals and predict larger chunks of code. In this chapter, we will call systems *live* if they achieve at least level 3 on Tanimoto's liveness scale.

Live-coding has frequently been implemented in special purpose programming environments.

In the past, several coding environments have implemented levels 3 liveness and above. The earliest implementation of a system that achieved level 3 liveness we are aware of is Visicalc in 1979 [Grad, 2007], the first spreadsheet environment. Henderson et al. [1985] discussed only a few years later, how the live concepts in Visicalc could be applied to a programming environment for textual programming languages. Burnett et al. [1998] showed how spreadsheets can be extended to achieve level 4 liveness by applying a lazy algorithm to mark cells that need to be reevaluated after events. Morphic [Maloney et al., 1995], the user interface toolkit of the Self programming language [Smith et al., 1995], first brought level 4 liveness to graphical user interfaces. In Morphic, developers can change both the source code of each widget and the widget hierarchy without stopping or restarting the application. Today, level 4 liveness is most commonly found in visual, flow-based programming environments, such as Quartz Composer[1] or Max[2]. In these environments, a program is assembled by connecting different patches on a two-dimensional canvas (i.e., methods). As we have discussed in Section 2.3.2, though, we will focus on textual programming languages in this thesis.

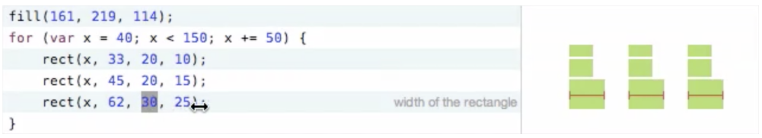
Example-based programming implements level 3 liveness for textual programming languages.

The most common feature related to liveness in textual programming environments is continuous compilation. It allows for faster startup times when performing manual tests, and it immediately reveals errors that can be detected using static analysis. Continuous execution of the application is less common for textual languages. One possible implementation of continuous execution are *programming with example* systems [Myers, 1990]. The examples provide the required input values for the re-execution of the application. Edwards [2004] implemented a programming with example system for object-oriented Java source code. Saff et al. [2004] used existing test cases to provide input for the continuous re-execution of the application. Their continuous testing tool provided only information about the test success of failure and no further runtime information, hence, it can detect defects but does not allow developers to inspect the runtime behavior to support the comprehension process.

Rehearse combines live coding with manual tests.

The tools discussed so far all require manual setup from developers to perform continuous executions. Rehearse [Choi et al., 2008] uses a manual program execution performed by a developer to set up a live coding environment. For the setup, developers specify which method will be defined interactively. When this function is first called during execution, the program halts and the developer can implement the method. Every line is now executed immediately after it was typed. However, to change previously typed lines

[1]https://developer.apple.com/library/mac/documentation/GraphicsImaging/Conceptual/QuartzComposerUserGuide
[2]https://cycling74.com/products/max/



```
fill(161, 219, 114);
for (var x = 40; x < 150; x += 50) {
    rect(x, 33, 20, 10);
    rect(x, 45, 20, 15);
    rect(x, 62, 30, 25);
}
```

Figure 6.1: Brett Victor's mockup of a live coding tool shows source code on the left and the resulting drawing on the right. Parameter values can be adjusted by dragging, the output changes immediately. Figure Source: http://worrydream.com/#!/LearnableProgramming

the developer has to undo all changes that happened after that line.

ALIVS Live! uses live coding to generate algorithm visualizations.

A similar tool, ALVIS Live!, was introduced by Hundhausen et al. [2007a]. To go back and forth between states of the execution, developers do not need to delete lines of code but use a playback control instead. In contrast to the tools discussed so far, ALVIS Live! is not primarily designed to provide developers with runtime information. Instead, it shows an algorithm visualization designed to teach programming to novices. After every newly entered line of source code, the visualization immediately updates.

Live coding is frequently used as a learning tool.

Recently, the idea of live coding regained attention after a highly successful talk by Brett Victor[3]. He showed a mock-up of a live coding editor for JavaScript drawing code, in which the generated drawing was displayed side-by-side with the source code and updated automatically after every change (see Figure 6.1). Later, Victor extended his ideas[4] to incorporate new visualizations for various kinds of data and he proposed to show one line of information per line of code. We adopted the latter idea in our prototype. Victor's designs inspired several tools, e.g., an online learning tool for novice programmers developed for the Kahn Academy[5] or the web development tool Lighttable[6] that allows inline inspection of all variables. This development is likely supported by the increase in available computing

[3]http://worrydream.com/#!/InventingOnPrinciple
[4]http://worrydream.com/#!/LearnableProgramming
[5]https://www.khanacademy.org/cs/programming
[6]http://lighttable.com/

power and the availability of powerful just-in-time compilers for scripting languages, as these technologies make continuous re-executions of an application feasible.

The behavior of developers using live coding is still relatively unknown.

In stark contrast to the multitude of tools that have been proposed, few results are available on the effect of working with a live coding environment for the developers' behavior. Snell [1997] reported on a preliminary study of a live coding environment according to which developers fixed more bugs during initial code entry and completed tasks faster than in traditional development environments, but he also warns that formal experiments are still missing. Wilcox et al. [1997] compared a live and a non-live version of Forms/3, an experimental spreadsheet environment. For bug fixing tasks, results were inconclusive and indicated that the effect of live coding depends on the task and the kind of bug. ALVIS Live! was used in a study to compare a version featuring live updates after every change to a version that required developers to refresh the visualization manually. In an evaluation with novice users, no difference in correctness of solutions could be found between these conditions [Hundhausen et al., 2007b].

Our work complements the corpus of studies on live coding environments. We are going to investigate how live coding affects the developers' coding behavior during code creation tasks and when working with a textual programming language.

6.2 How Live Coding Affects Developers Strategies

For our evaluation we required a live coding tool with interactive update rates.

Exploring how developers change their behavior when using a live coding tool for a realistic coding task was difficult in the past, because live coding tools have often been held back by a lack of processing power to return the results in time. Today, we have the tools available to implement a prototypical live coding environment that provides runtime information at interactive rates. In this section, we will introduce the prototype we have created to run our experiment before presenting the setup and results of our study. We will conclude this section with a discussion of the results and lessons learned from our experiment.

6.2.1 Prototype

Our prototype needed to fulfill two requirements to serve our purposes:

1. The prototype should provide a realistic experience of live coding, i.e., it needs to provide updated information at interactive rates.

2. Developers should be able to work without restrictions.

METIS is a fully functional live coding environment for JavaScript.

To fulfill these requirements, a mock-up with prerecorded information about certain code snippets was infeasible and the tool had to execute the code under development after every change. We implemented METIS, a live coding environment for JavaScript. We decided on using JavaScript because it can be executed without prior compilation. Further, JavaScript is the most popular language for the implementation of web-based applications, and a lot of effort is currently put into the creation of fast runtime environments for JavaScript, such as Google's V8[7]. We released METIS as an open-source prototype.

http://hci.rwth-aachen.de/liveCoding

METIS separates backend and frontend.

METIS is comprised of two parts: A backend server runs the JavaScript source code under development and records runtime information. A frontend displays the collected information visible to the user. In the following subsections, we will give an introduction to each component.

[7]https://developers.google.com/v8/

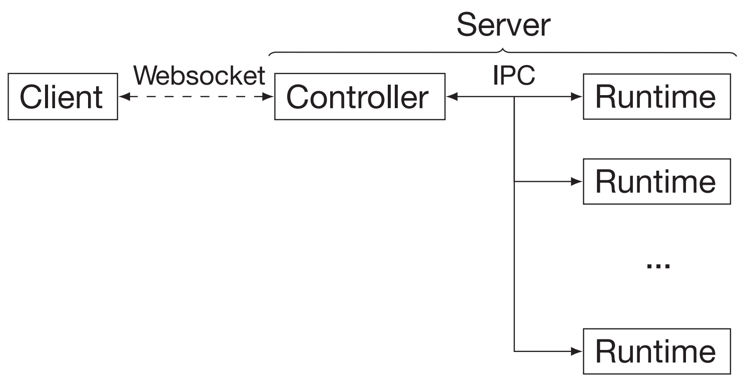


Figure 6.2: The backend communicates with clients via a WebSocket connection, and to individual sandboxes that execute versions of the code via the child process API in node.js. Figure adopted from [Belzmann, 2013].

Backend

The backend consists of a controller and sandboxed runtimes that execute the code under development.

The backend is responsible for executing a chunk of JavaScript source code and tracing the execution. It is comprised of two components: A *controller* communicates with the client and manages multiple *runtimes* that instrument and run the code. Separating these two parts allows to sandbox the execution of code under development, i.e., the code runs in a separate process. If the execution takes too long or does not terminate at all, the affected runtime can be safely killed without impacting the stability of the backend. The client communicates with the controller via a WebSocket connection, while the controller communicates with the individual instances of the runtime via the child process API of Node.js (see Figure 6.2). In the current version of the backend server, providing input to the running app is not possible. Still, in a future version this can be easily added by routing standard input and output streams of each runtime through the controller to the client. This would even allow the client to repeatedly replay a prerecorded interaction sequence.

To record runtime information, the backend instruments the code before execution. The instrumentation procedure adds callbacks to the source code that send variable values and other information about the code execution back to the backend controller where the information is chunked and sent to the client. We strive to insert the required callbacks without impacting the usual program behavior. To manipulate the original source code efficiently, the runtime first parses the source code to obtain an *abstract syntax tree* (AST). In the AST, each node represents a construct in the source code, such as a variable definition, a function, or a single literal. The types of outgoing edges of each node depend on the type of the node: For example, a literal node is always a leaf while a function node needs to have an outgoing edge referencing its body. All instrumentation is performed on the AST.

To record runtime information, the backend instruments the code.

Currently, the backend is capable of tracking information about the following events:

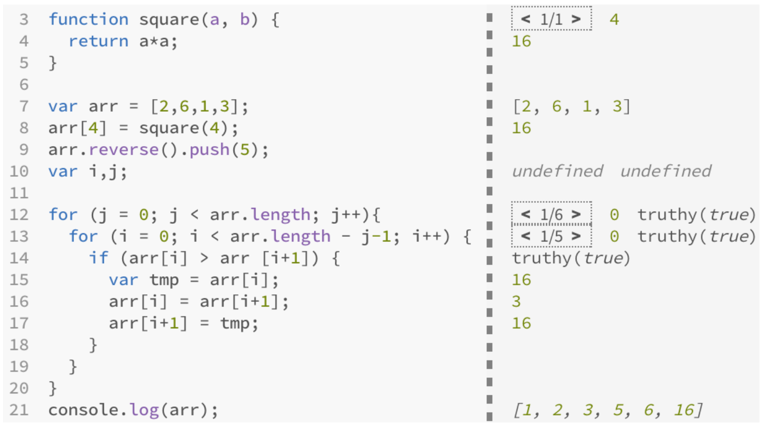
- assignments of variables
- function calls
- `if` and `switch` conditional statements
- `for`, `for-in`, and `while` loops
- error handling using `try-catch`
- uncaught exceptions
- log statements using `console.log`

A complete description of the instrumentation applied to the source code is part of Ewgenij Belzmann's Diploma thesis [Belzmann, 2013].

To test the efficiency of our runtime tracing backend we measured the time required to instrument different large JavaScript libraries. We found that the instrumentation takes 0.51ms on average ($SD = 0.29$) per line of code. Of course, the time required varies depending on the complexity of the source code.

Instrumentation takes less than 1 ms per line of code.

Next, we measured the performance impact of our instrumentation. For this test, we used three simple algorithms:



```
function square(a, b) {                    < 1/1 >  4
  return a*a;                              16
}

var arr = [2,6,1,3];                       [2, 6, 1, 3]
arr[4] = square(4);                        16
arr.reverse().push(5);
var i,j;                                   undefined  undefined

for (j = 0; j < arr.length; j++){          < 1/6 >  0  truthy(true)
  for (i = 0; i < arr.length - j-1; i++) { < 1/5 >  0  truthy(true)
    if (arr[i] > arr [i+1]) {              truthy(true)
      var tmp = arr[i];                    16
      arr[i] = arr[i+1];                   3
      arr[i+1] = tmp;                      16
    }
  }
}
console.log(arr);                          [1, 2, 3, 5, 6, 16]
```

Figure 6.3: METIS shows one line of runtime information next to each line in the source code editor. Figure from [Krämer et al., 2014]

Instrumented code runs significantly slower than the original version.

Merge sort and bubble sort to sort 4000 array elements, and the babylonian method to iteratively approximate the square root of a number using 100.000 iterations. Tracing the execution of these algorithms using our tool causes a significant performance decrease, with execution times of the instrumented code being slower than for the original code by a factor between 140 and 3200. This is caused mostly by the cost of message passing between the runtime, the controller, and the client. In our study, we managed to avoid this problem by designing our tasks appropriately. Hence, in our study the total delay between a change to the code and updated runtime information was always well below 500ms.

Frontend

METIS shows runtime information in a separate view on the right side of the source code editor.

We implemented the frontend for METIS as an extension for Brackets[8], an open-source JavaScript IDE. The METIS extension adds a separate area on the right side of the source code editor (see Figure 6.3). In this area, one line of information obtained from the backend is shown next to each line of code. Scrolling in the source code editor and the new view

[8]http://brackets.io/

is synchronized, thus, the runtime information for a line of code is always right next to the line in the editor. The design is adopted from Brett Victor's popular concept[9] of a live coding tool for applications without graphical output. Since this concept gained a lot of attention but was not yet evaluated, we considered it an interesting starting point for our evaluation. Based on Victor's concept, we tested several design concepts in informal user tests until we settled on the final design [Heinen, 2012; Kurz, 2013].

METIS can visualize all information provided by the backend.

METIS visualizes all information that can be provided by the backend. When visualizing the value of a condition, e.g., to provide information about an *if*-statement, METIS additionally shows if the value is truthy or falsy (i.e., evaluates to true or false in a Boolean context) in JavaScript (see line 12-14 in Figure 6.3). The distinction between truthy and falsy values is one of the common idiosyncrasies of JavaScript that is difficult to understand for novice JavaScript developers [Crockford, 2008]. For long values, such as long strings or complex objects, METIS abbreviates the visualization using "...". By clicking on the dots, users can expand the visualization to print out the string or object completely. To maintain the side by side view of source code and runtime information, Metis still displays the complete value in a single line and therefore requires horizontal scrolling.

Developers can inspect all executions of loops and functions.

For lines that are executed more than once, because they are inside of a loop or function, the execution to visualize can be selected using a selector next to the loop or function declaration (see line 1, 12, 13 in Figure 6.3). The selected execution is restored after live re-executions of the application, as long as the selected index still occurs in the new runtime trace. The selector supports scrubbing, i.e., clicking on the number and dragging the cursor left and right, for fast navigation through a large number of executions. The mapping between cursor movement and the selected execution is non-linear and designed to allow precise access to the first and last few iterations, as these are typical edge cases that might cause errors. If the execution fails due to an uncaught exception, METIS will show the exception in an overlay.

[9] http://worrydream.com/InventingOnPrinciple/

6.2.2 Study

We compared live coding with traditional development.

We want to analyze how working in a live coding environment affects developers' coding behavior. Our study used a between-groups design with two conditions: In the experimental condition, developers used METIS, the JavaScript live coding environment we developed; in the control condition, developers worked using a traditional development environment.

We expected developers in the experimental condition to be able to fix bugs right after they were introduced, because they do not need to switch to a dedicated debugging phase. We formulated three hypotheses about the behavioral changes we would observe:

Hypothesis 1 We call the time between introducing a bug and fixing it the *total fix time*. We expect the average total fix time to decrease in the experimental condition.

Hypothesis 2 As a consequence, we expect total task completion times to be lower in the experimental condition.

Hypothesis 3 We expect developers in the experimental condition to adapt a coding strategy in which code creation and bug fixing happen interleaved throughout the development process. In contrast, we expect developers in the control condition to perform a majority of bug fixing at the end of the trial. More precisely, we expect that editing activities are more evenly spread over the total time spent working on a task in the experimental condition.

Setup

Tasks were designed to circumvent limitations or our prototype.

Participants in our study had to solve three coding tasks that were designed to pose realistic challenges while circumventing the limitations of our prototype. All tasks had to be implemented in JavaScript for the Node.js JavaScript runtime.

Task 1 Developers should parse the RSS feed of daringfireball.net, a popular news website using the stream-based XML parser sax-js[10]. A stream-based XML parser reads the XML document serially and reports every node immediately. The challenges in this task were that developers had to use an unknown API, they had to work with an unfamiliar structure of the RSS feed, and they had to write asynchronous callback-based code.

Task 2 The developers should convert between two different date formats. This task was designed to provoke *programing slips*, i.e., small programming errors. They could be caused by mixing up different property names or by forgetting necessary conversions, e.g., from local time to UTC, or vice versa.

Task 3 The developers should implement Dijkstra's algorithm, an algorithm that calculates the shortest paths from one node in a graph to all other nodes. This task was challenging due to its algorithmic complexity.

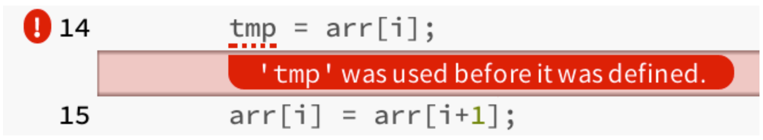
All participants worked on a 15-inch MacBook Pro with an external 23-inch screen and all required software pre-installed. Participants could choose Mac OS X or Windows as their operating system to allow them to use all shortcuts they were accustomed to. Participants in both conditions used Brackets[11]. In the experimental condition, we also enabled the METIS extension.

Participants worked on identical hardware but could choose their preferred operating system.

Live coding can partially replace two well-known features of modern IDEs: A debugger and continuous compilation. We provided both features in a consistent way in both conditions to isolate the effect of live coding from the established benefits of debuggers and continuous compilation.

In both conditions we provided continuous compilation and a debugger.

[10]https://github.com/isaacs/sax-js
[11]Sprint 24



```
tmp = arr[i];
'tmp' was used before it was defined.
arr[i] = arr[i+1];
```

Figure 6.4: Our continuous compilation plugin was available in both conditions. It shows error indicators next to the line number and indicates the position of the error in the line. Clicking the error indicator reveals a textual error description as shown here. Figure from [Krämer et al., 2014].

Participants could use the well-known Chrome debugger.

To provide a debugger, we installed node-inspector[12], a graphical debugger for Node.js applications. The debugger is mostly identical to the widely used debugger in the Chrome browser. We provided shortcuts that allowed participants to start the debugger without using the command line.

We created an extension for Brackets to provide continuous compilation.

Saff et al. [2003] reported that continuous compilation (for scripting languages this is equivalent to syntax checking or static error checking) increases the developers' performance. Live coding always implements continuous compilation as a side effect. Brackets includes JSLint, a style and syntax checker for JavaScript, to provide continuous compilation, but it is only invoked on file save and not continuously. Hence, we developed a continuous compilation extension for Brackets, that runs JSLint after every change and shows errors inline in the source code (see Figure 6.4). This extension was available in both conditions. It was released as an open-source project alongside METIS.

We provided templates as starting points including calls to the methods to be implemented to enable live coding.

We created templates as a starting point for every task. They included references to all required libraries and a definition of the method participants needed to implement. The template included calls to this method and documentation about the expected return value for these invocations, to allow participants to test whether or not their code was correct. Participants were allowed to use any external resource they found useful, e.g., code snippets from the web.

[12]https://github.com/node-inspector/node-inspector

They could also use third-party libraries they found online except for replacements for sax-js in Task 1.

We recorded time-stamped revisions of the code and videos of participants' screens.

During the study, participants were filmed and their screen was recorded. We also stored time-stamped revisions of the code after every change (on a character level) to the source code. There was no time limit to complete the tasks but participants could stop working on a task at any point. If they took a considerable amount of time without narrowing down a solution, the experimenter would ask the participant whether they want to stop. Participants received 25€ in compensation for their expenditure of time.

Results

We recruited students with prior experience in JavaScript.

When we recruited participants, we posted an online JavaScript self-assessment to allow developers to test whether or not they had the necessary experience to solve the tasks in our study successfully. However, we did not check if the participants completed this self-assessment or how they scored, accordingly, the study was open for everyone. Since overly huge differences in the developers' experience have been shown to potentially mask effects between conditions [R. Brooks, 1980; Sackman et al., 1968], we decided to remove outliers that had severe problems solving the task. After the outlier removal, we analyzed data from 10 participants (1 female, 9 males). All participants were students of computer science. They reported to have an average of 13.4 years ($SD = 9.23$) experience programming and an average of 1.25 years ($SD = 5.23$) experience programming with JavaScript. Participants engaged in programming for 13.0 hours ($SD = 13.0$) per week.

We annotated the introduction and fix of each bug in the time-stamped version history of each trial.

We first tested Hypothesis 1 and compared the average total fix time of bugs, i.e., the total time between the introduction of a bug and its fix, between both conditions. One researcher analyzed the introduction and fixing of each bug in the time-stamped version history of each participant's solution. We annotated 205 bugs in all trials.

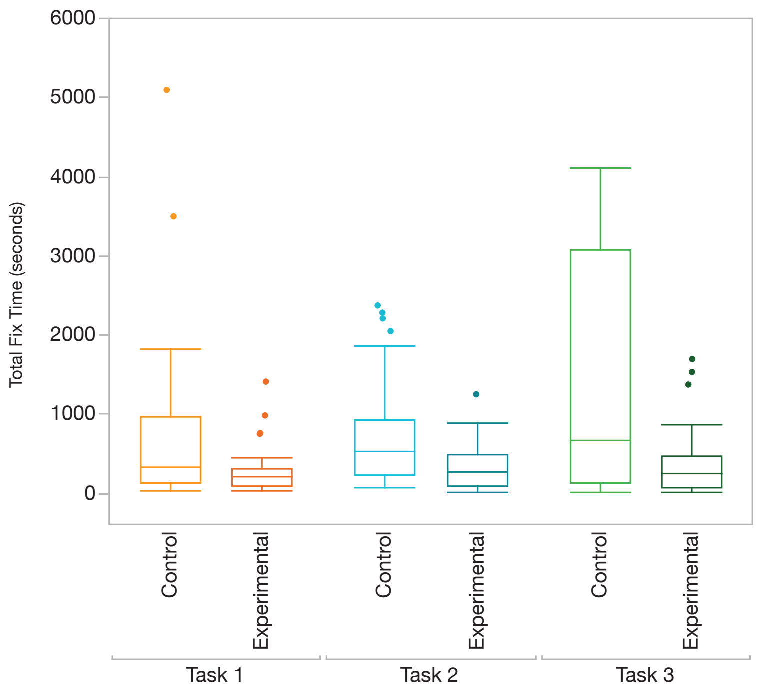


Figure 6.5: The total fix time decreases significantly in the experimental condition in all tasks.

We used a heuristic to not count planned modifications as bugs.

Any change to the source code that led to erroneous behavior in the context of the current source code version was counted as a bug. This potentially includes false positives: During planned modifications, the source code will likely be erroneous for a short time. To compensate for this error, we removed bugs that were fixed within 30 seconds after their introduction, unless unrelated code was written between the introduction of the bug and its correction. Again, this heuristic is not perfect and might filter out actual bugs that happened to be fixed within 30 seconds. As a result, the measured average total fix time will increase. Because we expect shorter total fix times in the experimental condition, where participants had live updating information available to identify bugs, the filtering mechanism effectively benefits the control condition.

Average total fix time decreased when using live coding.

To compare the average bug fixing time, we used a mixed linear model that included task, condition, and their interaction as fixed factors and the allocation of participants to conditions as a random factor. We found a significant decrease of the average total fix time in the experimental condition (see Figure 6.5).

Task:	$F(2, 197.1) = 2.635$	$p = 0.074$
Condition:	$F(1, 9.124) = 6.941$	$\mathbf{p = 0.027}$
Interaction:	$F(2, 197.1) = 2.025$	$p = 0.135$

This result confirms Hypothesis 1 and shows that developers fix bugs significantly faster when using live coding.

The number of bugs per trial was similar in both conditions.

We verified that the number of bugs per participant was similar in both conditions using a repeated-measures ANOVA with the task as within-groups factor and the condition as a between-groups factor. No effect of either condition was found. The effect of task is very close to being significant, but since the tasks are not designed to be similar, this result is not surprising.

Task:	$F(2, 22) = 3.429$	$p = 0.051$
Condition:	$F(1, 24) = 1.826$	$p = 0.189$
Interaction:	$F(2, 22) = 0.541$	$p = 0.590$

No effect of condition on task completion time was found.

Next, we tested Hypothesis 2: We expected that decreasing the average total fix time would also result in a decreased task completion time. Using an ANOVA to compare the task completion times between the different conditions (see Figure 6.6), we found no significant effect of the condition on the task completion time. Hence, we cannot confirm Hypothesis 2.

Task:	$F(2, 16) = 12.11$	$\mathbf{p = 0.006}$
Condition:	$F(1, 8) = 2.794$	$p = 0.133$
Interaction:	$F(2, 16) = 0.162$	$p = 0.851$

Condition had a significant influence on the distribution of edit activity over time.

Lastly, we tested if developers would adapt a different strategy in the experimental condition. More specifically, we suspected that developers working in a live coding environment would switch between creating new code and correcting code more frequently, and that phases in which

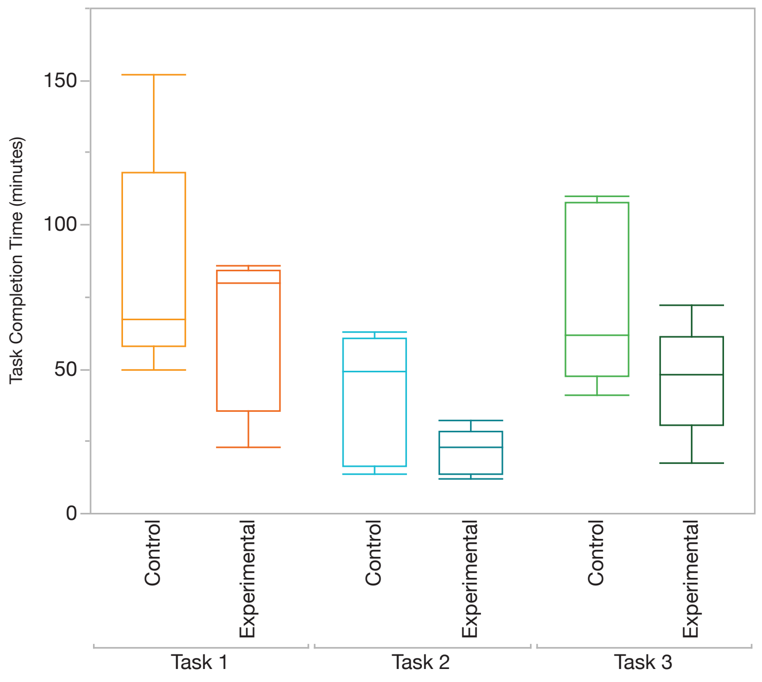


Figure 6.6: We found no significant decrease of task completion time in the experimental condition. We assume that this is caused by a relatively low sample size in our study.

code correction is performed are more homogeneously distributed over the course of the trial. To compare the temporal distribution of edits during the trial between conditions, we used a Kolmogorov-Smirnov test. We found that these distributions differ significantly between conditions ($D = 0.05, p < 0.001$). This confirms Hypothesis 3.

Questionnaire Results

No major usability flaws in METIS were reported.

We asked participants in the live coding condition to fill out a questionnaire after the trial, to gather informal feedback from participants. The questionnaire included a System Usability Scale test to confirm that no major usability flaws confounded the results of our study. METIS scored 81.4 on average ($SD = 10.7$) on this scale, which is an excellent result, according to the interpretation of the System Usability Scale presented by Bangor et al. [2008].

Participants state that working live gives them more confidence in their code.

All but one participants agreed or strongly agreed that they found live coding helpful for code understanding. Everyone agreed that working with the live coding environment gave them more confidence that their code is correct. For each task, participants using METIS agreed that this tool helped them to solve the task.

Discussion

Shorter total fix times did not result in a decrease of task completion time.

The analysis of our study confirms our first hypothesis: Developers fix bugs faster when using live coding compared to a traditional development environment. As a result of this effect, we expected task completion times do decrease in the experimental condition as well (Hypothesis 2). However, we could not confirm this statistically.

Our results could be influenced by a small sample size.

One reason for being able to confirm Hypothesis 1 but not Hypothesis 2 could be the statistical methodology: When testing Hypothesis 1, the units of analysis were the annotated bugs, allowing us to work with 205 samples. In contrast, when testing Hypothesis 2, the units of analysis were participants, hence, we only worked with 10 samples. This lower sample size causes small effect sizes, even if the effect exists, to be difficult to detect using a parametric test. Small sample sizes also cause inter-subject differences to be more likely to overshadow underlying effects. Our data suggests that this is a likely explanation for the non-significant results: The mean task completion time of each task is lower in the experimental condition than in the control condition, however, the standard deviations are in the same order of magnitude as the means (see Figure 6.6).

Developers used two different strategies.

In the experimental condition we found that edits by developers are more evenly distributed over time compared to the control condition (Hypothesis 3). This again indicates a change in the developers' strategies. We qualitatively could differentiate two strategies that participants chose to solve the tasks:

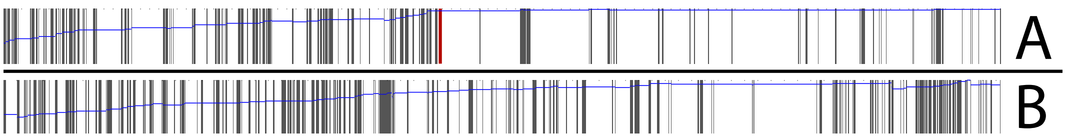


Figure 6.7: Each graph shows the edit activity of a developer over time. Developer A uses the sequential strategy, i.e., in the first phase many edit activities occur, while in the second phase long pauses between edits indicate debugging phases. Developer B uses the interleaved strategy with a more homogenous distribution of edits over time. Figure Source: [Krämer et al., 2014]

Sequential Strategy Participants using the sequential strategy started by implementing the complete application and then tried to fix all bugs in one debugging session.

Interleaved Strategy Participants using the interleaved strategy tested and verified each incremental change to keep the source code correct throughout the whole development process. This strategy corresponds to performing numerous, short edit-test-edit cycles.

When using the interleaved strategy, edit activities are evenly spread over time.

Figure 6.7 shows examples for both strategies from our study. Developer A, who is at the top, uses the sequential strategy and two phases can be visually discriminated: In the first phase, pauses between edits are short because the developer hardly ever stopped creating new code. At the same time, the length of his source file is monotonically increasing. In the second phase, edits only happen in short bursts, because the developer mostly works with the debugger to identify causes for bugs that he or she then fixes in short edit phases. For developer B, who is at the bottom, no phases can be discriminated: He or she uses the interleaved strategy and edits are uniformly spread over time.

Developers pick strategies depending on task and condition.

Which strategy developers adopt depends on the task and the condition. All participants in both conditions adopted the interleaved strategy when working on the first task. When working on the second and third task, all but one (two in the third task) participants used the sequential strategy in the control condition, while all participants in the experimental condition adopted the interleaved strategy. We assume that the first task was particularly easy to split up into smaller parts that could be tested individually, and therefore promoted performing multiple edit-test-edit cycles.

To be able to understand the effect of the different strategies, we re-ran our previous analysis of task completion times and included the strategy used as a separate factor. To exclude the first task, in which all developers used the interleaved strategy, we performed separate analyses for each task. For the second task, we found a significant reduction in task completion times in the experimental condition and for the interleaved strategy.

In some tasks adopting the interleaved strategy, which live coding encourages, can reduce task completion time.

$$\begin{array}{rll} \text{Condition:} & F(1,6) = 13.19 & \mathbf{p = 0.01} \\ \text{Strategy:} & F(1,6) = 7.41 & \mathbf{p = 0.03} \end{array}$$

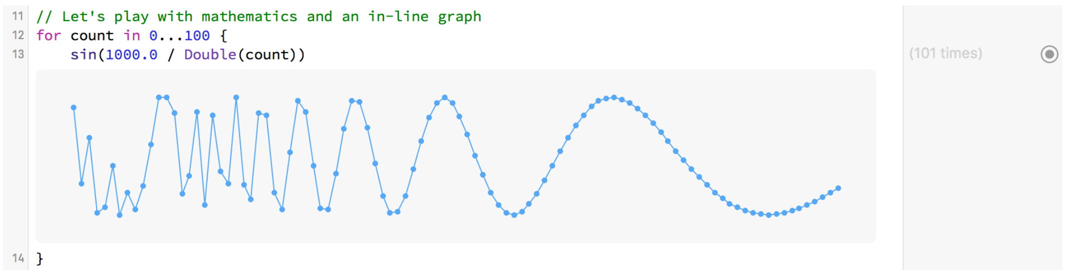
No significant effects were found for the third task. We conclude that adopting the interleaved strategy can improve completion times for certain tasks, even when performing manual edit-test-edit cycles. In summary, live coding environments encourage the adoption of the interleaved strategy.

6.2.3 Limitations and Future Work

We presented a first evaluation of the effect live coding environments have on the coding strategies developers exhibit. Our study is only a first step towards the perfect understanding how developers utilize these environments, though. A longer exposure to a live coding system could result in developers learning how to incorporate the tool into their workflow best. For future work, we propose to study developers using a live coding system for an extended period of time, ideally for their usual daily programming tasks.

Developers behavior might change further when learning how to use live coding over a longer timespan.

The development of live coding environments also leaves several open questions for future work. METIS is a prototype of a live coding environment that was developed specifically to allow us to run the study described above.



```swift
// Let's play with mathematics and an in-line graph
for count in 0...100 {
    sin(1000.0 / Double(count))
}
```

Figure 6.8: Xcode playgrounds combine a single line of runtime information in the right column with optional bigger visualizations that are presented inline with the source code.

Future work is needed to further improve the design and implementation of live coding tools.

Our design aimed to visualize all runtime information that we can capture. This design is useful to explore the fundamental effects a live coding environment has on a developer's work habits. However, we found several limitations both of the interface design of METIS and of the practical applicability of our implementation of live coding. We will discuss both of these problems in the following section.

Design Limitations

One line of information per code line impedes the visualization of complex objects.

To allow for a natural mapping between the source code and the information visualized in METIS, we presented no more than one line of runtime information per line of code. This impedes the effective visualization of complex objects, such as the HTML source of a website or a complex object. More effective visualizations are also required for applications generating graphical output.

Our visualization did not handle chains of method calls in one line.

The space limitation on information is also problematic for chains of mutating functions. For example, the line `arr.reverse().push(5)` reverses the array `arr` and then appends the value `5`. Both methods work in-place, i.e., the result of the expression is stored in `arr`. METIS would show no information for this line, because it is unclear if the user is interested in the value of `arr`, the return value of `arr.reverse()`, or in the result of the complete expression. In our study, developers often added additional calls to `console.log`, to output the information they required.

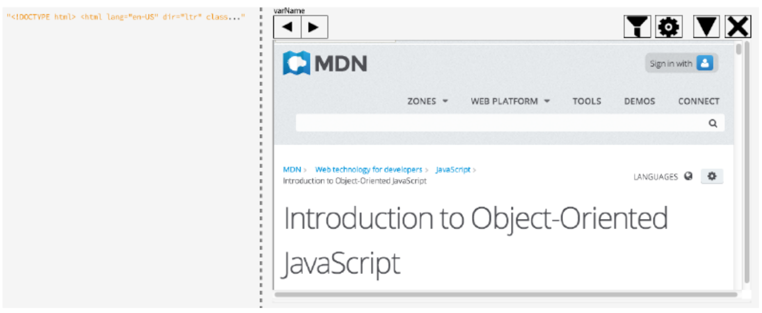

```
"<!DOCTYPE html> <html lang="en-US" dir="ltr" class..."
```
varName
MDN
Sign in with
ZONES
WEB PLATFORM
TOOLS
DEMOS
CONNECT
MDN › Web technology for developers › JavaScript ›
Introduction to Object-Oriented JavaScript
LANGUAGES
Introduction to Object-Oriented JavaScript

Figure 6.9: We proposed a three-column design, in which the right column can be used for detailed visualizations of runtime data. In this screenshot, the leftmost column that contains the source code is not shown. The central column shows one line of runtime information for an HTTP GET request that returns the complete HTML source code of a website. A user configured a widget to be displayed in the third column that renders a realistic preview of the HTML code. Figure Source: [Wolf, 2014]

Dedicated visualizations for complex runtime information can be opened on demand.

In the future, we propose to allow users to open visualizations that extend beyond a single line for specific runtime information. One approach to solve this problem is implemented in Xcode playgrounds feature[13]: For every value displayed next to the source code, a developer can open a larger visualization area that is inserted inline below the corresponding line of code (see Figure 6.8). The downside of this approach is that the opened visualizations disrupt the source code. This might impact its readability because it alters the code's visual gestalt.

We explored a three-column design with one column being dedicated for on-demand visualizations of complex runtime information.

We explored a different approach to provide richer visualizations for individual lines of code on demand [Wolf, 2014]. We propose to use a *probe metaphor* that allows the user to attach a probe to a specific variable or line of code. For this purpose, we introduce a third area on the right side of METIS. The user can select individual values in the central column, which shows the current information visualization, to open them in a larger visualization area in the new column (see Figure 6.9). These larger visualizations are highly customizable to suit the user's needs: He or she

[13]https://developer.apple.com/swift/blog/?id=24

can choose between different visualizations, e.g., an array of numbers could be visualized as a table or as a diagram. It is also possible to apply transformations to fully customize the visualization, e.g., using the `map` function to transform an array of objects into an array of string representations of these objects. A thorough evaluation of this design is left for future work.

For some coding activities automatic updates were disturbing.

Another visualization problem we found when analyzing METIS was, that users felt distracted by the continuous updates when working on a conceptually challenging chunk of code. The obvious solution is to allow developers to turn off live updates when required. However, more experiments are needed to analyze if live coding can still promote the more effective interleaved strategy when it is not activated constantly.

Live coding environments need to provide input the application to exercise a relevant code path.

METIS does not offer any means to provide input for the application under development. As discussed before, the backend could technically be extended to route the input and output streams of the application under development to the client. It is infeasible, however, to have the user repeatedly provide the necessary input to exercise a certain point in the source code. For future work, we propose two approaches to solve this problem:

1. If test cases for the software exist, test cases can be used to exercise specific portions of the source code. By using runtime analysis, a test that exercises the part of the source code that is currently being edited can be automatically selected [Kiel, 2009]. This approach is particularly promising, if a test-first development strategy is already applied. If multiple tests exist to describe multiple requirements for the same part of the source code, the interface could even enable developers to experiment with various inputs. The downside of this approach is that it requires developers to provide the execution context for their code manually in the form of a test. This approach is similar to several existing research projects, which we have already discussed in Section 6.1. Automated testing tools that generate a test harness for software

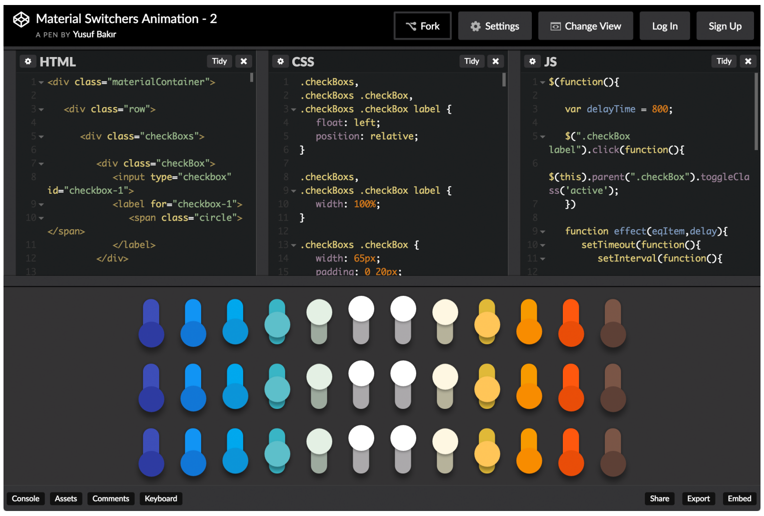
automatically [Fröhlich et al., 2000] could help reducing the required user interaction.

2. The live coding tool could allow the user to provide input to the application manually and store this input to recreate it for future executions of the code. This approach is particularly suitable for applications with a user interface of some kind [Burg et al., 2013]. To combine this approach with the previous one, the manually performed test of the application could even be stored in the form of a unit test. In this case, the system would provide a programming-by-demonstration [Cypher, 1993] interface for unit tests. This idea was implemented prototypically in Tanja Ulmen's Bachelor's thesis [Ulmen, 2014] and ultimately inspired the design of VESTA, which we have discussed elaborately in Chapter 4.

Technical Limitations

Our backend impeded performance and was not sandboxed in a way to prevent accidental resource access.

An application being developed in a live coding environment not only requires input for every re-execution but also resources, i.e., system memory or files on a hard drive. Accessing these resources repeatedly can yield unexpected side effects, e.g., a developer might accidentally delete files while working with APIs for file access. Similarly, if the application is doing memory or processing intensive calculations, a developer might experience severe performance issues when the application is re-executed after every keystroke. These issues might be addressed with sandboxing and optimization of re-executions on the level of JavaScript's just-in-time compiler. In the next section, we are going to present an alternative solution based on an interaction design that allows to selectively perform live coding for small code fragments.



```html
<div class="materialContainer">

   <div class="row">

      <div class="checkBoxs">

         <div class="checkBox">
            <input type="checkbox"
id="checkbox-1">
            <label for="checkbox-1">
               <span class="circle">
</span>
            </label>
         </div>
```

```css
.checkBoxs,
.checkBoxs .checkBox,
.checkBoxs .checkBox label {
   float: left;
   position: relative;
}

.checkBoxs,
.checkBoxs .checkBox label {
   width: 100%;
}

.checkBoxs .checkBox {
   width: 65px;
   padding: 0 20px;
```

```javascript
$(function(){

   var delayTime = 800;

   $(".checkBox
label").click(function(){

$(this).parent(".checkBox").toggleCla
ss('active');
   })

   function effect(eqItem,delay){
      setTimeout(function(){
         setInterval(function(){
```

Figure 6.10: Codepen.io is an online experimentation environment for HTML, CSS, and JavaScript. After every change to the code on top, the output below is automatically reloaded.

6.3 Fiddlets: Live Coding for Small Code Fragments

We propose to use live coding for small code fragments as needed.

Frequent re-executions of an entire application are often technically impracticable. Hence, we propose to only apply live coding to small code fragments that contain the currently edited line. This line should be executed in a context that is similar to its actual context when the complete application is executed.

Online experimentation tools allow tinkering with small code fragments but are disconnected from the IDE.

Currently, developers can execute small code fragments using online experimentation environments, such as jsfiddle[14], jsbin[15], or codepen[16] (see Figure 6.10). While the various experimentation environments differ slightly, they still share these fundamental properties: They allow to edit HTML, JavaScript, and CSS files for a single web site. The output is shown next to the code editors and can either be refreshed manually or automatically after each change.

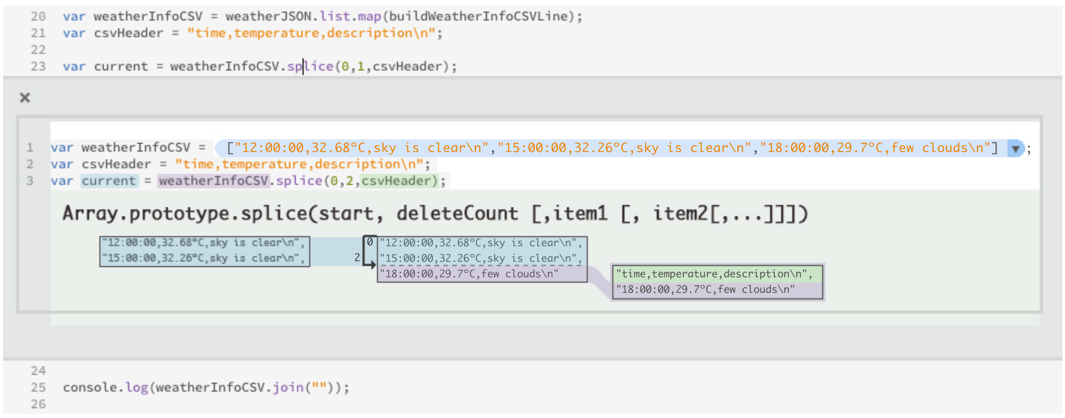
[14] http://jsfiddle.net
[15] http://jsbin.com
[16] http://codepen.io

Typically, inspection of runtime information is not possible, except for manually logging intermediate results to a console. The biggest problem of experimentation environments is that the user needs to set them up manually. This involves implementing a context for the part of code that users want to experiment with. Users can either create that context from scratch or by selectively copying code fragments from the source application. In practice, these online tools are used primarily for communication and sharing code, e.g., a solution to a specific problem [Squire et al., 2015].

Importing all classes of the software under development does not provide enough context for experimentation.

An alternative tool for experimenting with a source code in real time is a *Read-Eval-Print-Loop (REPL)*. This tool allow developers to enter a line of code and execute this line immediately, similar to a text chat in which the environment answers with the execution result of the executed line. In its most basic form, it suffers from the same problem as online experimentation environments, i.e., the user needs to set up a context manually for running the code he or she is interested in. To mitigate this problem, some tools load all classes that are part of the application, hence, the user can access the existing code. This is implemented, for example, in the development console of Ruby on Rails[17]. While this is useful sometimes, it still leaves several problems open: The imported classes still need to be initialized manually, which is especially hard if parts of the application are supposed to be set up by the system when the application launches. The code that users want to edit live will often be located in one of the imported classes, however, those are only loaded but cannot be edited from the REPL. Further, the code under development can be located in a block that is executed asynchronously, in response to an external event, e.g., a HTTP request. In consequence, actually exercising the relevant code path can still be challenging in a REPL or live coding environment.

[17]http://rubyonrails.org



```
20  var weatherInfoCSV = weatherJSON.list.map(buildWeatherInfoCSVLine);
21  var csvHeader = "time,temperature,description\n";
22
23  var current = weatherInfoCSV.splice(0,1,csvHeader);
```

```
1  var weatherInfoCSV =  ["12:00:00,32.68°C,sky is clear\n","15:00:00,32.26°C,sky is clear\n","18:00:00,29.7°C,few clouds\n"];
2  var csvHeader = "time,temperature,description\n";
3  var current = weatherInfoCSV.splice(0,2,csvHeader);
```

```
Array.prototype.splice(start, deleteCount [,item1 [, item2[,...]]])

"12:00:00,32.68°C,sky is clear\n",      0 "12:00:00,32.68°C,sky is clear\n",
"15:00:00,32.26°C,sky is clear\n",      2 "15:00:00,32.26°C,sky is clear\n",
                                          "18:00:00,29.7°C,few clouds\n"
                                                                           "time,temperature,description\n",
                                                                           "18:00:00,29.7°C,few clouds\n"
```

```
24
25  console.log(weatherInfoCSV.join(""));
26
```

Figure 6.11: Fiddlets shows an inline editor for the live execution of a part of the source code. In this screenshot, the developer wants to tweak the parameters for the splice function in line 23. The context for this line inferred by Fiddlets is shown at the top of the inline editor. The bottom half of the inline editor is a custom visualization for the output of the splice function. Figure Source: [Lewandowski, 2015]

6.3.1 Design and Implementation

Fiddlets provide a realistic context for working on a line of code live.

We propose to moderate the problems of online experimentation environments and REPLs by providing a potential context for a part of the source code automatically. Our prototypical implementation of this idea is called Fiddlets [Lewandowski, 2015]. Like METIS, it is implemented as a plugin for Brackets. Fiddlets are inline editors that can be invoked for any line of source code, called the *source line*. A Fiddlet allows to experiment with this line of code in an live inline editor that is pre-populated with a realistic context for the source line.

The context suggested by Fiddlets can be customized by the developer.

An example of a Fiddlet is shown in Figure 6.11. In this example, a developer is interested in tweaking the parameters for the splice function in line 23. The Fiddlet has inferred that the context for this line needs to include the variables `weatherInfoCSV` and `csvHeader`. These variables are copied over to the context of the source line, which is shown in the top part of the Fiddlet. The variables are initialized with values that are realistic and could occur in an actual execution of the program. For the `weatherInfoCSV` variable, multiple possible initializations are available and the user can switch between them. The context can also be manually edited by the developer. The bottom half of the Fiddlets shows the output of the source line, using a visualization that is tailored to the splice function. If no function-specific visualization is available, Fiddlets shows the return value of the function call.

Fiddlets are inspired by Codelets.

The design of Fiddlets is inspired by Codelets [Oney et al., 2012a]. Codelets are inline editors that allow the customization of inserted code snippets. Creators of snippets can completely customize the editor provided by Codelets to support the user in understanding and using the snippet. They were found to greatly improve the usefulness of snippets compared to the same snippets being available as part of an API documentation. Using Codelets, developers could solve certain tasks involving a set of snippets significantly faster than without it.

Fiddlets uses dedicated visualizations depending on the function being edited.

The visualization used in Fiddlets is also custom designed for the specific function being edited. In contrast to Codelets, though, this function is executed live in a context that is similar to its context in the actual application. This allows Fiddlets to be used not only for creating new code but also for changing, correcting, and finally understanding existing code.

To create the context for the source line, two techniques are used:

1. Fiddlets use static analysis to set up the context for the source line if possible. The static analysis algorithm first identifies all variables used in the source line and then adds them to a variable list. For each of these variables, the static analyzer searches all usages of the variable before the source line that change it or initialize it. If the variable is initialized from a literal, this initialization is copied to the context and the variable is removed from the variable list. If the variable is initialized from an expression, the analyzer adds all variables in this expression to the variable list and searches initializations for these variables as well. While this static analysis method seems to be re-

source intensive, it is limited to search in the scope of the source line and hence only works in a rather small part of the source code.

2. If no initialization from a literal was found for a variable, Fiddlets searches for values of the variable in runtime traces. A runtime trace can store the values of each variable at any point of the execution (for more details on runtime traces and their uses in software development environments see Chapter 4). Fiddlets uses runtime traces both from manual executions performed by the developer and from unit tests.

Developers can pick different possible initializations for each variable.

Fiddlets show all possible initializations for a variable it has found and allows users to choose between them. This enables testing the behavior of the source line in various scenarios.

6.3.2 Evaluation

In the evaluation of Fiddlets, we wanted to find out if the significantly reduced live execution capabilities are still sufficient enough to provide a benefit to developers. We explored two different hypotheses:

1. Developers will replace manual edit-test-edit cycles with usage of Fiddlets.
2. Developers can solve implementation and bug-fixing tasks faster when using Fiddlets than when performing manual edit-test-edit cycles.

Setup

We carried out a between-groups study with five tasks.

To explore these hypotheses, we carried out a between-groups study to compare developers using either Fiddlets or an unmodified version of Brackets. Participants had to solve five tasks, all of which required an implementation or change of single line of code using a given function. For example, participants had to provide the parameter for a regular expression that was required in the code, or they had to change the parameters of array manipulation functions such as split or splice. All tasks demanded the understanding of both, how the function being used works and with which values it is used. We created the tasks by changing the source code of mustache.js[18], a widely-used JavaScript templating engine. For example, we deleted a line that the participants needed to implement, or we introduced a bug into a line.

The Fiddlets prototype used a mockup of the proposed runtime analysis.

For the purpose of our study, we created a prototype of Fiddlets that supported all functions used in our study. The static analysis method to find variable initializations was implemented completely, but the runtime analysis technique was mocked up by reading runtime information from pre-recorded traces. A complete description of all individual Fiddlets and tasks is included in the Master's thesis by Dennis Lewandowski [2015].

Results

We removed two participants as outliers.

We recruited 14 participants (12 males, 2 female) for our study, two of which were removed as outliers after the study. One outlier (in the Fiddlets condition) only completed one of five tasks successfully, the other outlier (from the control condition) had unusually long task completion times (more than twice as long than the group average in two tasks). We believe the outliers were indicative of a huge variance in JavaScript experience among participants. For the remainder of this section, all presented results exclude the outliers.

Participants in the control condition were more experienced with JavaScript.

All participants were students, except for one IT consultant. They were 26.5 years old on average ($SD = 8.856$). Participants reported to be experienced with JavaScript for 3.8 years on average ($SD = 3.6$), with responses ranging from 1 year to 10 years. Plotting JavaScript experience by condition (see Figure 6.12) reveals that by randomizing partici-

[18]https://mustache.github.io/

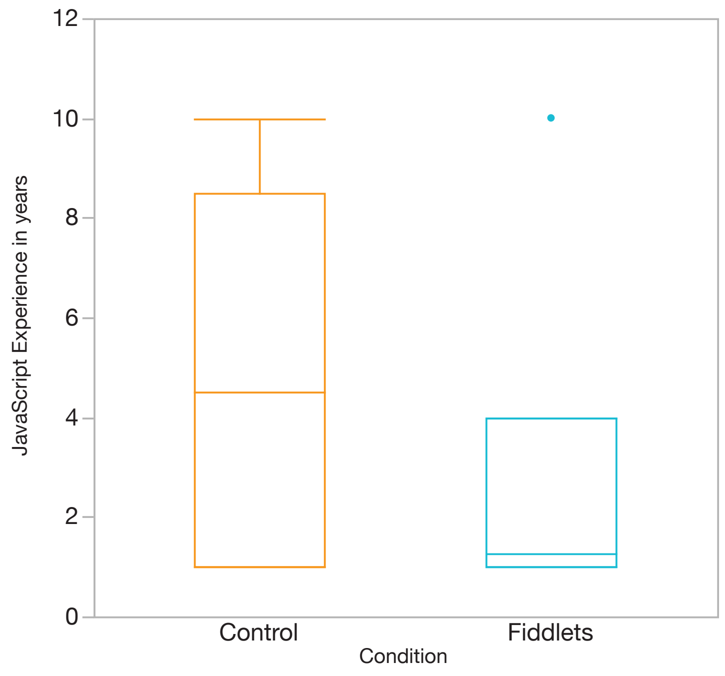


Figure 6.12: By randomizing participant to condition assignment we ended up with participants in the control condition being significantly more experienced than those in the Fiddlets condition.

pants to condition assignments we ended up having participants in the control condition that were substantially more experienced with JavaScript than those in the Fiddlets condition.

Participants opened a median of five Fiddlets during each task.

To analyze hypothesis 1, we used two metrics to characterize the adoption of Fiddlets: The number of times participants invoked Fiddlets per task, and the percentage of the total task completion time during which each Fiddlet was opened. The median count of Fiddlets invocations per task was 5, with a maximum of 45. Fiddlets remained open for an average of 21.7% ($SD = 15.8\%$) of the task completion time. The Fiddlet that remained open for the longest was open for an average of 52.4% ($SD = 27.7\%$) of the task completion time (see Figure 6.13). Half of the participants had a Fiddlet open for over 90% of the task completion time at least for one task.

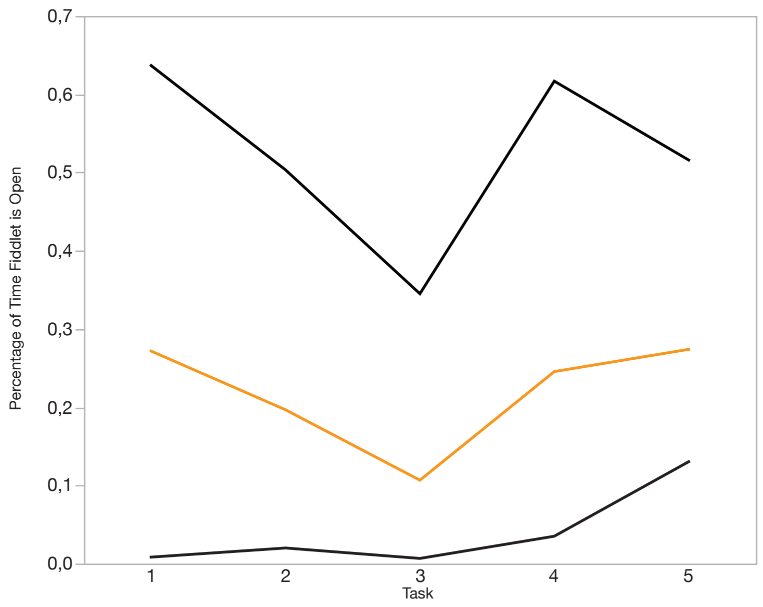


Figure 6.13: The figure shows the percentage of time, relative to the total task completion time, a Fiddlet remained open. The upper and lower lines indicate the average maximum and minimum values, the orange center line is the total average.

To analyze hypothesis 2, we compared task completion times (see Figure 6.14) of participants solving each task correctly using a repeated-measures ANOVA, modeling task as a within-groups factor and condition as a between-groups factor. We found a significant effect of task and a nearly significant effect of condition.

Participants in the control condition tend to solve tasks faster.

Task:	$F(4, 34.26) = 6.67$	$\mathbf{p < 0.001}$
Condition:	$F(1, 11.6) = 4.39$	$p = 0.059$
Interaction:	$F(4, 34.26) = 0.68$	$p = 0.609$

Participants in the control condition tended to be faster than those in the Fiddlets condition. Inspecting the data reveals that only for Task 5 the task completion times are substantially different, for all other tasks 95% confidence intervals overlap. In a second analysis, we tried to compensate the differences in JavaScript experience by using Task

When using one task for calibration, participants in the Fiddlets condition tend to be faster.

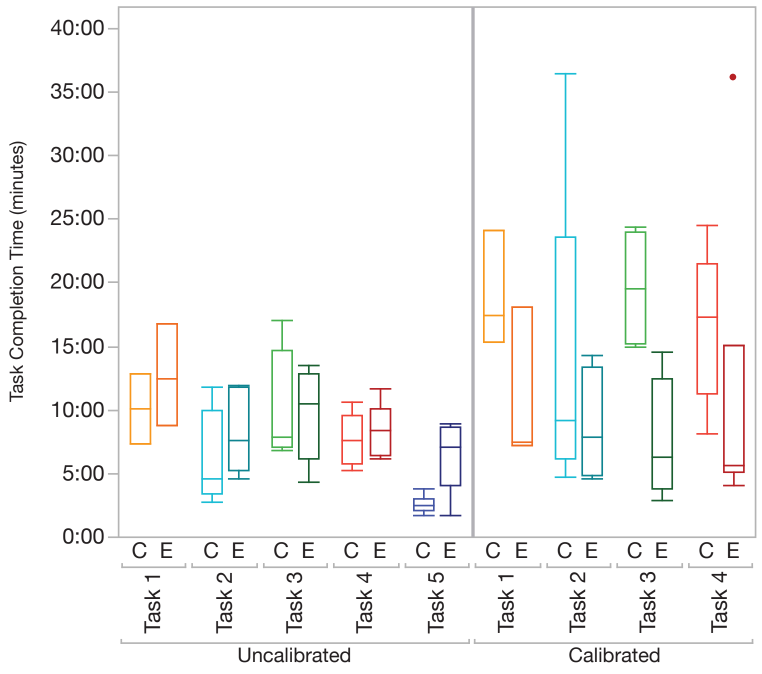


Figure 6.14: Task completion time was higher in the experimental condition (E) than in the control condition (C). When calibrating for inter-subject differences using Task 5 as a calibration task, the effect reverses.

5, in which we previously found the most striking difference, as a calibration task. More precisely, we calculated a correction factor $C_{exp} = TCT_5/Avg(TCT_5)$ for each participant, where TCT_5 is the participant's task completion time in Task 5 and $Avg(TCT_5)$ is the average task completion time in Task 5 for all participants. We then scaled the task completion time for all other tasks by dividing it by the correction factor. Using the scaled task completion times, we again found a nearly significant difference between conditions and lower corrected task completion times in the Fiddlets condition by a factor of 1.8.

Task:	$F(3, 19.82) = 1.30$	$p = 0.301$
Condition:	$F(1, 9.06) = 3.64$	$p = 0.089$
Interaction:	$F(3, 19.82) = 0.27$	$p = 0.849$

Discussion

Fiddlets were used as probes all around the code.

We found that participants quickly adopted Fiddlets for their work. They not only used Fiddlets to use live coding in the line they needed to edit, but also as probes all around the source code. Fiddlets allowed developers to explore the context of the line they needed to change in much detail without using a debugger. Participants often kept the Fiddlet for the line that needed to be changed open for an extended period to regularly check if they had solved the task correctly.

Task completion time of inexperienced developers using Fiddlets was on par with experienced developers without Fiddlets.

Inexperienced developers using Fiddlets could solve tasks as fast as experienced developers without Fiddlets. However, the analysis of task completion times was overshadowed by inter-subject differences. This was also apparent when observing participants while working on their tasks. Inexperienced developers often spent a considerable amount of time understanding how the function they had to use was useful to solve the task at all. In contrast, experienced developers spent most of their time finding the context in which the function they needed to implement is executed, i.e., how data that is passed in is formatted. After that, they typically found the correct parameters of this function quickly, even if it required some experimentation, e.g., to find a correct regular expression. The first step is vastly improved with Fiddlets, because Fiddlets automatically find examples of data that is used in the application. However, we observed that developers needed to get used to trusting those examples.

Our preliminary results are promising.

Overall, this preliminary evaluation proved that Fiddlets is a promising interaction technique in order to provide live coding without the need to execute the complete application repeatedly. Generating a realistic context for the execution of a single line using a mix of static slicing and cached runtime information seems to be an effective approach to generate a complete context in many scenarios. Fiddlets also carry over the key benefit of Codelets, i.e., they provide a visualization tailored to the currently edited function call. This allows Fiddlets to effectively communicate information about complex functions. A complete implementation of Fiddlets runtime analysis and an evaluation with similarly sampled participant groups are interesting directions for future work.

6.4 Conclusion and Future Work

Live coding encourages the interleaved coding strategy, which in turn reduces task completion time for some tasks.

In this chapter, we first presented a study to investigate how developers change their coding behavior when they work in a live coding environment. Our evaluation is among the first to use a live coding system with rich output of runtime information for a realistic coding task. We were able to demonstrate that working in this environment causes developers to adapt the interleaved coding strategy, in which they frequently switch between editing and correcting their application. This strategy caused developers to fix bugs faster after their introduction, and it led to a decrease in task completion time for some tasks. More studies are needed to fully understand in which cases the interleaved coding strategy is beneficial, whether or not it may even be harmful for other tasks, and how to design an interface that only encourages the strategy if it is suitable. Also, studies on the long term impact of working in a live coding environment are missing. Since live coding environments present a huge amount of information to developers that they are not used to, they may need some training time to learn how to best utilize the presented information.

A challenge of live coding tools is the presentation of huge amounts of runtime information.

This problem can also be addressed by the interaction design: To allow developers to cope with the large amount of runtime information, live coding environments need new types of interaction to quickly select which information to view in more or less detail. We propose a probing metaphor to set up more detailed visualizations for runtime information quickly, though an evaluation of the various possible designs is left for future work. Data visualization is also an important field of research for the future improvement of Fiddlets.

Live coding also poses technical challenges. We proposed an interaction technique that circumvents the continuous re-execution of the complete application and only applies live coding to small code fragments. Our first study results are promising: Developers quickly adopt Fiddlets as a code inspection tool, and we found no downsides of constraining live coding to a small code fragment. However, our study was flawed by an uneven distribution of programming experience between the conditions. Therefore, we want to replicate the study we have performed with more participants to obtain more similarly sampled groups in terms of JavaScript experience. Second, a more advanced prototype of Fiddlets using a complete implementation of runtime analysis is needed in order to test if the interaction concepts work in a real development task.

Fiddlets show how technical problems can be circumvented through interaction design.

Chapter 7

Summary and Future Work

"The trouble with programmers is that you can never tell what a programmer is doing until it's too late."

—Seymour Cray

This thesis aims to understand how interaction design affects developers cognitive processes.

Research on the human aspects of software development is a wide field that encompasses many facets: It encompasses questions from cognitive psychology, which is essential for understanding basic cognitive processes of software developers, questions of how to visualize information and how to design effective interactions, and technical issues of how to efficiently obtain useful information to support developers. All of these facets contribute to the goal of making software development easier, faster, and more enjoyable for developers. None of these facets can stand on its own: Technical capabilities of a system can limit or enable potential interaction designs. Knowledge of the cognitive processes of software developers should impact on the design of new software development tools on the one hand, but on the other hand, a newly designed tool can change the cognitive processes of software developers. In this thesis, we have tried to understand these interdependencies in more detail.

7.1 Summary and Contributions

For APIs, choosing the level of abstraction is one of the most important design decisions.

In the first research project we presented, we compared a procedural and a declarative framework for the programmatic authoring of animations in order to explore how their effects on the developers' behavior differ. The procedural framework allows developers to specify the properties of animated elements for each frame. Using the declarative framework, developers only specify keyframes, i.e., the properties of elements at certain points in time, and the framework interpolates between these keyframes automatically. Hence, the declarative framework provides a higher level of abstraction, i.e., it encapsulates the interpolation of properties. We found that the abstraction level is a crucial aspect in the design of frameworks. For many tasks the higher level of abstraction aligned well with the developers' programming plans. This resulted in less need for manual tests, a higher confidence of participants in their own solutions, and lower task completion time. However, providing a higher level of abstraction requires a careful design. In our study, the declarative framework implicitly created a timeline based on the keyframes created. Because the concept of a timeline was not made visible through the naming of functions, this behavior led to misconceptions of developers that can more than mitigate the previously described benefits. In this case, developers are actually faster when using the procedural framework, that is, a lower level of abstraction.

Integrating documentation and unit test authoring into the edit-test-edit cycle can encourage developers to write more of these documents.

Documentation and unit tests can help to comprehend software, but developers often do not create these documents sufficiently enough. We explored whether or not an improved interaction design can encourage developers to create these documents more. Our design is based on the idea to inform the authoring tool with runtime information captured during manual tests of the application performed by the developer. We propose to write documentation and unit tests as part of the edit-test-edit cycle developers already perform, in other words, to write documentation and unit tests for the part of the code a developer just tested manually. This ensures that runtime information is available for the part of the code that is about to be documented, and moreover, it ensures that developers have a complete and recent mental model of the code. We implemented this idea in a prototype called VESTA. In evaluating VESTA, we found that in order to encourage developers successfully it is crucial to provide near-term value for their current activity as part of the authoring process. We successfully implemented this in VESTA's documentation component: Because it provided type information for parameters in JavaScript, it was useful for developers to verify the correctness of code and to spot misconceptions of the return values of framework functions easily.

We proposed two design guidelines for call graph navigation tools: *proactive information visualization* and *comprehensible relevance*.

In our next research project, we compared different call graph navigation tools. Call graph navigation tools allow to navigate along method calls and aim to support developers in comprehending the procedural structure of object-oriented source code. We propose two design guidelines for the design of such tools: proactive information visualization and comprehensible relevance. We created Stacksplorer based on these design guidelines. It shows the call graph neighborhood of the focus method, i.e., the method currently being edited in the source code editor. Stacksplorer implements proactive information visualization and comprehensible relevance, because firstly it automatically updates when the developer opens a different method in the source code editor, and secondly, all visible navigation targets are directly connected to the focus method in the call graph. The downside of this design is that it does not allow to navigate to methods further away in the call graph efficiently. Blaze was designed to solve this problem, while still implementing our design guidelines. It shows one path through the call graph including the focus method. Because plenty of such paths might exist, Blaze allows developers to adjust the visualization to show the path they are most interested in.

Tools implementing these guidlines cause developers to change their navigation behavior.

We show that, compared to developers using an IDE without any call graph navigation tool, Stacksplorer and Blaze can substantially alter the developers' navigation behavior and encourage them to adopt a more effective navigation strategy. This ultimately results in lower task completion times for certain maintenance tasks. However, developers using the Call Hierarchy, a widely used call graph navigation tool in modern IDEs that does not implement either of our guidelines, do not alter their navigation behavior. In our study, they were still more successful than developers without any tool but not faster. Navigation to distant targets in the call graph, which was possible using Blaze or the Call Hierarchy, was rarely used. Hence, the additional interaction required is unlikely to be worth it.

We developed a new analysis technique to detect and describe changes in navigation behavior.

To be able to quantitatively describe and identify the differences in navigation behavior, we developed a novel analysis technique. Our method uses a set of predictive models that each represents a typical micro-navigation pattern, e.g., navigation to another method of the same object, or navigation to a recently visited method. For all models, we calculate the accuracy with which they predict a recorded navigation behavior. A vector combining all prediction accuracies represents the navigation behavior. This representation allows for quantitative comparisons and enabled the analysis described above. We show how our technique can be used in concert with other metrics to precisely describe how the navigation behavior changed between conditions.

Live coding tools encourage developers to adopt a different coding strategy that can decrease task completion time for some tasks.

In all projects summarized so far, we assumed that a developer comprehends source code by reading it. Frameworks can support this process by providing good naming and abstractions, while call graph navigation supports this process by allowing easier access to related parts of the source code. To complement the reading-based comprehension of source code, developers regularly execute the application using some exemplary input to observe its runtime behavior, either by only looking at the program output or by using a debugger. Live coding environments support this strategy: After every change, they automatically execute the application, analyze the execution, and provide rich runtime information. We implemented METIS, a working, prototypical live coding environment, and found that developers using METIS fix bugs they introduced in newly written code faster. This is caused by a change in their coding strategy: Developers using live coding are more likely to adopt the interleaved coding strategy, i.e., they quickly alternate between writing new code and correcting existing code. Adopting this strategy can decrease the task completion time significantly for some tasks, even when developers do not use a live coding environment.

Fiddlets allows to use live coding for small code fragments, while still reflecting how the code would behave when executed as part of the complete application.

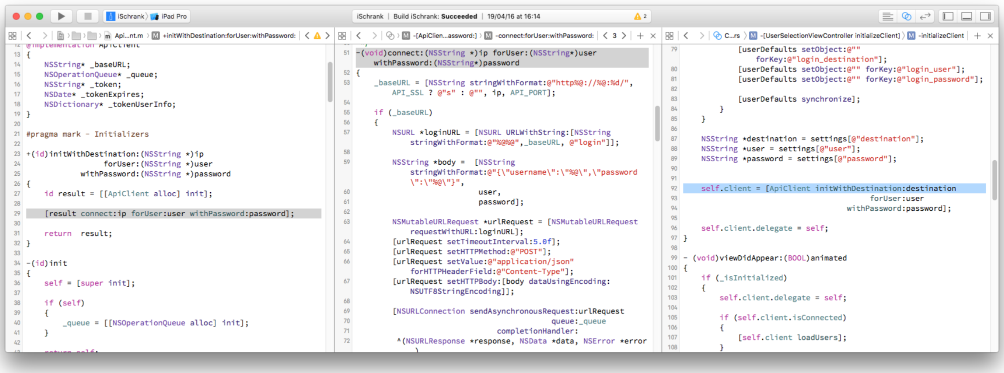
Live coding environments are also an example of development tools that need to be designed around technical limitations. Re-executing the complete application as described above leads to various technical problems. For example, the runtime analysis implemented in METIS introduces a substantial performance penalty, hence, it cannot be used to develop larger applications live. While these technical challenges in the implementation of live coding environments can likely be overcome in the future, we present Fiddlets, an interaction design to circumvent these problems. Fiddlets provide live coding environments for small fragments of source code as in-line editors, which are embedded into the source code editor. A developer can invoke a Fiddlet for a single line of source code. The Fiddlet automatically generates a context for the line of code that simulates the context in which the line would be executed during a complete execution of the application. A preliminary evaluation of Fiddlets shows that the benefits for developers are comparable to those of METIS, a full live coding environment. In our study, novice developers using Fiddlets could solve five programming tasks as fast as expert developers not using it.

7.2 Future Work and Closing Remarks

Each research project presented in this thesis offers manifold opportunities for future work that have been described as part of the individual chapters before. We will not repeat the previously reported ideas here and instead focus on how our research aligns with larger scale trends both in current development tools as well as in future development and research.

We found improved call graph navigation tools in more recent IDEs.

We are delighted to see that many of the interaction design concepts we have explored and proposed slowly make their way into widely used development tools. For example, newer versions of Apple's Xcode IDE implemented an assistant feature that implements proactive information vi-


iSchrank
iPad Pro
iSchrank | Build iSchrank: Succeeded | 19/04/16 at 16:14
2
Api...nt.m
+initWithDestination:forUser:withPassword:
-[ApiClien...assword:]
-connect:forUser:withPassword:
3
C...rs
-[UserSelectionViewController initializeClient]
-initializeClient

```
{
    NSString* _baseURL;
    NSOperationQueue* _queue;
    NSString* _token;
    NSDate* _tokenExpires;
    NSDictionary* _tokenUserInfo;
}

#pragma mark - Initializers

+(id)initWithDestination:(NSString *)ip
                 forUser:(NSString *)user
             withPassword:(NSString *)password
{
    id result = [[ApiClient alloc] init];

    [result connect:ip forUser:user withPassword:password];

    return  result;
}

-(id)init
{
    self = [super init];

    if (self)
    {
        _queue = [[NSOperationQueue alloc] init];
    }
```

```
-(void)connect:(NSString *)ip forUser:(NSString*)user
   withPassword:(NSString*)password
{
    _baseURL = [NSString stringWithFormat:@"http%@://%@:%d/",
       API_SSL ? @"s" : @"", ip, API_PORT];

    if (_baseURL)
    {
        NSURL *loginURL = [NSURL URLWithString:[NSString
           stringWithFormat:@"%@%@",_baseURL, @"login"]];

        NSString *body =  [NSString
           stringWithFormat:@"{\"username\":\"%@\",\"password
           \":\"%@\"}",
                               user,
                               password];

        NSMutableURLRequest *urlRequest = [NSMutableURLRequest
           requestWithURL:loginURL];
        [urlRequest setTimeoutInterval:5.0f];
        [urlRequest setHTTPMethod:@"POST"];
        [urlRequest setValue:@"application/json"
           forHTTPHeaderField:@"Content-Type"];
        [urlRequest setHTTPBody:[body dataUsingEncoding:
           NSUTF8StringEncoding]];

        [NSURLConnection sendAsynchronousRequest:urlRequest
                                           queue:_queue
                               completionHandler:
         ^(NSURLResponse *response, NSData *data, NSError *error
```

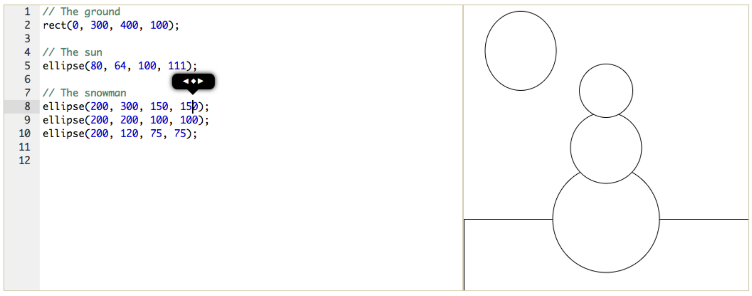
```
            [userDefaults setObject:@""
                forKey:@"login_destination"];
            [userDefaults setObject:@"" forKey:@"login_user"];
            [userDefaults setObject:@"" forKey:@"login_password"];

            [userDefaults synchronize];
        }
    }

    NSString *destination = settings[@"destination"];
    NSString *user = settings[@"user"];
    NSString *password = settings[@"password"];


    self.client = [ApiClient initWithDestination:destination
                                            forUser:user
                                       withPassword:password];

    self.client.delegate = self;
}

- (void)viewDidAppear:(BOOL)animated
{
    if (_isInitialized)
    {
        self.client.delegate = self;

        if (self.client.isConnected)
        {
            [self.client loadUsers];
        }
```

Figure 7.1: Newer versions of Xcode allow to show Assistant columns (middle and right) next to the primary source code editor (left). They can be configured to show source code related to the currently edited method, e.g., one of its callers or callees.

sualization (see Figure 7.1). The feature adds new editors to the view, that show source code related to the currently edited method in the primary editor and, like Stacksplorer and Blaze, update when the user navigates in the primary editor. For example, assistant editors can show one caller or callee of the currently edited method. However, Apple's implementation only shows a single method. Hence, it is unlikely to provide sufficient information scent to substantially increase the discoverability of important related code. We propose to use our analysis methods to quantify the effects of such interface design choices, and use the results to iteratively improve the design of newly developed tools.

Anonymous functions are increasingly used and add complexity to the procedural hierarchy of software.

At the same time, the requirements for navigation tools change with recent advancements in programming languages and common patterns. For example, many modern languages implement lambda functions, i.e., anonymous functions that can be passed as arguments to other functions. Lambda functions are not accessible using call graph navigation, because they are not instance methods of a class. Still, they contribute to the procedural structure of the application, and, hence, developers have to face the problem of understanding their purpose in the applications. These new features in programming languages pose new challenges and opportunities for the design of future navigation tools. Further, little is known about the specific



```
// The ground
rect(0, 300, 400, 100);

// The sun
ellipse(80, 64, 100, 111);

// The snowman
ellipse(200, 300, 150, 150);
ellipse(200, 200, 100, 100);
ellipse(200, 120, 75, 75);
```

Figure 7.2: The Kahn Academy uses a live coding editor in their programming courses. The output on the right side is re-rendered after every change to the source code. Numeric values can be adjusted interactively by dragging with the mouse instead of typing.

cognitive models developers use to mentally comprehend these constructs.

Live coding is increasingly used, especially as experimentation environment.

Another example for a development tool that is increasingly adopted in real-world development tools is live coding. Light Table[1] is a newly developed live coding environment for web development. Xcode implements live coding in a feature called Playgrounds that allows developers to use live coding for a short source code snippet. It is mainly built for ad-hoc experimentations. Our research on live coding and the Playgrounds feature were released around the same time and our prototype METIS is very similar to Xcode Playgrounds. While Xcode Playgrounds allow to import classes used in the application a developer is working on, these need to be instantiated and contextualized with realistic parameters by the developer. With Fiddlets we proposed an interaction design that can simplify this task and could substantially increase the usefulness of Playgrounds.

Probably, the most active field adopting live coding is programming education: Live coding editors are used in the courses by the Kahn Academy[2] (see Figure 7.2), the Start

[1]http://lighttable.com/

[2]https://www.khanacademy.org/computing/computer-programming

Analyzing the effect of using live coding to learn programming is an interesting future research direction.

Coding Initiative[3], and many others. Research has already started to investigate the didactic effect of live coding, and in the long term, we believe that this trend is interesting to investigate in. Future research could explore which comprehension strategies programmers develop when they learn programming right from the start using a live coding environment. Reasoning is often driven by prior experience, and the investigation of learning tools and comprehension strategies of learners could allow to control this effect to some degree.

Extensible development environments can support the research about development tools.

One of the hardest problems researchers have to face when studying the comprehension process of developers, is to control for inter-subject differences, e.g., due to different prior experience. Deploying a tool into a real development environment would allow the study of how a large number of developers use the tool over an extended period of time and how they learn to integrate the tool into their workflow. However, these studies are often impractical, because the development effort to create a robust prototype is very high, developers are often cautious about changes to their working environment, and capturing data is difficult because the source code of real world projects is often a business secret. In the future, we propose that development tools could implement better support for researchers to deploy prototype features. A plug-in architecture tailored to the researcher's needs should offer support for radical interface changes (e.g., for drawing the overlays used in Stacksplorer and Blaze, or the rich formatting embedded into the source code editor we used in VESTA), it should also provide an anonymized and centralized framework for the collection of usage data, and it should implement sandboxing for plug-ins to ensure stability and security for cautious developers. Many recent development tools, such as Brackets[4] or Atom[5], provide a first step in this direction.

Future research should explore how cognitive processes of other knowledge workers are influenced by the tools they use.

The research presented in this thesis focused entirely on software developers. We believe that software developers can be a window into the larger space of *knowledge workers*, who face various cognitively demanding comprehen-

[3]http://start-coding.de/
[4]http://brackets.io/
[5]https://atom.io/

sion tasks. In the future, it would be interesting to investigate how our results translate to other domains. For example, one interesting type of knowledge workers to investigate are literary scholars. Similar to developers, they need to comprehend information that is spread across different resources. Future research might help to understand how their use of resources and the conclusions they draw changes depending on the tools they use. Another interesting domain might be stock trading: Do the decisions of stock traders change depending on the tools they use to browse the information at their disposal? Similar questions can even arise for non-professionals: For example, it would be interesting to explore how the perception of movies changes when we offer novel non-linear navigation tools based on the content of the movie [Pavel et al., 2015].

In all of these examples, a knowledge worker needs to comprehend some kind of media, such as source code, prose text, numerical information, or a movie. A tool in these examples would support the comprehension of some structural aspect of the media, e.g., links between different textual resources, the relevance of specific performance indicators, or the relationships between actors in a movie. We could compare tools across different types of media by first identifying the underlying structure of the specific kind of media [Karrer, 2013] and then analyzing how tools facilitate browsing it. Using this approach, we hope to generalize our findings about the interdependence of tools and cognitive processes for various types of media.

We believe that understanding how tools not only support cognitively demanding tasks but change how humans approach these tasks is a fascinating field of research in HCI for years to come. The research presented in this thesis aimed to contribute some insights into this topic.

Bibliography

Adelson, Beth (1981). “Problem solving and the development of abstract categories in programming languages”. *Memory & Cognition* 9.4, pp. 422–433. DOI: 10.3758/BF03197568.

Artzi, Shay, Dolby, Julian, Jensen, Simon Holm, Møller, Anders, Tip, Frank (2011). “A Framework for Automated Testing of Javascript Web Applications”. *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, pp. 571–580. DOI: 10.1145/1985793.1985871.

Baeza-Yates, Ricardo A., Ribeiro-Neto, Berthier (1999). *Modern Information Retrieval*. Addison-Wesley Professional. ISBN: 978-0321416919.

Bangor, Aaron, Kortum, Philip T, Miller, James T (2008). “An Empirical Evaluation of the System Usability Scale”. *International Journal of Human-Computer Interaction* 24.6, pp. 574–594. DOI: 10.1080/10447310802205776.

Belzmann, Ewgenij (2013). “Utilization and Visualization of Program State as Input Data in a Live Coding Environment”. Diploma Thesis. RWTH Aachen University.

Bhat, Thirumalesh, Nagappan, Nachiappan (2006). “Evaluating the Efficacy of Test-driven Development: Industrial Case Studies”. *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*. New York, NY, USA: ACM, pp. 356–363. DOI: 10.1145/1159733.1159787.

Bragdon, Andrew, Zeleznik, Robert, Reiss, Steven P, Karumuri, Suman, Cheung, William, Kaplan, Joshua, Coleman, Christopher, Adeputra, Ferdi, LaViola, Joseph J, Jr (2010). “Code bubbles: a working set-based interface for code understanding and maintenance”. *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 2503–2512. DOI: 10.1145/1753326.1753706.

Brandt, Joel, Guo, Philip J, Lewenstein, Joel, Dontcheva, Mira, Klemmer, Scott R (2009a). “Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover”. *IEEE Software* 26.5, pp. 18–24. DOI: 10.1109/MS.2009.147.

Brandt, Joel, Guo, Philip J, Lewenstein, Joel, Dontcheva, Mira, Klemmer, Scott R (2009b). “Two studies of opportunistic programming: interleaving web foraging, learning, and writing code”. *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*. New York, NY, USA: ACM, pp. 1589–1598. DOI: 10.1145/1518701.1518944.

Brinkmann, Ron (2008). *The Art and Science of Digital Compositing*. 2nd ed. The Morgan Kaufmann Series in Computer Graphics. Amsterdam: Morgan Kaufmann Publishers/Elsevier. ISBN: 978-0123706386.

Brooks Jr., Frederick P. (1987). “No Silver Bullet: Essence and Accidents of Software Engineering”. *Computer* 20.4, pp. 10–19. ISSN: 0018-9162. DOI: 10.1109/MC.1987.1663532.

Brooks, Ruven (1980). “Studying programmer behavior experimentally: the problems of proper methodology”. *Communications of the ACM* 23.4, pp. 207–213. DOI: 10.1145/358841.358847.

Brooks, Ruven (1983). “Towards a theory of the comprehension of computer programs”. *International Journal of Man-Machine Studies* 18.6, pp. 543–554. DOI: 10.1016/S0020-7373(83)80031-5.

Burg, Brian, Bailey, Richard, Ko, Andrew J., Ernst, Michael D. (2013). “Interactive Record/Replay for Web Application Debugging”. *UIST '13: Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, pp. 473–484. DOI: 10.1145/2501988.2502050.

Burkhardt, Jean-Marie, Détienne, Françoise, Wiedenbeck, Susan (1997). “Mental Representations Constructed by Experts and Novices in Object-Oriented Program Comprehension”. *INTERACT '97: Human-Computer Interaction*. Boston, MA: Springer US, pp. 339–346. DOI: 10.1007/978-0-387-35175-9_55.

Burnett, Margaret M, Atwood Jr, John W, Welch, Zachary T (1998). “Implementing level 4 liveness in declarative visual programming languages”. *VL '98: Proceedings of the IEEE Symposium on Visual Languages*. Washington, DC, USA: IEEE, pp. 126–133. DOI: 10.1109/VL.1998.706155.

Buse, Raymond P.L., Weimer, Westley R. (2008). “Automatic Documentation Inference for Exceptions”. *ISSTA '08: Proceedings of the 2008 International Symposium* *on Software Testing and Analysis*. New York, NY, USA: ACM, pp. 273–282. DOI: 10.1145/1390630.1390664.

Buse, Raymond P.L., Weimer, Westley R. (2010). "Automatically Documenting Program Changes". *ASE '10: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, pp. 33–42. DOI: 10.1145/1858996.1859005.

Chang, Bay-Wei, Ungar, David (1993). "Animation: From Cartoons to the User Interface". *UIST '93: Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, pp. 45–55. DOI: 10.1145/168642.168647.

Choi, William, Brandt, Joel, Klemmer, Scott R (2008). "Rehearse: Coding Interactively while Prototyping". *UIST '08: Adjunct Proceedings of the 21th Annual ACM Symposium on User Interface Software & Technology*. New York, NY, USA: ACM.

Clarke, Steven (2004). "Measuring API Usability". *Dr. Dobb's Journal* 29, pp. 6–9.

Corritore, Cynthia L, Wiedenbeck, Susan (2001). "An exploratory study of program comprehension strategies of procedural and object-oriented programmers". *International Journal of Human-Computer Studies* 54.1, pp. 1–23. DOI: 10.1006/ijhc.2000.0423.

Crockford, Douglas (2008). *JavaScript: The Good Parts*. O'Reilly Media, Inc.

Čubranić, Davor, Murphy, Gail C (2003). "Hipikat: recommending pertinent software development artifacts". *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE, pp. 408–418. DOI: 10.1109/ICSE.2003.1201219.

Cypher, Allen, ed. (1993). *Watch What I Do: Programming by Demonstration*. 1st ed. Cambridge, Massachusetts: The MIT Press. ISBN: 978-0262032131.

Dahotre, Aniket, Zhang, Yan, Scaffidi, Christopher (2010). "A Qualitative Study of Animation Programming in the Wild". *ESEM '10: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: ACM. DOI: 10.1145/1852786.1852825.

De Alwis, Brian, Murphy, Gail C (2006). "Using Visual Momentum to Explain Disorientation in the Eclipse IDE". *VL/HCC '06: IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE, pp. 51–54. DOI: 10.1109/VLHCC.2006.49.

De Alwis, Brian, Murphy, Gail C, Robillard, Martin P (2007). "A Comparative Study of Three Program Exploration Tools". *ICPC '07: 15th IEEE International Conference on Program Comprehension*, pp. 103–112. DOI: 10.1109/ICPC.2007.6.

DeLine, R, Czerwinski, Mary, Robertson, G (2005). "Easing program comprehension by sharing navigation data". *VL/HCC '05: IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 241–248. DOI: 10.1109/VLHCC.2005.32.

DeLine, Robert, Rowan, Kael (2010). "Code canvas: zooming towards better development environments". *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, pp. 207–210. DOI: 10.1145/1810295.1810331.

Détienne, Françoise (2002). *Software Design—cognitive Aspects*. Ed. by Frank Bott. New York, NY, USA: Springer-Verlag New York, Inc. ISBN: 1-85233-253-0.

Dijk, Teun A van, Kintsch, Walter (1983). *Strategies of Discourse Comprehension*. New York: Academic Press. ISBN: 978-0127120508.

Duisberg, Robert Adámy (2009). "Animation Using Temporal Constraints: An Overview of the Animus System". *Human–Computer Interaction* 3.3, pp. 275–307. DOI: 10.1207/s15327051hci0303_3.

Eaddy, Marc, Zimmermann, Thomas, Sherwood, Kaitlin D., Garg, Vibhav, Murphy, Gail C., Nagappan, Nachiappan, Aho, Alfred V. (2008). "Do Crosscutting Concerns Cause Defects?" *IEEE Transactions on Software Engineering* 34.4, pp. 497–515. DOI: 10.1109/TSE.2008.36.

Edwards, Jonathan (2004). "Example centric programming". *ACM SIGPLAN Notices* 39.12, pp. 84–91. DOI: 10.1145/1052883.1052894.

Ernst, Michael D, Perkins, Jeff H, Guo, Philip J, McCamant, Stephen, Pacheco, Carlos, Tschantz, Matthew S, Xiao, Chen (2007). "The Daikon system for dynamic detection of likely invariants". *Science of Computer Programming* 69.1-3, pp. 35–45. DOI: 10.1016/j.scico.2007.01.015.

Fairbanks, George, Garlan, David, Scherlis, William (2006). "Design Fragments Make Using Frameworks Easier". *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, pp. 75–88. DOI: 10.1145/1167473.1167480.

Feathers, Michael (2004). *Working effectively with legacy code*. 1st ed. Prentice Hall. ISBN: 978-0131177055.

Fisher, D. A. (1978). "DoD's Common Programming Language Effort". *Computer* 11.3, pp. 24–33. DOI: `10.1109/C-M.1978.218092`.

Fouse, Adam, Weibel, Nadir, Hutchins, Edwin, Hollan, James D. (2011). "ChronoViz: A System for Supporting Navigation of Time-coded Data". *CHI EA '11: CHI '11 Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 299–304. DOI: `10.1145/1979742.1979706`.

Francel, Margaret Ann, Rugaber, Spencer (2001). "The value of slicing while debugging". *Science of Computer Programming - Special issue on program comprehension (IWPC '99)* 40.2-3, pp. 151–169. DOI: `10.1016/S0167-6423(01)00013-2`.

Fröhlich, Peter, Link, Johannes (2000). "Automated Test Case Generation from Dynamic Models". *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*. London, UK, UK: Springer-Verlag, pp. 472–492. DOI: `10.1007/3-540-45102-1_23`.

Fry, Christopher (1997). "Programming on an already full brain". *Communications of the ACM* 40.4, pp. 55–64. DOI: `10.1145/248448.248459`.

Furnas, G W (1986). "Generalized Fisheye Views". *CHI '86: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 16–23. DOI: `10.1145/22627.22342`.

Gilmore, D J, Green, T R G (1984). "The comprehensibility of programming notations". *INTERACT '84: Human-Computer-Interaction*, pp. 461–464.

González, Victor M, Mark, Gloria (2004). "Constant, constant, multi-tasking craziness: managing multiple working spheres". *CHI '04: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 113–120. DOI: `10.1145/985692.985707`.

Grad, Burton (2007). "The Creation and the Demise of VisiCalc". *IEEE Annals of the History of Computing* 29.3, pp. 20–31. DOI: `10.1109/MAHC.2007.49`.

Green, T R G, Petre, M (1996). "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". *Journal of Visual Languages and Computing* 7.2, pp. 131–174. DOI: `10.1006/jvlc.1996.0009`.

Hartmann, Björn, Dhillon, Mark, Chan, Matthew K (2011). "HyperSource: bridging the gap between source and code-related web sites". *CHI '11: Proceedings of the 2011 annual conference on Human factors in computing systems*. New York, NY, USA: ACM, pp. 2207–2210. DOI: `10.1145/1978942.1979263`.

Heinen, Björn (2012). "A Live Coding Editor". Bachelor's Thesis. Aachen: RWTH Aachen University.

Henderson, Peter, Weiser, Mark (1985). "Continuous Execution: The VisiProg Environment". *ICSE '85: Proceedings of the 8th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, pp. 68–74.

Hennings, Michael (2016). "Programming Animations - Current Practice and the Effect of Integrated Live Evaluation". Master's Thesis. Aachen: RWTH Aachen University.

Hennings, Michael, Krämer, Jan-Peter, Brandt, Joel, Borchers, Jan (2016a). *Programmatic Animations - Cheat Sheets*. DOI: `10.5281/zenodo.45263`.

Hennings, Michael, Krämer, Jan-Peter, Brandt, Joel, Borchers, Jan (2016b). *Programmatic Animations - User Study Tasks*. DOI: `10.5281/zenodo.45262`.

Herman, Ivan, Melancon, Guy, Marshall, M Scott (2000). "Graph visualization and navigation in information visualization: A survey". *IEEE Transactions on Visualization and Computer Graphics* 6.1, pp. 24–43. DOI: `10.1109/2945.841119`.

Hill, William C, Hollan, James D, Wroblewski, Dave, McCandless, Tim (1992). "Edit wear and read wear". *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM Press, pp. 3–9. DOI: `10.1145/142750.142751`.

Horvitz, Eric (1999). "Principles of Mixed-initiative User Interfaces". *CHI '99: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 159–166. DOI: `10.1145/302979.303030`.

Huang, Mao Lin, Eades, Peter, Wang, Junhu (1998). "On-line Animated Visualization of Huge Graphs using a Modified Spring Algorithm". *Journal of Visual Languages and Computing* 9.6, pp. 623–645. DOI: `10.1006/jvlc.1998.0094`.

Hundhausen, Christopher D, Brown, Jonathan L (2007a). "What You See Is What You Code: A "live" algorithm development and visualization environment for novice learners". *Journal of Visual Languages and Computing* 18.1, pp. 22–47. DOI: `10.1016/j.jvlc.2006.03.002`.

Hundhausen, Christopher D, Brown, Jonathan Lee (2007b). "An experimental study of the impact of visual semantic feedback on novice programming". *Journal of Visual Languages and Computing* 18.6, pp. 537–559. DOI: `10.1016/j.jvlc.2006.09.001`.

Hundhausen, Christopher D, Douglas, Sarah A, Stasko, John T (2002). "A Meta-Study of Algorithm Visualization Effectiveness". *Journal of Visual Languages and Computing* 13.3, pp. 259–290. DOI: `10.1006/jvlc.2002.0237`.

Hundhausen, Christopher D, Farley, Sean F, Brown, Jonathan L (2009). "Can direct manipulation lower the barriers to computer programming and promote transfer of training?" *ACM Transactions on Computer-Human Interaction* 16.3, 13:1–13:40. DOI: `10.1145/1592440.1592442`.

Janzen, Doug, De Volder, Kris (2003). "Navigating and querying code without getting lost". *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, pp. 178–187. DOI: `10.1145/643603.643622`.

Johnson, Ralph E (1997). "Frameworks = (components + patterns)". *Communications of the ACM* 40.10, pp. 39–42. DOI: `10.1145/262793.262799`.

Kajko-Mattsson, Mira (2001). "The state of documentation practice within corrective maintenance". *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance*, pp. 354–363. DOI: `10.1109/ICSM.2001.972748`.

Karrer, Thorsten (2013). "Semantic Navigation in Digital Media". PhD Thesis. Aachen: RWTH Aachen University.

Karrer, Thorsten, Krämer, Jan-Peter, Diehl, Jonathan, Hartmann, Björn, Borchers, Jan (2011). "Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency". *UIST '11: Proceedings of the 24th annual ACM symposium on User Interface Software and Technology*. New York, NY, USA: ACM, pp. 217–224. DOI: `10.1145/2047196.2047225`.

Kersten, Mik, Murphy, Gail C (2005). "Mylar: a degree-of-interest model for IDEs". *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, pp. 159–168. DOI: `10.1145/1052898.1052912`.

Kiel, Henning (2009). "Reducing mental context switches during programming". Master's Thesis. RWTH Aachen University.

Ko, Andrew J., Aung, Htet Htet, Myers, Brad A. (2005). "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks". *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, New York, USA: ACM, pp. 126–135. DOI: 10.1145/1062455.1062492.

Ko, Andrew J., Myers, Brad A. (2008). "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior". *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, pp. 301–310. DOI: 10.1145/1368088.1368130.

Ko, Andrew J., Myers, Brad A. (2009). "Finding causes of program output with the Java Whyline". *CHI '09: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 1569–1578. DOI: 10.1145/1518701.1518942.

Ko, Andrew J., Myers, Brad A., Coblenz, Michael J., Aung, Htet Htet (2006). "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks". *IEEE Transactions on Software Engineering* 32.12, pp. 971–987. DOI: 10.1109/TSE.2006.116.

Kotula, Jeffrey (2000). "Source Code Documentation: An Engineering Deliverable". *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems*. Washington, DC, USA: IEEE Computer Society, p. 505. DOI: 10.1109/TOOLS.2000.10052.

Krämer, Jan-Peter (2011). "Stacksplorer Understanding Dynamic Program Behavior". Diploma Thesis. RWTH Aachen University.

Krämer, Jan-Peter, Brandt, Joel, Borchers, Jan (2016a). "Using Runtime Traces to Improve Documentation and Unit Test Authoring for Dynamic Languages". *CHI '16: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 3232–3237. DOI: 10.1145/2858036.2858311.

Krämer, Jan-Peter, Hennings, Michael, Brandt, Joel, Borchers, Jan (2016b). "An Empirical Study of Programming Paradigms for Animation". *CHASE '16: Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. New York, NY, USA: ACM, pp. 58–61. DOI: 10.1145/2897586.2897597.

Krämer, Jan-Peter, Hennings, Michael, Brandt, Joel, Borchers, Jan (2016c). *Programmatic Animations - Performance Data*. DOI: 10.5281/zenodo.45273.

Krämer, Jan-Peter, Karrer, Thorsten, Diehl, Jonathan, Borchers, Jan (2010). "Stacksplorer: Understanding Dynamic Program Behavior". *UIST '10: Adjunct Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, pp. 433–434. DOI: `10.1145/1866218.1866257`.

Krämer, Jan-Peter, Karrer, Thorsten, Kurz, Joachim, Wittenhagen, Moritz, Borchers, Jan (2013). "How Tools in IDEs Shape Developers' Navigation Behavior". *CHI '13: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 3073–3082. DOI: `10.1145/2470654.2466419`.

Krämer, Jan-Peter, Kurz, Joachim, Karrer, Thorsten, Borchers, Jan (2012a). "Blaze". *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, pp. 1457–1458. DOI: `10.1109/ICSE.2012.6227066`.

Krämer, Jan-Peter, Kurz, Joachim, Karrer, Thorsten, Borchers, Jan (2012b). "Blaze: Supporting Two-phased Call Graph Navigation in Source Code". *CHI EA '12: CHI '12 Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 2195–2200. DOI: `10.1145/2212776.2223775`.

Krämer, Jan-Peter, Kurz, Joachim, Karrer, Thorsten, Borchers, Jan (2014). "How live coding affects developers' coding behavior". *VL/HCC '14: IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE, pp. 5–8. DOI: `10.1109/VLHCC.2014.6883013`.

Kurz, Joachim (2011). "Blaze - Navigating Source Code via Call Stack Contexts". Bachelor's Thesis. RWTH Aachen University.

Kurz, Joachim (2013). "Evaluating Developer Strategies in a Live Coding Environment". Master's Thesis. Aachen: RWTH Aachen University.

LaToza, Thomas D., Garlan, David, Herbsleb, James D, Myers, Brad A. (2007). "Program comprehension as fact finding". *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, pp. 361–370. DOI: `10.1145/1287624.1287675`.

LaToza, Thomas D., Myers, Brad A. (2010a). "Developers ask reachability questions". *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, pp. 185–194. DOI: `10.1145/1806799.1806829`.

LaToza, Thomas D., Myers, Brad A. (2010b). "Hard-to-answer questions about code". *PLATEAU '10: Evaluation and Usability of Programming Languages and Tools*. New York, NY, USA: ACM, 8:1–8:6. DOI: 10.1145/1937117.1937125.

LaToza, Thomas D., Myers, Brad A. (2010c). "Searching Across Paths". *SUITE '10: Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*. New York, NY, USA: ACM, pp. 29–32. DOI: 10.1145/1809175.1809183.

LaToza, Thomas D., Venolia, Gina, DeLine, Robert (2006). "Maintaining mental models: a study of developer work habits". *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, pp. 492–501. DOI: 10.1145/1134285.1134355.

Lawrance, Joseph, Bellamy, Rachel, Burnett, Margaret, Rector, Kyle (2008). "Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks". *CHI '08: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 1323–1332. DOI: 10.1145/1357054.1357261.

Lawrance, Joseph, Burnett, Margaret, Bellamy, Rachel, Bogart, Christopher, Swart, Calvin (2010). "Reactive information foraging for evolving goals". *CHI '10: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 25–34. DOI: 10.1145/1753326.1753332.

Letovsky, Stanley (1987). "Cognitive processes in program comprehension". *Journal of Systems and Software* 7.4, pp. 325–339. DOI: 10.1016/0164-1212(87)90032-X.

Lewandowski, Dennis (2015). "Fiddlets: Contextualised Inline Code-Execution Environments". Master's Thesis. RWTH Aachen University.

Lewis, Harry R, Papadimitriou, Christos H (1997). *Elements of the Theory of Computation*. 2nd. Upper Saddle River, NJ, USA: Prentice Hall. ISBN: 978-0132624787.

Liblit, Ben, Begel, Andrew, Sweetser, Eve (2006). "Cognitive Perspectives on the Role of Naming in Computer Programs". *Proceedings of the 18th Annual Psychology of Programming Interest Group Workshop*.

Lieber, Tom, Brandt, Joel R., Miller, Rob C. (2014). "Addressing Misconceptions About Code with Always-on Programming Visualizations". *CHI '14: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 2481–2490. DOI: 10.1145/2556288.2557409.

Littman, David C, Pinto, Jeannine, Letovsky, Stanley, Soloway, Elliot (1987). "Mental models and software maintenance". *Journal of Systems and Software* 7.4, pp. 341–355. DOI: `10.1016/0164-1212(87)90033-1`.

Maalej, Walid, Tiarks, Rebecca, Roehm, Tobias, Koschke, Rainer (2014). "On the Comprehension of Program Comprehension". *ACM Transactions on Software Engineering and Methodology* 23.4, 31:1–31:37. DOI: `10.1145/2622669`.

Maloney, John H, Smith, Randall B (1995). "Directness and liveness in the morphic user interface construction environment". *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*. New York, NY, USA: ACM, pp. 21–28. DOI: `10.1145/215585.215636`.

Mandelin, David, Xu, Lin, Bodík, Rastislav, Kimelman, Doug (2005). "Jungloid mining: helping to navigate the API jungle". *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, pp. 48–61. DOI: `10.1145/1065010.1065018`.

Mayer, Richard E (1997). "Multimedia learning: Are we asking the right questions". *Educational Psychologist* 32.1, pp. 1–19. DOI: `10.1207/s15326985ep3201_1`.

Mesbah, Ali, Van Deursen, Arie (2009). "Invariant-based automatic testing of AJAX user interfaces". *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, pp. 210–220. DOI: `10.1109/ICSE.2009.5070522`.

Meyer, Bertrand (1997). *Object-oriented Software Construction*. 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall. ISBN: 978-0136291558.

Miara, Richard J, Musselman, Joyce A, Navarro, Juan A, Shneiderman, Ben (1983). "Program indentation and comprehensibility". *Communications of the ACM* 26.11, pp. 861–867. DOI: `10.1145/182.358437`.

Mills, Carol Bergfeld, Diehl, Virginia A, Birkmire, Deborah P, Mou, Lien-Chong (1995). "Reading procedural texts: Effects of purpose for reading and predictions of reading comprehension models". *Discourse Processes* 20.1, pp. 79–107. DOI: `10.1080/01638539509544932`.

Murphy, Gail C, Kersten, Mik, Findlater, Leah (2006). "How Are Java Software Developers Using the Eclipse IDE?" *IEEE Software* 23.4, pp. 76–83. DOI: `10.1109/MS.2006.105`.

Myers, Brad A. (1990). “Taxonomies of visual programming and program visualization”. *Journal of Visual Languages and Computing* 1.1, pp. 97–123. DOI: `10.1016/S1045-926X(05)80036-9`.

Myers, Brad A., Pane, John F, Ko, Andy (2004). “Natural programming languages and environments”. *Communications of the ACM* 47.9, pp. 47–52. DOI: `10.1145/1015864.1015888`.

Oney, Stephen, Brandt, Joel (2012a). “Codelets: Linking Interactive Documentation and Example Code in the Editor”. *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 2697–2706. DOI: `10.1145/2207676.2208664`.

Oney, Stephen, Myers, Brad, Brandt, Joel (2012b). “ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States”. *UIST '12: Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, pp. 229–238. DOI: `10.1145/2380116.2380146`.

Pair, C (2011). “Programming, Programming Languages and Programming Methods”. *Psychology of Programming*. Ed. by J M Hoc, T R G Green, R Samurçay, D J Gilmore, pp. 9–19.

Parent, Rick (2012). *Computer Animation: Algorithms and Techniques*. 3rd ed. Algorithms and Techniques. Morgan Kaufmann. ISBN: 978-0124158429.

Paulson, Linda Dailey (2007). “Developers shift to dynamic programming languages”. *Computer* 40.2, pp. 12–15. DOI: `10.1109/MC.2007.53`.

Pavel, Amy, Goldman, Dan B, Hartmann, Björn, Agrawala, Maneesh (2015). “SceneSkim”. *UIST '15: Proceedings of the 28th annual ACM symposium on User Interface Software and Technology*. New York, NY, USA: ACM, pp. 181–190. DOI: `10.1145/2807442.2807502`.

Pennington, Nancy (1987). “Stimulus structures and mental representations in expert comprehension of computer programs”. *Cognitive Psychology* 19.3, pp. 295–341. DOI: `10.1016/0010-0285(87)90007-7`.

Petre, Marian (1995). “Why looking isn't always seeing: readership skills and graphical programming”. *Communications of the ACM* 38.6, pp. 33–44. DOI: `10.1145/203241.203251`.

Piorkowski, D, Fleming, S D, Scaffidi, C, John, L, Bogart, C, John, B E, Burnett, M, Bellamy, R (2011). "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models". *VL/HCC '11: IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 109–116. DOI: `10.1109/VLHCC.2011.6070387`.

Pirolli, Peter, Card, Stuart (1999). "Information foraging". *Psychological Review* 106.4, pp. 643–675. DOI: `10.1037/0033-295X.106.4.643`.

Pressman, Roger S, Maxim, Bruce R (2014). *Software Engineering: A Practitioner's Approach*. 8th ed. McGraw-Hill Education. ISBN: 978-0078022128.

Resnick, Mitchel, Maloney, John, Monroy-Hernández, Andrés, Rusk, Natalie, Eastmond, Evelyn, Brennan, Karen, Millner, Amon, Rosenbaum, Eric, Silver, Jay, Silverman, Brian, Kafai, Yasmin (2009). "Scratch". *Communications of the ACM* 52.11, pp. 60–67. DOI: `10.1145/1592761.1592779`.

Richards, Gregor, Lebresne, Sylvain, Burg, Brian, Vitek, Jan (2010). "An Analysis of the Dynamic Behavior of JavaScript Programs". *PLDI '10: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, pp. 1–12. DOI: `10.1145/1806596.1806598`.

Rist, Robert S (1996). "System structure and design". *Empirical Studies of Programmers, Sixth Workshop*, pp. 163–194.

Robillard, Martin P. (2009). "What Makes APIs Hard to Learn? Answers from Developers". *IEEE Software* 26.6, pp. 27–34. DOI: `10.1109/MS.2009.193`.

Robillard, Martin P., Coelho, Wesley, Murphy, Gail C. (2004). "How Effective Developers Investigate Source Code: An Exploratory Study". *IEEE Transactions on Software Engineering* 30.12, pp. 889–903. DOI: `10.1109/TSE.2004.101`.

Rothermel, Karen J., Cook, Curtis R., Burnett, Margaret M., Schonfeld, Justin, Green, T. R. G., Rothermel, Gregg (2000). "WYSIWYT Testing in the Spreadsheet Paradigm: An Empirical Evaluation". *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*. New York, NY, USA: ACM, pp. 230–239. DOI: `10.1145/337180.337206`.

Sackman, H, Erikson, W J, Grant, E E (1968). "Exploratory experimental studies comparing online and offline programming performance". *Communications of the ACM* 11.1, pp. 3–11. DOI: `10.1145/362851.362858`.

Saff, David, Ernst, Michael D (2003). “Reducing wasted development time via continuous testing”. *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE, pp. 281–292. DOI: `10.1109/ISSRE.2003.1251050`.

Saff, David, Ernst, Michael D (2004). “An experimental evaluation of continuous testing during development”. *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, pp. 76–85. DOI: `10.1145/1007512.1007523`.

Seaman, Carolyn B (1999). “Qualitative Methods in Empirical Studies of Software Engineering”. *IEEE Transactions on Software Engineering* 25.4, pp. 557–572. DOI: `10.1109/32.799955`.

Shneiderman, Ben (1983). “Direct Manipulation: A Step Beyond Programming Languages”. *Computer* 16.8, pp. 57–69. DOI: `10.1109/MC.1983.1654471`.

Shneiderman, Ben, Mayer, Richard (1979). “Syntactic/semantic interactions in programmer behavior: A model and experimental results”. *International Journal of Computer & Information Sciences* 8.3, pp. 219–238. DOI: `10.1007/BF00977789`.

Sillito, Jonathan, Murphy, Gail C, De Volder, Kris (2008). “Asking and Answering Questions during a Programming Change Task”. *IEEE Transactions on Software Engineering* 34.4, pp. 434–451. DOI: `10.1109/TSE.2008.26`.

Singer, Janice, Elves, Robert, Storey, Margaret-Anne (2005). “NavTracks: supporting navigation in software maintenance”. *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE, pp. 325–334. DOI: `10.1109/ICSM.2005.66`.

Smith, Randall B, Ungar, David (1995). “Programming as an Experience: The Inspiration for Self”. *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*. Springer Berlin Heidelberg. DOI: `10.1007/3-540-49538-X_15`.

Snell, James L (1997). “Ahead-of-time Debugging, or Programming Not in the Dark”. *STEP '97: Proceedings of the 8th International Workshop on Software Technology and Engineering Practice*. Washington, DC, USA: IEEE, pp. 288–293. DOI: `10.1109/STEP.1997.615516`.

Soloway, Elliot, Ehrlich, Kate (1984). “Empirical Studies of Programming Knowledge”. *IEEE Transactions on Software Engineering* 10.5, pp. 595–609. DOI: `10.1109/TSE.1984.5010283`.

Squire, Megan, Smith, Amber K (2015). "The Diffusion of Pastebin Tools to Enhance Communication in FLOSS Mailing Lists". *OSS '15: 11th International Conference on Open Source Systems: Adoption and Impact*. Springer International Publishing, pp. 45–57. DOI: 10.1007/978-3-319-17837-0_5.

Sridhara, Giriprasad, Hill, Emily, Muppaneni, Divya, Pollock, Lori, Vijay-Shanker, K. (2010). "Towards Automatically Generating Summary Comments for Java Methods". *ASE '10: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, pp. 43–52. DOI: 10.1145/1858996.1859006.

Starke, Jamie, Luce, Chris, Sillito, Jonathan (2009). "Searching and skimming: An exploratory study". *ICSM '09: IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE, pp. 157–166. DOI: 10.1109/ICSM.2009.5306335.

Stylos, Jeffrey, Graf, Benjamin, Busse, Daniela K, Ziegler, Carsten, Ehret, Ralf, Karstens, Jan (2008). "A case study of API redesign for improved usability". *VL-HCC '08: Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE, pp. 189–192. DOI: 10.1109/VLHCC.2008.4639083.

Tan, Shin Hwei, Marinov, Darko, Tan, Lin, Leavens, Gary T (2012). "@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies". *ICST '12: IEEE Fifth International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE, pp. 260–269. DOI: 10.1109/ICST.2012.106.

Tanimoto, Steven L. (1990). "VIVA: A visual language for image processing". *Journal of Visual Languages and Computing* 1.2, pp. 127–139. DOI: 10.1016/S1045-926X(05)80012-6.

Tanimoto, Steven L. (2013). "A Perspective on the Evolution of Live Programming". *LIVE '13: Proceedings of the 1st International Workshop on Live Programming*. Piscataway, NJ, USA: IEEE Press, pp. 31–34. DOI: 10.1109/LIVE.2013.6617346.

Taylor, M J, McWilliam, J, Forsyth, H, Wade, S (2002). "Methodologies and website development: a survey of practice". *Information and Software Technology* 44.6, pp. 381–391. DOI: 10.1016/S0950-5849(02)00024-1.

Ulmen, Tanja (2014). "Combining Live Coding and Continuous Testing". Bachelor's Thesis. Aachen: RWTH Aachen University.

Vans, A Marie, Mayrhauser, Anneliese von, Somlo, Gabriel (1999). "Program understanding behavior during corrective maintenance of large-scale software". *International Journal of Human-Computer Studies* 51.1, pp. 31–70. DOI: `10.1006/ijhc.1999.0268`.

Vessey, Iris (1989). "Toward a theory of computer program bugs: an empirical test". *International Journal of Man-Machine Studies* 30.1, pp. 23–46. DOI: `10.1016/S0020-7373(89)80019-7`.

Weiser, Mark (1982). "Programmers use slices when debugging". *Communications of the ACM* 25.7, pp. 446–452. DOI: `10.1145/358557.358577`.

Wilcox, E M, Atwood, J W, Burnett, M M, Cadiz, J J, Cook, C R (1997). "Does continuous visual feedback aid debugging in direct-manipulation programming systems?" *CHI '97: Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. New York, NY, USA: ACM, pp. 258–265. DOI: `10.1145/258549.258721`.

Winograd, Terry (1979). "Beyond Programming Languages". *Communications of the ACM* 22.7, pp. 391–401. DOI: `10.1145/359131.359133`.

Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall. ISBN: 978-0130224187.

Wolf, Hendrik (2014). "Detail Visualization for Live Coding". Bachelor's Thesis. RWTH Aachen University.

Xie, Tao, Notkin, David (2006). "Tool-assisted unit-test generation and selection based on operational abstractions". *Automated Software Engineering* 13.3, pp. 345–371. DOI: `10.1007/s10851-006-8530-6`.

Own Publications

Papers (Peer-reviewed, archival)

Karrer, Thorsten, **Krämer, Jan-Peter**, Diehl, Jonathan, Hartmann, Björn, Borchers, Jan (2011). "Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency". *UIST '11: Proceedings of the 24th annual ACM symposium on User Interface Software and Technology*. New York, NY, USA: ACM, pp. 217–224. DOI: 10.1145/2047196.2047225.

Krämer, Jan-Peter, Brandt, Joel, Borchers, Jan (2016a). "Using Runtime Traces to Improve Documentation and Unit Test Authoring for Dynamic Languages". *CHI '16: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 3232–3237. DOI: 10.1145/2858036.2858311.

Krämer, Jan-Peter, Karrer, Thorsten, Kurz, Joachim, Wittenhagen, Moritz, Borchers, Jan (2013). "How Tools in IDEs Shape Developers' Navigation Behavior". *CHI '13: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 3073–3082. DOI: 10.1145/2470654.2466419.

Krämer, Jan-Peter, Kurz, Joachim, Karrer, Thorsten, Borchers, Jan (2014). "How live coding affects developers' coding behavior". *VL/HCC '14: IEEE Symposium on Visual Languages and Human-Centric Computing*. Washington, DC, USA: IEEE, pp. 5–8. DOI: 10.1109/VLHCC.2014.6883013.

Posters (Peer-reviewed, non-archival)

Diehl, Jonathan, **Krämer, Jan-Peter**, Borchers, Jan (2008). "A Framework for using the iPhone as a Wireless Input Device for Interactive Systems". *UIST '08 EA:*

Extended Abstracts of the 21st Annual ACM symposium on User Interface Software and Technology. New York, NY, USA: ACM.

Krämer, Jan-Peter (2010). "PIM-Mail: Consolidating Task and Email Management". *CHI '10 EA: CHI '10 Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 4411–4416. DOI: 10.1145/1753846.1754162.

Krämer, Jan-Peter, Karrer, Thorsten, Diehl, Jonathan, Borchers, Jan (2010). "Stacksplorer: Understanding Dynamic Program Behavior". *UIST '10: Adjunct Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology*. New York, NY, USA: ACM, pp. 433–434. DOI: 10.1145/1866218.1866257.

Krämer, Jan-Peter, Kurz, Joachim, Karrer, Thorsten, Borchers, Jan (2012b). "Blaze: Supporting Two-phased Call Graph Navigation in Source Code". *CHI EA '12: CHI '12 Extended Abstracts on Human Factors in Computing Systems*. New York, NY, USA: ACM, pp. 2195–2200. DOI: 10.1145/2212776.2223775.

Demos (Juried, non-archival)

Krämer, Jan-Peter, Kurz, Joachim, Karrer, Thorsten, Borchers, Jan (2012a). "Blaze". *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, pp. 1457–1458. DOI: 10.1109/ICSE.2012.6227066.

Workshop papers (Juried, non-archival)

Krämer, Jan-Peter, Hennings, Michael, Brandt, Joel, Borchers, Jan (2016b). "An Empirical Study of Programming Paradigms for Animation". *CHASE '16: Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. New York, NY, USA: ACM, pp. 58–61. DOI: 10.1145/2897586.2897597.

Theses

Krämer, Jan-Peter (2011). "Stacksplorer Understanding Dynamic Program Behavior". Diploma Thesis. RWTH Aachen University.

Index

Typeset January 15, 2017